Contents

- Welcome
- .NET Guide
  - Get Started with .NET
  - Tour of .NET
  - .NET Architectural Components
  - .NET Standard
    - What's new in the .NET Standard
  - Target Frameworks
  - .NET Glossary
  - Open-source Library Guidance
  - Choosing between .NET Core and .NET Framework for server apps
  - What is "managed code"?
    - Automatic Memory Management
  - Common Language Runtime (CLR)
  - Language Independence
    - Language Independence and Language-Independent Components
  - Framework Libraries
    - Class Library Overview
    - Base Types
  - .NET Class libraries
  - Analyzers
    - API Analyzer
    - Portability Analyzer
    - Framework Analyzer
  - Handling and throwing exceptions
  - Assemblies in .NET
  - Garbage Collection
  - Generic types
  - Delegates and lambdas

- LINQ
- Common Type System & Common Language Specification
- Parallel Processing, Concurrency, and Async
  - Asynchronous programming
    - Asynchronous programming in depth
    - Asynchronous Programming Patterns
  - Parallel Programming
  - Threading
- Memory and span-related types
  - Memory<T> and Span<T> usage guidelines
- Native interoperability
  - P/Invoke
  - Type marshaling
  - Customizing structure marshaling
  - Customizing parameter marshaling
  - Interop guidance
  - Charsets and marshaling
  - COM Interop
    - COM Wrappers
      - Runtime Callable Wrapper
      - COM Callable Wrapper
    - Qualifying .NET types for COM interoperation
  - Applying Interop Attributes
- Collections and Data Structures
- Numerics in .NET
- Dates, times, and time zones
- Events
- Managed Execution Process
- Metadata and Self-Describing Components
- Building Console Applications
- Application Essentials
- File and Stream I/O

- Globalization and Localization
- Attributes
- Framework Design Guidelines
- XML Documents and Data
- Security
- Serialization
- Developing for Multiple Platforms

.NET Core Guide

- About .NET Core
- Get started
  - Get started with C# and Visual Studio Code
  - Build a C# Hello World app with .NET Core in Visual Studio 2017
  - Build a Visual Basic Hello World app with .NET Core in Visual Studio 2017
  - Create a .NET Core application with plugins
  - Debug your C# or Visual Basic .NET Core Hello World application using Visual Studio 2017
  - Publish your Hello World application with Visual Studio 2017
  - Build a .NET Standard library with C# and .NET Core in Visual Studio 2017
  - Build a .NET Standard library with Visual Basic and .NET Core in Visual Studio 2017
  - Test a .NET Standard library with .NET Core in Visual Studio 2017
  - Consume a .NET Standard library in Visual Studio 2017
- Windows Prerequisites
- macOS Prerequisites
- Linux Prerequisites
- What's new in .NET Core
  - What's new in .NET Core 3.0
  - What's new in .NET Core 2.2
  - What's new in .NET Core 2.1
  - What's new in .NET Core 2.0
- Application Compatibility
  - Evaluate breaking changes
  - Breaking change categories
- Tutorials

- Templates for the CLI
  - 1 - Create an item template
  - 2 - Create a project template
  - 3 - Create a template pack
- Get started with .NET Core on macOS
- Get started with .NET Core on macOS using Visual Studio for Mac
- Building a complete .NET Core solution on macOS using Visual Studio for Mac
- Get started with .NET Core using the CLI tools
- Organizing and testing projects with the .NET Core command line
- Developing Libraries with Cross Platform Tools
- Developing ASP.NET Core applications
- Hosting .NET Core from native code

Native Interoperability
- Exposing .NET Core Components to COM

Packages, Metapackages and Frameworks

Changes in CLI overview
- Dependency management
- Additions to the csproj format

Migration
- .NET Core 2.0 to 2.1
- Migrating from project.json
- Mapping between project.json and csproj
- Migrating from DNX

Application Deployment
- Deploy apps with CLI tools
- Deploy apps with Visual Studio
- Creating a NuGet Package with Cross Platform Tools
- Self-contained deployment runtime roll forward
- Runtime package store

Docker
- Introduction to .NET and Docker
- Containerize a .NET Core app

- Container Tools in Visual Studio
- Diagnostic tools
  - Overview
  - Managed debuggers
  - Logging and tracing
- Unit Testing
  - Unit testing best practices
  - C# unit testing with xUnit
  - C# unit testing with NUnit
  - C# unit testing with MSTest
  - F# unit testing with xUnit
  - F# unit testing with NUnit
  - F# unit testing with MSTest
  - VB unit testing with xUnit
  - VB unit testing with NUnit
  - VB unit testing with MSTest
  - Running selective unit tests
  - Unit testing published output
  - Live unit testing .NET Core projects with Visual Studio
- Continuous Integration
- Versioning
  - .NET Core version selection
  - Removing outdated runtimes and SDKs
- Runtime Identifier (RID) catalog
- .NET Core SDK Overview
- .NET Core CLI
  - Overview
  - Global Tools
    - Overview
    - Create a Global Tool
  - Elevated access
  - Extensibility Model

- Custom templates
- Enable TAB completion
- Telemetry
- global.json overview
- Reference
  - dotnet
  - dotnet build
  - dotnet build-server
  - dotnet clean
  - dotnet help
  - dotnet migrate
  - dotnet msbuild
  - dotnet new
  - dotnet nuget
    - dotnet nuget delete
    - dotnet nuget locals
    - dotnet nuget push
  - dotnet pack
  - dotnet publish
  - dotnet restore
  - dotnet run
  - dotnet sln
  - dotnet store
  - dotnet test
  - dotnet tool
    - dotnet tool install
    - dotnet tool list
    - dotnet tool uninstall
    - dotnet tool update
  - dotnet vstest
  - dotnet-install scripts
  - Project reference commands

- dotnet add reference
- dotnet list reference
- dotnet remove reference
- Project package commands
- dotnet add package
- dotnet list package
- dotnet remove package

.NET Core Additional Tools
- WCF Web Service Reference Provider
- dotnet-svcutil
- dotnet-svcutil.xmlserializer
- XML Serializer Generator

Porting from .NET Framework
- .NET Framework technologies unavailable for .NET Core
- Analyzing third-party dependencies
- Porting libraries
- Organizing projects for .NET Core
- Tools to help with porting to .NET Core
- Using the Windows Compatibility Pack
- Port Windows Forms projects
- Port WPF projects

Dependency loading
- Overview
- Understanding AssemblyLoadContext
- Loading details
- Default dependency probing
- Loading managed assemblies
- Loading satellite assemblies
- Loading unmanaged libraries
- Tutorials
- Create a .NET Core application with plugins
- How to use and debug assembly unloadability in .NET Core

- Build .NET Core from source
  - .NET Core distribution packaging
- VS 2015/project.json docs

.NET Framework Guide

- What's New
- Get Started
- Installation guide
- Migration Guide
- Development Guide
  - Application Domains and Assemblies
  - Resources in Desktop Apps
  - Accessibility
  - Data and Modeling
  - Client Applications
    - Windows Presentation Foundation
    - Windows Forms
  - Service-Oriented Applications with WCF
  - Windows Workflow Foundation
  - Windows Service Applications
  - 64-bit Applications
  - Web Applications with ASP.NET
  - Network Programming in the .NET Framework
  - Configuring Apps
  - Compiling Apps with .NET Native
  - Windows Identity Foundation
  - Debugging, Tracing, and Profiling
  - Deployment
  - Performance
  - Dynamic Programming
  - Managed Extensibility Framework (MEF)
  - Interoperating with Unmanaged Code
  - Unmanaged API Reference

- XAML Services
- Tools
- Additional Class Libraries and APIs

C# Guide

- Get Started
- Tutorials
  - Overview
  - Introduction to programming with C#
    - Choose your first lesson
    - Hello world
    - Numbers in C#
    - Branches and loops
    - List collections
    - Work in your local environment
      - Setup your environment
      - Numbers in C#
      - Branches and loops
      - List collections
      - Introduction to classes
  - Explore C# 6
  - Explore string interpolation - interactive
  - Explore string interpolation - in your environment
  - Advanced scenarios for string Interpolation
  - Safely update interfaces with default interface members
  - Explore indexes and ranges
  - Work with nullable reference types
  - Upgrade an app to nullable reference types
  - Generate and consume asynchronous streams
  - Extend data capabilities using pattern matching
  - Console Application
  - REST Client
  - Inheritance in C# and .NET

- Working with LINQ
- Using Attributes
- Tour of C#
- What's new in C#
  - C# 8.0 - Preview 5
  - C# 7.3
  - C# 7.2
  - C# 7.1
  - C# 7.0
  - C# 6
  - C# Version History
  - Relationships between language and framework
  - Version and update considerations
- C# Concepts
  - C# Type system
  - Nullable reference types
  - Describe nullable APIs using attributes and constraints
  - Namespaces
  - Basic Types
  - Classes
  - Structs
  - Tuples
  - Deconstructing tuples and other types
  - Interfaces
  - Methods
  - Lambda Expressions
  - Properties
  - Indexers
  - Discards
  - Generics
  - Iterators
  - Delegates & events

- Introduction to Delegates
- System.Delegate and the delegate keyword
- Strongly Typed Delegates
- Common Patterns for Delegates
- Introduction to Events
- Standard .NET event patterns
- The Updated .NET Event Pattern
- Distinguishing Delegates and Events
- Language Integrated Query (LINQ)
- Asynchronous programming
- Pattern Matching
- Write safe, efficient code
- Expression Trees
  - Introduction to expression trees
  - Expression Trees Explained
  - Framework Types Supporting Expression Trees
  - Executing Expressions
  - Interpreting Expressions
  - Building Expressions
  - Translating Expressions
  - Summary
- Native interoperability
- Documenting your code
- Versioning

How To C# Topics

- Topic index
- Parse strings using `String.Split`
- Concatenate strings
- Convert a string to a DateTime
- Search strings
- Modify string contents
- Compare strings

- Safely cast using pattern matching, is and as operators
- The .NET Compiler Platform SDK (Roslyn APIs)
- C# Programming Guide
- Language Reference
- Walkthroughs
- F# Guide
- Get Started
  - Install F#
  - Get Started with F# in Visual Studio
  - Get Started with F# in Visual Studio for Mac
  - Get Started with F# in Visual Studio Code and Ionide
  - Get Started with F# with the .NET Core CLI
- What is F#
- Tour of F#
- Tutorials
  - Introduction to Functional Programming
    - First-class functions
  - Asynchronous and Concurrent Programming
    - Asynchronous Programming
  - Type Providers
    - Create a Type Provider
    - Type provider Security
    - Troubleshooting Type Providers
  - F# Interactive
- F# Language Reference
  - Keyword Reference
  - Symbol and Operator Reference
    - Arithmetic Operators
    - Boolean Operators
    - Bitwise Operators
    - Nullable Operators
  - Functions

- let Bindings
- do Bindings
- Lambda Expressions: the fun keyword
- Recursive Functions: the rec keyword
- Entry Point
- External Functions
- Inline Functions

Values
- Null Values

Literals

F# Types

Type Inference

Basic Types

Unit Type

Strings

Tuples

F# Collection Types

Lists

Arrays

Sequences

Slices

Options

Value Options

Results

Generics
- Automatic Generalization
- Constraints
- Statically Resolved Type Parameters

Records

Anonymous Records

Copy and Update Record Expressions

Discriminated Unions

Enumerations
Type Abbreviations
Classes
Structures
Inheritance
Interfaces
Abstract Classes
Members
  let Bindings in Classes
  do Bindings in Classes
  Properties
  Indexed Properties
  Methods
  Constructors
  Events
  Explicit Fields: The `val` Keyword
Type Extensions
Parameters and Arguments
Operator Overloading
Flexible Types
Delegates
Object Expressions
Casting and Conversions
Access Control
Conditional Expressions: if...then...else
Match Expressions
Pattern Matching
Active Patterns
Loops: for...to Expression
Loops: for...in Expression
Loops: while...do Expression
Assertions

- Exception Handling
  - Exception Types
  - The try...with Expression
  - The try...finally Expression
  - The raise Function
  - The failwith Function
  - The invalidArg Function
- Attributes
- Resource Management: the use Keyword
- Namespaces
- Modules
- Import Declarations: The open Keyword
- Signature Files
- Units of Measure
- XML Documentation
- Lazy Expressions
- Computation Expressions
- Asynchronous Workflows
- Query Expressions
- Code Quotations
- Fixed keyword
- Byrefs
- Reference Cells
- Compiler Directives
- Compiler Options
- F# Interactive Options
- Source Line, File, and Path Identifiers
- Caller Information
- Verbose Syntax

F# style guide

- F# code formatting guidelines
- F# coding conventions

- F# component design guidelines
- Using F# on Azure
  - Get started with Azure Blob storage using F#
  - Get started with Azure File storage using F#
  - Get started with Azure Queue storage using F#
  - Get started with Azure Table storage using F#
  - Package Management for F# Azure Dependencies
- Visual Basic Guide
  - What's New for Visual Basic
  - Get Started
    - Visual Basic Breaking Changes in Visual Studio
    - Additional Resources for Visual Basic Programmers
  - Developing Applications
    - Programming in Visual Basic
      - Accessing Computer Resources
      - Logging Information from the Application
      - Accessing User Data
      - Accessing Application Forms
      - Accessing Application Web Services
        - How to: Call a Web Service Asynchronously
      - Accessing Application Settings
      - Processing Drives, Directories, and Files
    - Development with My
      - Performing Tasks with My.Application, My.Computer, and My.User
      - Default Object Instances Provided by My.Forms and My.WebServices
      - Rapid Application Development with My.Resources and My.Settings
      - Overview of the Visual Basic Application Model
      - How My Depends on Project Type
    - Accessing Data
    - Creating and Using Components
    - Windows Forms Application Basics
    - Customizing Projects and Extending My with Visual Basic

- Extending the My Namespace
- Packaging and Deploying Custom My Extensions
- Extending the Visual Basic Application Model
- Customizing Which Objects are Available in My
- Programming Concepts
  - Assemblies
    - Friend Assemblies
      - How to: Create Unsigned Friend Assemblies
      - How to: Create Signed Friend Assemblies
    - How to: Create and Use Assemblies Using the Command Line
    - How to: Determine If a File Is an Assembly
    - How to: Load and Unload Assemblies
    - How to: Share an Assembly with Other Applications
    - Walkthrough: Embedding Types from Managed Assemblies
  - Asynchronous Programming with Async and Await
  - Attributes
  - Caller Information
  - Collections
  - Covariance and Contravariance
  - Expression Trees
  - Iterators
  - Language-Integrated Query (LINQ)
  - Object-Oriented Programming
  - Reflection
  - Serialization
- Program Structure and Code Conventions
  - Structure of a Program
  - Main Procedure
  - References and the Imports Statement
  - Namespaces
  - Naming Conventions
  - Coding Conventions

- Conditional Compilation
- How to: Break and Combine Statements in Code
- How to: Collapse and Hide Sections of Code
- How to: Label Statements
- Special Characters in Code
- Comments in Code
- Keywords as Element Names in Code
- Me, My, MyBase, and MyClass
- Limitations

Language Features

- Arrays
- Collection Initializers
- Constants and Enumerations
- Control Flow
- Data Types
  - Type Characters
  - Elementary Data Types
    - Numeric Data Types
    - Character Data Types
    - Miscellaneous Data Types
  - Composite Data Types
    - How to: Hold More Than One Value in a Variable
  - Generic Types
    - How to: Define a Class That Can Provide Identical Functionality on Different Data Types
    - How to: Use a Generic Class
  - Generic Procedures
  - Nullable Value Types
  - Value Types and Reference Types
  - Type Conversions
    - Widening and Narrowing Conversions
    - Implicit and Explicit Conversions
    - Conversions Between Strings and Other Types

- How to: Convert an Object to Another Type
- Array Conversions
- Structures
  - How to: Declare a Structure
  - Structure Variables
  - Structures and Other Programming Elements
  - Structures and Classes
- Tuples
- Efficient Use of Data Types
- Troubleshooting Data Types
- Declared Elements
- Delegates
- Early and Late Binding
- Error Types
- Events
- Interfaces
  - Walkthrough: Creating and Implementing Interfaces
- LINQ
- Objects and Classes
- Operators and Expressions
- Procedures
- Statements
- Strings
- Variables
- XML

COM Interop

- Introduction to COM Interop
- How to: Reference COM Objects
- How to: Work with ActiveX Controls
- Walkthrough: Calling Windows APIs
- How to: Call Windows APIs
- How to: Call a Windows Function that Takes Unsigned Types

- Walkthrough: Creating COM Objects
- Troubleshooting Interoperability
- COM Interoperability in .NET Framework Applications
- Walkthrough: Implementing Inheritance with COM Objects

Language Reference

- Configure language version
- Typographic and Code Conventions
- Visual Basic Runtime Library Members
- Keywords
  - Arrays Summary
  - Collection Object Summary
  - Control Flow Summary
  - Conversion Summary
  - Data Types Summary
  - Dates and Times Summary
  - Declarations and Constants Summary
  - Directories and Files Summary
  - Errors Summary
  - Financial Summary
  - Information and Interaction Summary
  - Input and Output Summary
  - Math Summary
    - Derived Math Functions
  - My Reference
  - Operators Summary
  - Registry Summary
  - String Manipulation Summary
- Attributes
- Constants and Enumerations
- Data Type Summary
  - Boolean Data Type
  - Byte Data Type

- Char Data Type
- Date Data Type
- Decimal Data Type
- Double Data Type
- Integer Data Type
- Long Data Type
- Object Data Type
- SByte Data Type
- Short Data Type
- Single Data Type
- String Data Type
- UInteger Data Type
- ULong Data Type
- User-Defined Data Type
- UShort Data Type

Directives

- #Const Directive
- #ExternalSource Directive
- #If...Then...#Else Directives
- #Region Directive

Functions

- Conversion Functions
- Math Functions
- String Functions
- Type Conversion Functions
  - Return Values for the CStr Function
- CType Function

Modifiers

- Ansi
- Assembly
- Async
- Auto

ByRef

ByVal

Default

Friend

In (Generic Modifier)

Iterator

Key

Module

MustInherit

MustOverride

Narrowing

NotInheritable

NotOverridable

Optional

Out (Generic Modifier)

Overloads

Overridable

Overrides

ParamArray

Partial

Private

Private Protected

Protected

Protected Friend

Public

ReadOnly

Shadows

Shared

Static

Unicode

Widening

WithEvents

- WriteOnly
- Modules
- Nothing
- Objects
  - My.Application Object
    - My.Application.Info Object
    - My.Application.Log Object
  - My.Computer Object
    - My.Computer.Audio Object
    - My.Computer.Clipboard Object
    - My.Computer.Clock Object
    - My.Computer.FileSystem Object
      - My.Computer.FileSystem.SpecialDirectories Object
    - My.Computer.Info Object
    - My.Computer.Keyboard Object
    - My.Computer.Mouse Object
    - My.Computer.Network Object
    - My.Computer.Ports Object
    - My.Computer.Registry Object
  - My.Forms Object
  - My.Log Object
  - My.Request Object
  - My.Response Object
  - My.Resources Object
  - My.Settings Object
  - My.User Object
  - My.WebServices Object
  - TextFieldParser Object
- Operators
  - Operator Precedence
  - Data Types of Operator Results
  - Operators Listed by Functionality

Arithmetic Operators
Assignment Operators
Bit Shift Operators
Comparison Operators
Concatenation Operators
Logical-Bitwise Operators
Miscellaneous Operators
List of Visual Basic Operators
& Operator
&= Operator
* Operator
*= Operator
+ Operator
+= Operator
= Operator
- Operator
-= Operator
< Operator
<= Operator
> Operator
>= Operator
<< Operator
<<= Operator
>> Operator
>>= Operator
/ Operator
/= Operator
\ Operator
\= Operator
^ Operator
^= Operator
?() Operator

?. Operator
AddressOf Operator
And Operator
AndAlso Operator
Await Operator
DirectCast Operator
Function Expression
GetType Operator
GetXmlNamespace Operator
If Operator
Is Operator
IsFalse Operator
IsNot Operator
IsTrue Operator
Like Operator
Mod Operator
New Operator
Not Operator
Or Operator
OrElse Operator
Sub Expression
TryCast Operator
TypeOf Operator
Xor Operator
Properties
Queries
Aggregate Clause
Distinct Clause
Equals Clause
From Clause
Group By Clause
Group Join Clause

- Join Clause
- Let Clause
- Order By Clause
- Select Clause
- Skip Clause
- Skip While Clause
- Take Clause
- Take While Clause
- Where Clause

Statements

- A-E Statements
  - AddHandler Statement
  - Call Statement
  - Class Statement
  - Const Statement
  - Continue Statement
  - Declare Statement
  - Delegate Statement
  - Dim Statement
  - Do...Loop Statement
  - Else Statement
  - End Statement
  - End <keyword> Statement
  - Enum Statement
  - Erase Statement
  - Error Statement
  - Event Statement
  - Exit Statement
- F-P Statements
  - For Each...Next Statement
  - For...Next Statement
  - Function Statement

- Get Statement
- GoTo Statement
- If...Then...Else Statement
- Implements Statement
- Imports Statement (.NET Namespace and Type)
- Imports Statement (XML Namespace)
- Inherits Statement
- Interface Statement
- Mid Statement
- Module Statement
- Namespace Statement
- On Error Statement
- Operator Statement
- Option <keyword> Statement
- Option Compare Statement
- Option Explicit Statement
- Option Infer Statement
- Option Strict Statement
- Property Statement

Q-Z Statements

- RaiseEvent Statement
- ReDim Statement
- REM Statement
- RemoveHandler Statement
- Resume Statement
- Return Statement
- Select...Case Statement
- Set Statement
- Stop Statement
- Structure Statement
- Sub Statement
- SyncLock Statement

Then Statement
Throw Statement
Try...Catch...Finally Statement
Using Statement
While...End While Statement
With...End With Statement
Yield Statement
Clauses
Alias Clause
As Clause
Handles Clause
Implements Clause
In Clause
Into Clause
Of Clause
Declaration Contexts and Default Access Levels
Attribute List
Parameter List
Type List
XML Comment Tags
<c>
<code>
<example>
<exception>
<include>
<list>
<para>
<param>
<paramref>
<permission>
<remarks>
<returns>

<see>

<seealso>

<summary>

<typeparam>

<value>

XML Axis Properties

XML Attribute Axis Property

XML Child Axis Property

XML Descendant Axis Property

Extension Indexer Property

XML Value Property

XML Literals

XML Element Literal

XML Document Literal

XML CDATA Literal

XML Comment Literal

XML Processing Instruction Literal

Error Messages

'#ElseIf' must be preceded by a matching '#If' or '#ElseIf'

'#Region' and '#End Region' statements are not valid within method bodies-multiline lambdas

'<attribute>' cannot be applied because the format of the GUID '<number>' is not correct

'<classname>' is not CLS-compliant because the interface '<interfacename>' it implements is not CLS-compliant

'<elementname>' is obsolete (Visual Basic Warning)

'<eventname>' is an event, and cannot be called directly

'<expression>' cannot be used as a type constraint

'<functionname>' is not declared (Smart Device-Visual Basic Compiler Error)

'<interfacename>.<membername>' is already implemented by the base class '<baseclassname>'. Re-implementation of <type> assumed

'<keyword>' is valid only within an instance method

'<membername>' cannot expose type '<typename>' outside the project through <containertype> '<containertypename>'

'<membername>' is ambiguous across the inherited interfaces '<interfacename1>' and '<interfacename2>'

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>'

'<methodname>' has multiple definitions with identical signatures

'<name>' is ambiguous in the namespace '<namespacename>'

'<name1>' is ambiguous, imported from the namespaces or types '<name2>'

<proceduresignature1> is not CLS-compliant because it overloads <proceduresignature2> which differs from it only by array of array parameter types or by the rank of the array parameter types

<type1>'<typename>' must implement '<membername>' for interface '<interfacename>'

<type1>'<typename>' must implement '<methodname>' for interface '<interfacename>'

'<typename>' cannot inherit from <type> '<basetypename>' because it expands the access of the base <type> outside the assembly

'<typename>' is a delegate type

'<typename>' is a type and cannot be used as an expression

A double quote is not a valid comment token for delimited fields where EscapeQuote is set to True

A property or method call cannot include a reference to a private object, either as an argument or as a return value

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'

A startup form has not been specified

Access of shared member through an instance; qualifying expression will not be evaluated

'AddressOf' operand must be the name of a method (without parentheses)

An unexpected error has occurred because an operating system resource required for single instance startup cannot be acquired

Anonymous type member name can be inferred only from a simple or qualified name with no arguments

Argument not optional

Array bounds cannot appear in type specifiers

Array declared as for loop control variable cannot be declared with an initial size

Array subscript expression missing

Arrays declared as structure members cannot be declared with an initial size

'As Any' is not supported in 'Declare' statements

Attribute '<attributename>' cannot be applied multiple times

Automation error

Bad checksum value, non hex digits or odd number of hex digits

Bad DLL calling convention

Bad file mode

Bad file name or number

Bad record length

Because this call is not awaited, the current method continues to run before the call is completed

Cannot convert anonymous type to expression tree because it contains a field that is used in the initialization of another field

Cannot create ActiveX Component

Cannot refer to '<name>' because it is a member of the value-typed field '<name>' of class '<classname>' which has 'System.MarshalByRefObject' as a base class

Cannot refer to an instance member of a class from within a shared method or shared member initializer without an explicit instance of the class

Can't create necessary temporary file

Can't open '<filename>' for writing

Class '<classname>' cannot be found

Class does not support Automation or does not support expected interface

'Class' statement must end with a matching 'End Class'

Clipboard format is not valid

Constant expression not representable in type '<typename>'

Constants must be of an intrinsic or enumerated type, not a class, structure, type parameter, or array type

Constructor '<name>' cannot call itself

Copying the value of 'ByRef' parameter '<parametername>' back to the matching argument narrows from type '<typename1>' to type '<typename2>'

'Custom' modifier is not valid on events declared without explicit delegate types

Data type(s) of the type parameter(s) cannot be inferred from these arguments

Declaration expected

Default property '<propertyname1>' conflicts with default property '<propertyname2>' in '<classname>' and so should be declared 'Shadows'

Default property access is ambiguous between the inherited interface members '<defaultpropertyname>' of interface '<interfacename1>' and '<defaultpropertyname>' of interface '<interfacename2>'

Delegate class '<classname>' has no Invoke method, so an expression of this type cannot be the target of a method call

Derived classes cannot raise base class events

Device I/O error

'Dir' function must first be called with a 'PathName' argument

End of statement expected

Error creating assembly manifest: <error message>

Error creating Win32 resources: <error message>

Error in loading DLL

Error saving temporary Win32 resource file '<filename>': <error message>

Errors occurred while compiling the XML schemas in the project

Evaluation of expression or statement timed out

Event '<eventname1>' cannot implement event '<eventname2>' on interface '<interface>' because their delegate types '<delegate1>' and '<delegate2>' do not match

Events cannot be declared with a delegate type that has a return type

Events of shared WithEvents variables cannot be handled by non-shared methods

Expression does not produce a value

Expression has the type '<typename>' which is a restricted type and cannot be used to access members inherited from 'Object' or 'ValueType'

Expression is a value and therefore cannot be the target of an assignment

Expression of type <type> is not queryable

Expression recursively calls the containing property '<propertyname>'

Expression too complex

'Extension' attribute can be applied only to 'Module', 'Sub', or 'Function' declarations

File already open

File is too large to read into a byte array

File name or class name not found during Automation operation

File not found (Visual Basic Run-Time Error)

First operand in a binary 'If' expression must be nullable or a reference type

First statement of this 'Sub New' must be a call to 'MyBase.New' or 'MyClass.New' (No Accessible Constructor Without Parameters)

First statement of this 'Sub New' must be an explicit call to 'MyBase.New' or 'MyClass.New' because the '<constructorname>' in the base class '<baseclassname>' of '<derivedclassname>' is marked obsolete: '<errormessage>'

'For Each' on type '<typename>' is ambiguous because the type implements multiple instantiations of 'System.Collections.Generic.IEnumerable(Of T)'

Friend assembly reference <reference> is invalid

Function '<procedurename>' doesn't return a value on all code paths

Function evaluation is disabled because a previous function evaluation timed out

Generic parameters used as optional parameter types must be class constrained

'Get' accessor of property '<propertyname>' is not accessible

Handles clause requires a WithEvents variable defined in the containing type or one of its base types

Identifier expected

Identifier is too long

Initializer expected

Input past end of file

Internal error happened at <location>

Implicit conversion from '<typename1>' to '<typename2>' in copying the value of 'ByRef' parameter '<parametername>' back to the matching argument.

'Is' requires operands that have reference types, but this operand has the value type '<typename>'

'IsNot' operand of type 'typename' can only be compared to 'Nothing', because 'typename' is a nullable type

Labels that are numbers must be followed by colons

Lambda expression will not be removed from this event handler

Lambda expressions are not valid in the first expression of a 'Select Case' statement

Late bound resolution; runtime errors could occur

Latebound overload resolution cannot be applied to '<procedurename>' because the accessing instance is an interface type

Leading '.' or '!' can only appear inside a 'With' statement

Line is too long

'Line' statements are no longer supported (Visual Basic Compiler Error)

Method does not have a signature compatible with the delegate

Methods of 'System.Nullable(Of T)' cannot be used as operands of the 'AddressOf' operator

'Module' statements can occur only at file or namespace level

Name <membername> is not CLS-compliant

Name '<name>' is not declared

Name <namespacename> in the root namespace <fullnamespacename> is not CLS-compliant

Namespace or type specified in the Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

Need property array index

Nested function does not have a signature that is compatible with delegate '<delegatename>'

No accessible 'Main' method with an appropriate signature was found in '<name>'

Non-CLS-compliant <membername> is not allowed in a CLS-compliant interface

Nullable type inference is not supported in this context

Number of indices exceeds the number of dimensions of the indexed array

Object or class does not support the set of events

Object required

Object variable or With block variable not set

Operator declaration must be one of: +,-,*,-,-,^, &, Like, Mod, And, Or, Xor, Not, <<, >>, =, <>, <, <=, >, >=, CType, IsTrue, IsFalse

'Optional' expected

Optional parameters must specify a default value

Ordinal is not valid

Out of memory (Visual Basic Compiler Error)

Out of stack space

Out of string space

Overflow (Visual Basic Error)

Overflow (Visual Basic Run-Time Error)

Path not found

Path-File access error

Permission denied

Procedure call or argument is not valid

Property '<propertyname>' doesn't return a value on all code paths

Property array index is not valid

Property let procedure not defined and property get procedure did not return an object

Property not found

Property or method not found

Range variable <variable> hides a variable in an enclosing block, a previously defined range variable, or an implicitly declared variable in a query expression

Range variable name can be inferred only from a simple or qualified name with no arguments

Reference required to assembly '<assemblyidentity>' containing type '<typename>', but a suitable reference could not be found due to ambiguity between projects '<projectname1>' and '<projectname2>'

Reference required to assembly '<assemblyname>' containing the base class '<classname>'

Requested operation is not available (BC35000)

Resume without error

Return type of function '<procedurename>' is not CLS-compliant

'Set' accessor of property '<propertyname>' is not accessible

Some subkeys cannot be deleted

Statement cannot end a block outside of a line 'If' statement

Statement is not valid in a namespace

Statement is not valid inside a method-multiline lambda

String constants must end with a double quote

Structure '<structurename>' must contain at least one instance member variable or at least one instance event declaration not marked 'Custom'

'Sub Main' was not found in '<name>'

Sub or Function not defined

Subscript out of range

TextFieldParser is unable to complete the read operation because maximum buffer

size has been exceeded

The type for variable '<variablename>' will not be inferred because it is bound to a field in an enclosing scope

This array is fixed or temporarily locked

This key is already associated with an element of this collection

Too many files

Type '<typename>' has no constructors

Type <typename> is not CLS-compliant

Type '<typename>' is not defined

Type arguments could not be inferred from the delegate

Type mismatch

Type of '<variablename>' cannot be inferred because the loop bounds and the step variable do not widen to the same type

Type of member '<membername>' is not CLS-compliant

Type of optional value for optional parameter <parametername> is not CLS-compliant

Type of parameter '<parametername>' is not CLS-compliant

Type parameters cannot be used as qualifiers

Unable to create strong-named assembly from key file '<filename>': <error>

Unable to embed resource file '<filename>': <error message>

Unable to emit assembly: <error message>

Unable to find required file '<filename>'

Unable to get serial port names because of an internal system error

Unable to link to resource file '<filename>': <error message>

Unable to load information for class '<classname>'

Unable to write output to memory

Unable to write temporary file because temporary path is not available

Unable to write to output file '<filename>': <error>

Underlying type <typename> of Enum is not CLS-compliant

Using the iteration variable in a lambda expression may have unexpected results

Value of type '<typename1>' cannot be converted to '<typename2>'

Value of type '<typename1>' cannot be converted to '<typename2>' (Multiple file references)

Value of type 'type1' cannot be converted to 'type2'
Variable '<variablename>' hides a variable in an enclosing block
Variable '<variablename>' is used before it has been assigned a value
Variable uses an Automation type not supported in Visual Basic
XML axis properties do not support late binding
XML comment exception must have a 'cref' attribute
XML entity references are not supported
XML literals and XML properties are not supported in embedded code within ASP.NET
XML namespace URI '<uri>' can be bound only to 'xmlns'
Reference
Command-Line Compiler
Building from the Command Line
How to: Invoke the Command-Line Compiler
Sample Compilation Command Lines
Compiler Options Listed Alphabetically
@ (Specify Response File)
-addmodule
-baseaddress
-bugreport
-codepage
-debug
-define
-delaysign
-deterministic
-doc
-errorreport
-filealign
-help, /?
-highentropyva
-imports
-keycontainer
-keyfile

-langversion
-libpath
-link
-linkresource
-main
-moduleassemblyname
-netcf
-noconfig
-nologo
-nostdlib
-nowarn
-nowin32manifest
-optimize
-optioncompare
-optionexplicit
-optioninfer
-optionstrict
-out
-platform
-quiet
-recurse
-reference
-refonly
-refout
-removeintchecks
-resource
-rootnamespace
-sdkpath
-target
-subsystemversion
-utf8output
-vbruntime

-verbose
-warnaserror
-win32icon
-win32manifest
-win32resource
Compiler Options Listed by Category
.NET Framework Reference Information
Language Specification
Sample Applications
Walkthroughs
ML.NET Guide
.NET for Apache Spark Guide
.NET Application Architecture Guide
Samples and Tutorials

Welcome to .NET

5/15/2019 • 2 minutes to read • Edit Online

See Get started with .NET Core to learn how to create .NET Core apps.

Build many types of apps with .NET, such as cloud, IoT, and games using free cross-platform tools. Your apps can run on Android, iOS, Linux, macOS, and Windows. Deploy apps to servers or desktops and publish to app stores for deployment on mobile devices. .NET is accessible to students and hobbyists, and all are welcome to participate in a lively international developer community and make direct contributions to many of the .NET technologies.

News

For news about .NET, check the following blogs:

- .NET Blog
- ASP.NET Blog
- The Visual Studio Blog
- The Visual Basic Team

Also follow the latest .NET events:

- Microsoft Connect()
- Microsoft Build 2018
- .NET Conf

For information about the latest features added to the .NET implementations and supported languages, see the following articles:

- What's new in .NET Core
- What's new in the .NET Framework
- What's new in C#
- What's new for Visual Basic

Documentation

This documentation covers the breadth of .NET across platforms and languages. You can get started with .NET and its languages in any of the following sections:

- .NET Guide
- .NET Core Guide
- .NET Framework Guide
- C# Guide
- F# Guide
- Visual Basic Guide
- ML.NET Guide
- .NET for Apache Spark

Additionally, you can browse the .NET API reference.

Open source

This documentation is completely open source. You can contribute in any way you like, from creating issues to writing documentation. Additionally, much of .NET itself is also open source. The following list shows the main .NET GitHub repositories:

- .NET Core Home
- .NET Libraries
- .NET Core Runtime
- .NET Core CLI
- Roslyn (C# and Visual Basic) Compiler Platform and IDE Tools
- F# Compiler and IDE Tools
- Windows Forms
- WPF

You can join other people who are already active in the .NET community to find out what's new or ask for help.

.NET Guide

1/19/2019 • 2 minutes to read • Edit Online

The .NET Guide provides a large amount of information about .NET. Depending on your familiarity with .NET, you may wish to explore different sections of this guide and other sections of the .NET documentation.

New to .NET

If you want a high-level overview about .NET, check out What is .NET?.

If you're new to .NET, check out the Get Started article.

If you prefer to have a guided tour through major features of .NET, check out the Tour of .NET.

You can also read about .NET Architectural Components to get an overview of the various "pieces" of .NET and how they fit together.

New to .NET Core

If you're new to .NET Core, check out Get Started with .NET Core.

New to .NET Standard

If you're new to .NET Standard, check out .NET Standard.

Porting .NET Framework Code to .NET Core

If you're looking to port an application, service, or some component of a system to .NET Core, check out Porting to .NET Core from .NET Framework.

Porting a NuGet package from .NET Framework to .NET Standard or .NET Core

If you're looking to port a NuGet package to .NET Standard, check out Porting to .NET Core from .NET Framework. Tooling for .NET Standard and .NET Core are shared, so the content will be relevant for porting to .NET Standard as well as .NET Core.

Interested in Major .NET Concepts

If you're interested in some of the major concepts of .NET, check out:

- .NET Architectural Components
- .NET Standard
- Native Interoperability
- Garbage Collection
- Base Types in .NET
- Collections
- Dates, times, and time zones
- Asynchronous Programming

Additionally, check out each language guide to learn about the three major .NET languages:

- C# Guide
- F# Guide
- Visual Basic Guide

API Reference

Check out the .NET API Reference to see the breadth of APIs available.

Get Started

5/15/2019 • 2 minutes to read • Edit Online

There are a number of ways to get started with .NET. Because .NET is a massive platform, there are multiple articles in this documentation which show how you can get started with .NET, each from a different perspective.

Get started using .NET languages

- The C# Getting Started articles and C# Tutorials provide a number of ways to get started in a C#-centric way.
- The F# Getting Started tutorials provide three primary ways you can use F#: with Visual Studio, Visual Studio Code, or command-line tools.
- The Visual Basic Getting Started articles provide guides for using Visual Basic in Visual Studio.

Get started using .NET Core

- Getting Started with .NET Core provides an overview of articles which show how to get started with .NET Core on different operating systems and using different tools.
- The .NET Core Tutorials detail a number of ways you can get started with .NET Core using your operating system and tooling of choice.

Get started using .NET Core on Docker

Introduction to .NET and Docker shows how you can use .NET Core on Windows Docker containers.

Tour of .NET

1/19/2019 • 9 minutes to read • Edit Online

.NET is a general purpose development platform. It has several key features, such as support for multiple programming languages, asynchronous and concurrent programming models, and native interoperability, which enable a wide range of scenarios across multiple platforms.

This article offers a guided tour through some of the key features of the .NET. See the .NET Architectural Components topic to learn about the architectural pieces of .NET and what they're used for.

How to run the code samples

To learn how to set up a development environment to run the code samples, see the Getting Started topic. Copy and paste code samples from this page into your environment to execute them.

Programming languages

.NET supports multiple programming languages. The .NET implementations implement the Common Language Infrastructure (CLI), which among other things specifies a language-independent runtime and language interoperability. This means that you choose any .NET language to build apps and services on .NET.

Microsoft actively develops and supports three .NET languages: C#, F#, and Visual Basic (VB).

- C# is simple, powerful, type-safe, and object-oriented, while retaining the expressiveness and elegance of C-style languages. Anyone familiar with C and similar languages finds few problems in adapting to C#. Check out the C# Guide to learn more about C#.
- F# is a cross-platform, functional-first programming language that also supports traditional object-oriented and imperative programming. Check out the F# Guide to learn more about F#.
- Visual Basic is an easy language to learn that you use to build a variety of apps that run on .NET. Among the .NET languages, the syntax of VB is the closest to ordinary human language, often making it easier for people new to software development.

Automatic memory management

.NET uses garbage collection (GC) to provide automatic memory management for programs. The GC operates on a lazy approach to memory management, preferring app throughput to the immediate collection of memory. To learn more about the .NET GC, check out Fundamentals of garbage collection (GC).

The following two lines both allocate memory:

```
var title = ".NET Primer";
var list = new List<string>();
```

There's no analogous keyword to de-allocate memory, as de-allocation happens automatically when the garbage collector reclaims the memory through its scheduled run.

The garbage collector is one of the services that help ensure *memory safety*. A program is memory safe if it accesses only allocated memory. For instance, the runtime ensures that an app doesn't access unallocated memory beyond the bounds of an array.

In the following example, the runtime throws an `InvalidIndexException` exception to enforce memory safety:

```
int[] numbers = new int[42];
int number = numbers[42]; // Will throw an exception (indexes are 0-based)
```

Working with unmanaged resources

Some objects reference *unmanaged resources*. Unmanaged resources are resources that aren't automatically maintained by the .NET runtime. For example, a file handle is an unmanaged resource. A FileStream object is a managed object, but it references a file handle, which is unmanaged. When you're done using the FileStream, you need to release the file handle.

In .NET, objects that reference unmanaged resources implement the IDisposable interface. When you're done using the object, you call the object's Dispose() method, which is responsible for releasing any unmanaged resources. .NET languages provide a convenient `using` syntax for such objects, as shown in the following example:

```
using System.IO;

using (FileStream stream = GetFileStream(context))
{
    // Operations on the stream
}
```

Once the `using` block completes, the .NET runtime automatically calls the `stream` object's Dispose() method, which releases the file handle. The runtime also does this if an exception causes control to leave the block.

For more details, see the following topics:

- For C#, see the using Statement (C# Reference) topic.
- For F#, see Resource Management: The use Keyword.
- For VB, see the Using Statement (Visual Basic) topic.

Type safety

An object is an instance of a specific type. The only operations allowed for a given object are those of its type. A `Dog` type may have `Jump` and `WagTail` methods but not a `SumTotal` method. A program only calls the methods belonging to a given type. All other calls result in either a compile-time error or a run-time exception (in case of using dynamic features or `object`).

.NET languages are object-oriented with hierarchies of base and derived classes. The .NET runtime only allows object casts and calls that align with the object hierarchy. Remember that every type defined in any .NET language derives from the base Object type.

```
Dog dog = AnimalShelter.AdoptDog(); // Returns a Dog type.
Pet pet = (Pet)dog; // Dog derives from Pet.
pet.ActCute();
Car car = (Car)dog; // Will throw - no relationship between Car and Dog.
object temp = (object)dog; // Legal - a Dog is an object.
```

Type safety is also used to help enforce encapsulation by guaranteeing the fidelity of the accessor keywords. Accessor keywords are artifacts which control access to members of a given type by other code. These are usually used for various kinds of data within a type that are used to manage its behavior.

```
private Dog _nextDogToBeAdopted = AnimalShelter.AdoptDog()
```

C#, VB, and F# support local *type inference*. Type inference means that the compiler deduces the type of the expression on the left-hand side from the expression on the right-hand side. This doesn't mean that the type safety is broken or avoided. The resulting type does have a strong type with everything that implies. From the previous example, `dog` is rewritten to introduce type inference, and the remainder of the example is unchanged:

```
var dog = AnimalShelter.AdoptDog();
var pet = (Pet)dog;
pet.ActCute();
Car car = (Car)dog; // will throw - no relationship between Car and Dog
object temp = (object)dog; // legal - a Dog is an object
car = (Car)temp; // will throw - the runtime isn't fooled
car.Accelerate() // the dog won't like this, nor will the program get this far
```

F# has even further type inference capabilities than the method-local type inference found in C# and VB. To learn more, see Type Inference.

Delegates and lambdas

A delegate is represented by a method signature. Any method with that signature can be assigned to the delegate and is executed when the delegate is invoked.

Delegates are like C++ function pointers except that they're type safe. They're a kind of disconnected method within the CLR type system. Regular methods are attached to a class and are only directly callable through static or instance calling conventions.

In .NET, delegates are commonly used in event handlers, in defining asynchronous operations, and in lambda expressions, which are a cornerstone of LINQ. Learn more in the Delegates and lambdas topic.

Generics

Generics allow the programmer to introduce a *type parameter* when designing their classes that allows the client code (the users of the type) to specify the exact type to use in place of the type parameter.

Generics were added to help programmers implement generic data structures. Before their arrival in order for a type such as the `List` type to be generic, it would have to work with elements that were of type `object`. This had various performance and semantic problems, along with possible subtle runtime errors. The most notorious of the latter is when a data structure contains, for instance, both integers and strings, and an `InvalidCastException` is thrown on working with the list's members.

The following sample shows a basic program running using an instance of List<T> types:

```
using System;
using System.Collections.Generic;

namespace GenericsSampleShort
{
    public static void Main(string[] args)
    {
        // List<string> is the client way of specifying the actual type for the type parameter T
        List<string> listOfStrings = new List<string> { "First", "Second", "Third" };

        // listOfStrings can accept only strings, both on read and write.
        listOfStrings.Add("Fourth");

        // Below will throw a compile-time error, since the type parameter
        // specifies this list as containing only strings.
        listOfStrings.Add(1);
    }
}
```

For more information, see the Generic types (Generics) overview topic.

Async programming

Async programming is a first-class concept within .NET with async support in the runtime, framework libraries, and .NET language constructs. Internally, they're based on objects (such as `Task`), which take advantage of the operating system to perform I/O-bound jobs as efficiently as possible.

To learn more about async programming in .NET, start with the Async overview topic.

Language Integrated Query (LINQ)

LINQ is a powerful set of features for C# and VB that allow you to write simple, declarative code for operating on data. The data can be in many forms (such as in-memory objects, a SQL database, or an XML document), but the LINQ code you write typically doesn't differ by data source.

To learn more and see some samples, see the LINQ (Language Integrated Query) topic.

Native interoperability

Every operating system includes an application programming interface (API) that provides system services. .NET provides several ways to call those APIs.

The main way to do native interoperability is via "platform invoke" or P/Invoke for short, which is supported across Linux and Windows platforms. A Windows-only way of doing native interoperability is known as "COM interop," which is used to work with COM components in managed code. It's built on top of the P/Invoke infrastructure, but it works in subtly different ways.

Most of Mono's (and thus Xamarin's) interoperability support for Java and Objective-C are built similarly, that is, they use the same principles.

For more information about native interoperability, see the Native interoperability article.

Unsafe code

Depending on language support, the CLR lets you access native memory and do pointer arithmetic via `unsafe` code. These operations are needed for certain algorithms and system interoperability. Although powerful, use of unsafe code is discouraged unless it's necessary to interop with system APIs or implement the most efficient algorithm. Unsafe code may not execute the same way in different environments and also loses the benefits of a

garbage collector and type safety. It's recommended to confine and centralize unsafe code as much as possible and test that code thoroughly.

The following example is a modified version of the `ToString()` method from the `StringBuilder` class. It illustrates how using `unsafe` code can efficiently implement an algorithm by moving around chunks of memory directly:

```
public override String ToString()
{
    if (Length == 0)
        return String.Empty;

    string ret = string.FastAllocateString(Length);
    StringBuilder chunk = this;
    unsafe
    {
        fixed (char* destinationPtr = ret)
        {
            do
            {
                if (chunk.m_ChunkLength > 0)
                {
                    // Copy these into local variables so that they are stable even in the presence of ----s
(hackers might do this)
                    char[] sourceArray = chunk.m_ChunkChars;
                    int chunkOffset = chunk.m_ChunkOffset;
                    int chunkLength = chunk.m_ChunkLength;

                    // Check that we will not overrun our boundaries.
                    if ((uint)(chunkLength + chunkOffset) <= ret.Length && (uint)chunkLength <=
(uint)sourceArray.Length)
                    {
                        fixed (char* sourcePtr = sourceArray)
                            string.wstrcpy(destinationPtr + chunkOffset, sourcePtr, chunkLength);
                    }
                    else
                    {
                        throw new ArgumentOutOfRangeException("chunkLength",
Environment.GetResourceString("ArgumentOutOfRange_Index"));
                    }
                }
                chunk = chunk.m_ChunkPrevious;
            } while (chunk != null);
        }
    }
    return ret;
}
```

Next steps

If you're interested in a tour of C# features, check out Tour of C#.

If you're interested in a tour of F# features, see Tour of F#.

If you want to get started with writing code of your own, visit Getting Started.

To learn about important components of .NET, check out .NET Architectural Components.

.NET architectural components

8/24/2019 • 4 minutes to read • Edit Online

A .NET app is developed for and runs in one or more *implementations of .NET*. Implementations of .NET include the .NET Framework, .NET Core, and Mono. There is an API specification common to all implementations of .NET that's called the .NET Standard. This article gives a brief introduction to each of these concepts.

.NET Standard

The .NET Standard is a set of APIs that are implemented by the Base Class Library of a .NET implementation. More formally, it's a specification of .NET APIs that make up a uniform set of contracts that you compile your code against. These contracts are implemented in each .NET implementation. This enables portability across different .NET implementations, effectively allowing your code to run everywhere.

The .NET Standard is also a target framework. If your code targets a version of the .NET Standard, it can run on any .NET implementation which supports that version of the .NET Standard.

To learn more about the .NET Standard and how to target it, see the .NET Standard topic.

.NET implementations

Each implementation of .NET includes the following components:

- One or more runtimes. Examples: CLR for .NET Framework, CoreCLR and CoreRT for .NET Core.
- A class library that implements the .NET Standard and may implement additional APIs. Examples: .NET Framework Base Class Library, .NET Core Base Class Library.
- Optionally, one or more application frameworks. Examples: ASP.NET, Windows Forms, and Windows Presentation Foundation (WPF) are included in the .NET Framework and .NET Core.
- Optionally, development tools. Some development tools are shared among multiple implementations.

There are four primary .NET implementations that Microsoft actively develops and maintains: .NET Core, .NET Framework, Mono, and UWP.

.NET Core

.NET Core is a cross-platform implementation of .NET and designed to handle server and cloud workloads at scale. It runs on Windows, macOS and Linux. It implements the .NET Standard, so code that targets the .NET Standard can run on .NET Core. ASP.NET Core, Windows Forms, and Windows Presentation Foundation (WPF) all run on .NET Core.

To learn more about .NET Core, see the .NET Core Guide and Choosing between .NET Core and .NET Framework for server apps.

.NET Framework

The.NET Framework is the original .NET implementation that has existed since 2002. It's the same .NET Framework that existing .NET developers have always used. Versions 4.5 and later implement the .NET Standard, so code that targets the .NET Standard can run on those versions of the .NET Framework. It contains additional Windows-specific APIs, such as APIs for Windows desktop development with Windows Forms and WPF. The .NET Framework is optimized for building Windows desktop applications.

To learn more about the .NET Framework, see the .NET Framework Guide.

Mono

Mono is a .NET implementation that is mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, Mac, iOS, tvOS and watchOS and is focused primarily on a small footprint. Mono also powers games built using the Unity engine.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of the .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a just-in-time compiler, but it also features a full static compiler (ahead-of-time compilation) that is used on platforms like iOS.

To learn more about Mono, see the Mono documentation.

Universal Windows Platform (UWP)

UWP is an implementation of .NET that is used for building modern, touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phablets, phones, and even the Xbox. UWP provides many services, such as a centralized app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, VB.NET, and JavaScript. When using C# and VB.NET, the .NET APIs are provided by .NET Core.

To learn more about UWP, see Intro to the Universal Windows Platform.

.NET runtimes

A runtime is the execution environment for a managed program. The OS is part of the runtime environment but is not part of the .NET runtime. Here are some examples of .NET runtimes:

- Common Language Runtime (CLR) for the .NET Framework
- Core Common Language Runtime (CoreCLR) for .NET Core
- .NET Native for Universal Windows Platform
- The Mono runtime for Xamarin.iOS, Xamarin.Android, Xamarin.Mac, and the Mono desktop framework

.NET tooling and common infrastructure

You have access to an extensive set of tools and infrastructure components that work with every implementation of .NET. These include, but are not limited to the following:

- The .NET languages and their compilers
- The .NET project system (based on *.csproj*, *.vbproj*, and *.fsproj* files)
- MSBuild, the build engine used to build projects
- NuGet, Microsoft's package manager for .NET
- Open-source build orchestration tools, such as CAKE and FAKE

See also

- Choosing between .NET Core and .NET Framework for server apps
- .NET Standard
- .NET Core Guide
- .NET Framework Guide
- C# Guide
- F# Guide
- VB.NET Guide

.NET Standard

4/25/2019 • 9 minutes to read • Edit Online

The .NET Standard is a formal specification of .NET APIs that are intended to be available on all .NET implementations. The motivation behind the .NET Standard is establishing greater uniformity in the .NET ecosystem. ECMA 335 continues to establish uniformity for .NET implementation behavior, but there's no similar spec for the .NET Base Class Libraries (BCL) for .NET library implementations.

The .NET Standard enables the following key scenarios:

- Defines uniform set of BCL APIs for all .NET implementations to implement, independent of workload.
- Enables developers to produce portable libraries that are usable across .NET implementations, using this same set of APIs.
- Reduces or even eliminates conditional compilation of shared source due to .NET APIs, only for OS APIs.

The various .NET implementations target specific versions of .NET Standard. Each .NET implementation version advertises the highest .NET Standard version it supports, a statement that means it also supports previous versions. For example, the .NET Framework 4.6 implements .NET Standard 1.3, which means that it exposes all APIs defined in .NET Standard versions 1.0 through 1.3. Similarly, the .NET Framework 4.6.1 implements .NET Standard 1.4, while .NET Core 1.0 implements .NET Standard 1.6.

.NET implementation support

The following table lists the **minimum** platform versions that support each .NET Standard version. That means that later versions of a listed platform also support the corresponding .NET Standard version. For example, .NET Core 2.2 supports .NET Standard 2.0 and earlier.

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1

1 The versions listed for .NET Framework apply to .NET Core 2.0 SDK and later versions of the tooling. Older versions used a different mapping for .NET Standard 1.5 and higher. You can download tooling for .NET Core tools for Visual Studio 2015 if you cannot upgrade to Visual Studio 2017.

2 The versions listed here represent the rules that NuGet uses to determine whether a given .NET Standard library is applicable. While NuGet considers .NET Framework 4.6.1 as supporting .NET Standard 1.5 through 2.0, there are several issues with consuming .NET Standard libraries that were built for those versions from .NET Framework 4.6.1 projects. For .NET Framework projects that need to use such libraries, we recommend that you upgrade the project to target .NET Framework 4.7.2 or higher.

- The columns represent .NET Standard versions. Each header cell is a link to a document that shows which APIs got added in that version of .NET Standard.
- The rows represent the different .NET implementations.
- The version number in each cell indicates the *minimum* version of the implementation you'll need in order to target that .NET Standard version.
- For an interactive table, see .NET Standard versions.

To find the highest version of .NET Standard that you can target, do the following steps:

1. Find the row that indicates the .NET implementation you want to run on.
2. Find the column in that row that indicates your version starting from right to left.
3. The column header indicates the .NET Standard version that your target supports. You may also target any lower .NET Standard version. Higher .NET Standard versions will also support your implementation.
4. Repeat this process for each platform you want to target. If you have more than one target platform, you should pick the smaller version among them. For example, if you want to run on .NET Framework 4.5 and .NET Core 1.0, the highest .NET Standard version you can use is .NET Standard 1.1.

Which .NET Standard version to target

When choosing a .NET Standard version, you should consider this trade-off:

- The higher the version, the more APIs are available to you.
- The lower the version, the more platforms implement it.

In general, we recommend you to target the *lowest* version of .NET Standard possible. So, after you find the highest .NET Standard version you can target, follow these steps:

1. Target the next lower version of .NET Standard and build your project.
2. If your project builds successfully, repeat step 1. Otherwise, retarget to the next higher version and that's the version you should use.

However, targeting lower .NET Standard versions introduces a number of support dependencies. If your project targets .NET Standard 1.*x*, we recommend that you *also* target .NET Standard 2.0. This simplifies the dependency graph for users of your library that run on .NET Standard 2.0 compatible frameworks, and it reduces the number of packages they need to download.

.NET Standard versioning rules

There are two primary versioning rules:

- Additive: .NET Standard versions are logically concentric circles: higher versions incorporate all APIs from previous versions. There are no breaking changes between versions.
- Immutable: Once shipped, .NET Standard versions are frozen. New APIs first become available in specific .NET implementations, such as .NET Core. If the .NET Standard review board believes the new APIs should

be available for all .NET implementations, they're added in a new .NET Standard version.

Specification

The .NET Standard specification is a standardized set of APIs. The specification is maintained by .NET implementors, specifically Microsoft (includes .NET Framework, .NET Core, and Mono) and Unity. A public feedback process is used as part of establishing new .NET Standard versions through GitHub.

Official artifacts

The official specification is a set of .cs files that define the APIs that are part of the standard. The ref directory in the dotnet/standard repository defines the .NET Standard APIs.

The NETStandard.Library metapackage (source) describes the set of libraries that define (in part) one or more .NET Standard versions.

A given component, like `System.Runtime`, describes:

- Part of .NET Standard (just its scope).
- Multiple versions of .NET Standard, for that scope.

Derivative artifacts are provided to enable more convenient reading and to enable certain developer scenarios (for example, using a compiler).

- API list in markdown
- Reference assemblies, distributed as NuGet packages and referenced by the NETStandard.Library metapackage.

Package representation

The primary distribution vehicle for the .NET Standard reference assemblies is NuGet packages. Implementations are delivered in a variety of ways, appropriate for each .NET implementation.

NuGet packages target one or more frameworks. The .NET Standard packages target the ".NET Standard" framework. You can target the .NET Standard framework using the `netstandard` compact TFM (for example, `netstandard1.4`). Libraries that are intended to run on multiple runtimes should target this framework. For the broadest set of APIs, target `netstandard2.0` since the number of available APIs more than doubled between .NET Standard 1.6 and 2.0.

The `NETStandard.Library` metapackage references the complete set of NuGet packages that define .NET Standard. The most common way to target `netstandard` is by referencing this metapackage. It describes and provides access to the ~40 .NET libraries and associated APIs that define .NET Standard. You can reference additional packages that target `netstandard` to get access to additional APIs.

Versioning

The specification is not singular, but an incrementally growing and linearly versioned set of APIs. The first version of the standard establishes a baseline set of APIs. Subsequent versions add APIs and inherit APIs defined by previous versions. There is no established provision for removing APIs from the standard.

.NET Standard is not specific to any one .NET implementation, nor does it match the versioning scheme of any of those runtimes.

APIs added to any of the implementations (such as, .NET Framework, .NET Core and Mono) can be considered as candidates to add to the specification, particularly if they are thought to be fundamental in nature. New versions of .NET Standard are created based on .NET implementation releases, enabling you to target new APIs from a .NET Standard PCL. The versioning mechanics are described in more detail in .NET Core Versioning.

.NET Standard versioning is important for usage. Given a .NET Standard version, you can use libraries that

target that same or lower version. The following approach describes the workflow for using .NET Standard PCLs, specific to .NET Standard targeting.

- Select a .NET Standard version to use for your PCL.
- Use libraries that depend on the same .NET Standard version or lower.
- If you find a library that depends on a higher .NET Standard version, you either need to adopt that same version or decide not to use that library.

Targeting .NET Standard

You can build .NET Standard Libraries using a combination of the `netstandard` framework and the NETStandard.Library metapackage. You can see examples of targeting the .NET Standard with .NET Core tools.

.NET Framework compatibility mode

Starting with .NET Standard 2.0, the .NET Framework compatibility mode was introduced. This compatibility mode allows .NET Standard projects to reference .NET Framework libraries as if they were compiled for .NET Standard. Referencing .NET Framework libraries doesn't work for all projects, such as libraries that use Windows Presentation Foundation (WPF) APIs.

For more information, see .NET Framework compatibility mode.

.NET Standard libraries and Visual Studio

In order to build .NET Standard libraries in Visual Studio, make sure you have Visual Studio 2017 version 15.3 or later installed on Windows, or Visual Studio for Mac version 7.1 or later installed on macOS.

If you only need to consume .NET Standard 2.0 libraries in your projects, you can also do that in Visual Studio 2015. However, you need NuGet client 3.6 or higher installed. You can download the NuGet client for Visual Studio 2015 from the NuGet downloads page.

Comparison to Portable Class Libraries

.NET Standard is the replacement for Portable Class Libraries (PCL). The .NET Standard improves on the experience of creating portable libraries by curating a standard BCL and establishing greater uniformity across .NET implementations as a result. A library that targets .NET Standard is a PCL or a ".NET Standard-based PCL". Existing PCLs are "profile-based PCLs".

.NET Standard and PCL profiles were created for similar purposes but also differ in key ways.

Similarities:

- Define APIs that can be used for binary code sharing.

Differences:

- .NET Standard is a curated set of APIs, while PCL profiles are defined by intersections of existing platforms.
- .NET Standard linearly versions, while PCL profiles do not.
- PCL profiles represents Microsoft platforms while the .NET Standard is platform-agnostic.

PCL compatibility

.NET Standard is compatible with a subset of PCL profiles. .NET Standard 1.0, 1.1 and 1.2 each overlap with a set of PCL profiles. This overlap was created for two reasons:

- Enable .NET Standard-based PCLs to reference profile-based PCLs.
- Enable profile-based PCLs to be packaged as .NET Standard-based PCLs.

Profile-based PCL compatibility is provided by the Microsoft.NETCore.Portable.Compatibility NuGet package. This dependency is required when referencing NuGet packages that contain profile-based PCLs.

Profile-based PCLs packaged as `netstandard` are easier to consume than typically packaged profile-based PCLs. `netstandard` packaging is compatible with existing users.

You can see the set of PCL profiles that are compatible with the .NET Standard:

PCL PROFILE	.NET STANDARD	PCL PLATFORMS
Profile7	1.1	.NET Framework 4.5, Windows 8
Profile31	1.0	Windows 8.1, Windows Phone Silverlight 8.1
Profile32	1.2	Windows 8.1, Windows Phone 8.1
Profile44	1.2	.NET Framework 4.5.1, Windows 8.1
Profile49	1.0	.NET Framework 4.5, Windows Phone Silverlight 8
Profile78	1.0	.NET Framework 4.5, Windows 8, Windows Phone Silverlight 8
Profile84	1.0	Windows Phone 8.1, Windows Phone Silverlight 8.1
Profile111	1.1	.NET Framework 4.5, Windows 8, Windows Phone 8.1
Profile151	1.2	.NET Framework 4.5.1, Windows 8.1, Windows Phone 8.1
Profile157	1.0	Windows 8.1, Windows Phone 8.1, Windows Phone Silverlight 8.1
Profile259	1.0	.NET Framework 4.5, Windows 8, Windows Phone 8.1, Windows Phone Silverlight 8

See also

- .NET Standard Versions

What's new in the .NET Standard

3/9/2019 • 3 minutes to read • Edit Online

The .NET Standard is a formal specification that defines a versioned set of APIs that must be available on .NET implementations that comply with that version of the standard. The .NET Standard is targeted at library developers. A library that targets a .NET Standard version can be used on any .NET Framework, .NET Core, or Xamarin implementation that supports that version of the standard.

The most recent version of the .NET Standard is 2.0. It is included with the .NET Core 2.0 SDK, as well as with Visual Studio 2017 Version 15.3 with the .NET Core workload installed.

Supported .NET implementations

The .NET Standard 2.0 is supported by the following .NET implementations:

- .NET Core 2.0 or later
- .NET Framework 4.6.1 or later
- Mono 5.4 or later
- Xamarin.iOS 10.14 or later
- Xamarin.Mac 3.8 or later
- Xamarin.Android 8.0 or later
- Universal Windows Platform 10.0.16299 or later

What's new in the .NET Standard 2.0

The .NET Standard 2.0 includes the following new features:

A vastly expanded set of APIs

Through version 1.6, the .NET Standard included a comparatively small subset of APIs. Among those excluded were many APIs that were commonly used in the .NET Framework or Xamarin. This complicates development, since it requires that developers find suitable replacements for familiar APIs when they develop applications and libraries that target multiple .NET implementations. The .NET Standard 2.0 addresses this limitation by adding over 20,000 more APIs than were available in .NET Standard 1.6, the previous version of the standard. For a list of the APIs that have been added to the .NET Standard 2.0, see .NET Standard 2.0 vs 1.6.

Some of the additions to the System namespace in .NET Standard 2.0 include:

- Support for the AppDomain class.
- Better support for working with arrays from additional members in the Array class.
- Better support for working with attributes from additional members in the Attribute class.
- Better calendar support and additional formatting options for DateTime values.
- Additional Decimal rounding functionality.
- Additional functionality in the Environment class.
- Enhanced control over the garbage collector through the GC class.
- Enhanced support for string comparison, enumeration, and normalization in the String class.
- Support for daylight saving adjustments and transition times in the TimeZoneInfo.AdjustmentRule and TimeZoneInfo.TransitionTime classes.
- Significantly enhanced functionality in the Type class.
- Better support for deserialization of exception objects by adding an exception constructor with SerializationInfo

and StreamingContext parameters.

Support for .NET Framework libraries

The overwhelming majority of libraries target the .NET Framework rather than .NET Standard. However, most of the calls in those libraries are to APIs that are included in the .NET Standard 2.0. Starting with the .NET Standard 2.0, you can access .NET Framework libraries from a .NET Standard library by using a compatibility shim. This compatibility layer is transparent to developers; you don't have to do anything to take advantage of .NET Framework libraries.

The single requirement is that the APIs called by the .NET Framework class library must be included in the .NET Standard 2.0.

Support for Visual Basic

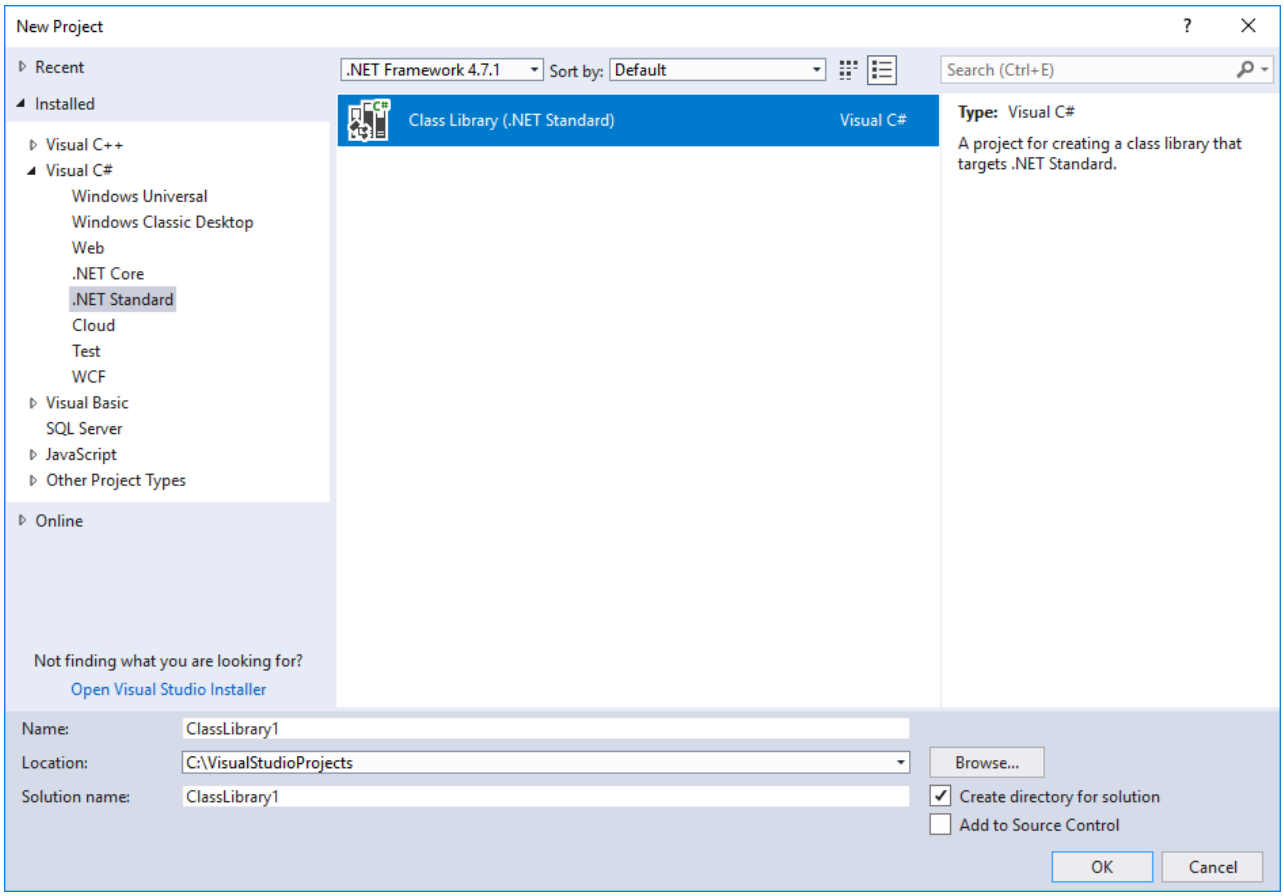
You can now develop .NET Standard libraries in Visual Basic. For Visual Basic developers using Visual Studio 2017 Version 15.3 or later with the .NET Core workload installed, Visual Studio now includes a .NET Standard Class Library template. For Visual Basic developers who use other development tools and environments, you can use the dotnet new command to create a .NET Standard Library project. For more information, see the Tooling support for .NET Standard libraries.

Tooling support for .NET Standard libraries

With the release of .NET Core 2.0 and .NET Standard 2.0, both Visual Studio 2017 and the .NET Core Command Line Interface (CLI) include tooling support for creating .NET Standard libraries.

If you install Visual Studio with the **.NET Core cross-platform development** workload, you can create a .NET Standard 2.0 library project by using a project template, as the following figure shows:

- C#
- Visual Basic



If you're using the .NET Core CLI, the following dotnet new command creates a class library project that targets the .NET Standard 2.0:

```
dotnet new classlib
```

See also

- .NET Standard
- Introducing .NET Standard

Target frameworks

8/20/2019 • 4 minutes to read • Edit Online

When you target a framework in an app or library, you're specifying the set of APIs that you'd like to make available to the app or library. You specify the target framework in your project file using Target Framework Monikers (TFMs).

An app or library can target a version of .NET Standard. .NET Standard versions represent standardized sets of APIs across all .NET implementations. For example, a library can target .NET Standard 1.6 and gain access to APIs that function across .NET Core and .NET Framework using the same codebase.

An app or library can also target a specific .NET implementation to gain access to implementation-specific APIs. For example, an app that targets Xamarin.iOS (for example, `Xamarin.iOS10`) gets access to Xamarin-provided iOS API wrappers for iOS 10, or an app that targets the Universal Windows Platform (UWP, `uap10.0`) has access to APIs that compile for devices that run Windows 10.

For some target frameworks (for example, the .NET Framework), the APIs are defined by the assemblies that the framework installs on a system and may include application framework APIs (for example, ASP.NET).

For package-based target frameworks (for example, .NET Standard and .NET Core), the APIs are defined by the packages included in the app or library. A *metapackage* is a NuGet package that has no content of its own but is a list of dependencies (other packages). A NuGet package-based target framework implicitly specifies a metapackage that references all the packages that together make up the framework.

Latest target framework versions

The following table defines the most common target frameworks, how they're referenced, and which version of the .NET Standard they implement. These target framework versions are the latest stable versions. Pre-release versions aren't shown. A Target Framework Moniker (TFM) is a standardized token format for specifying the target framework of a .NET app or library.

TARGET FRAMEWORK	LATEST STABLE VERSION	TARGET FRAMEWORK MONIKER (TFM)	IMPLEMENTED .NET STANDARD VERSION
.NET Standard	2.0	netstandard2.0	N/A
.NET Core	2.2	netcoreapp2.2	2.0
.NET Framework	4.8	net48	2.0

Supported target framework versions

A target framework is typically referenced by a TFM. The following table shows the target frameworks supported by the .NET Core SDK and the NuGet client. Equivalents are shown within brackets. For example, `win81` is an equivalent TFM to `netcore451`.

TARGET FRAMEWORK	TFM

TARGET FRAMEWORK	TFM
.NET Standard	netstandard1.0 netstandard1.1 netstandard1.2 netstandard1.3 netstandard1.4 netstandard1.5 netstandard1.6 netstandard2.0
.NET Core	netcoreapp1.0 netcoreapp1.1 netcoreapp2.0 netcoreapp2.1 netcoreapp2.2
.NET Framework	net11 net20 net35 net40 net403 net45 net451 net452 net46 net461 net462 net47 net471 net472 net48
Windows Store	netcore [netcore45] netcore45 [win] [win8] netcore451 [win81]
.NET Micro Framework	netmf
Silverlight	sl4 sl5
Windows Phone	wp [wp7] wp7 wp75 wp8 wp81 wpa81
Universal Windows Platform	uap [uap10.0] uap10.0 [win10] [netcore50]

How to specify target frameworks

Target frameworks are specified in your project file. When a single target framework is specified, use the **TargetFramework** element. The following console app project file demonstrates how to target .NET Core 2.2:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

</Project>
```

When you specify multiple target frameworks, you may conditionally reference assemblies for each target framework. In your code, you can conditionally compile against those assemblies by using preprocessor symbols with *if-then-else* logic.

The following library project file targets APIs of .NET Standard (`netstandard1.4`) and APIs of the .NET Framework (`net40` and `net45`). Use the plural **TargetFrameworks** element with multiple target frameworks. Note how the `Condition` attributes include implementation-specific packages when the library is compiled for the two .NET Framework TFMs:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.0 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net40' ">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.5 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net45' ">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>

</Project>
```

Within your library or app, you write conditional code to compile for each target framework:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        Console.WriteLine("Target framework: .NET Framework 4.0");
#elif NET45
        Console.WriteLine("Target framework: .NET Framework 4.5");
#else
        Console.WriteLine("Target framework: .NET Standard 1.4");
#endif
    }
}
```

The build system is aware of preprocessor symbols representing the target frameworks shown in the Supported target framework versions table. When using a symbol that represents a .NET Standard or .NET Core TFM, replace the dot with an underscore and change lowercase letters to uppercase (for example, the symbol for `netstandard1.4` is `NETSTANDARD1_4`).

The complete list of preprocessor symbols for .NET Core target frameworks is:

TARGET FRAMEWORKS	SYMBOLS
.NET Framework	`NETFRAMEWORK`, `NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`, `NET48`
.NET Standard	`NETSTANDARD`, `NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`

Deprecated target frameworks

The following target frameworks are deprecated. Packages targeting these target frameworks should migrate to the indicated replacements.

DEPRECATED TFM	REPLACEMENT
aspnet50 aspnetcore50 dnxcore50 dnx dnx45 dnx451 dnx452	netcoreapp
dotnet dotnet50 dotnet51 dotnet52 dotnet53 dotnet54 dotnet55 dotnet56	netstandard
netcore50	uap10.0
win	netcore45
win8	netcore45
win81	netcore451
win10	uap10.0
winrt	netcore45

See also

- Packages, Metapackages and Frameworks
- Developing Libraries with Cross Platform Tools
- .NET Standard

- .NET Core Versioning
- dotnet/standard GitHub repository
- NuGet Tools GitHub Repository
- Framework Profiles in .NET

.NET Glossary

3/8/2019 • 11 minutes to read • Edit Online

The primary goal of this glossary is to clarify meanings of selected terms and acronyms that appear frequently in the .NET documentation without definitions.

AOT

Ahead-of-time compiler.

Similar to JIT, this compiler also translates IL to machine code. In contrast to JIT compilation, AOT compilation happens before the application is executed and is usually performed on a different machine. Because AOT tool chains don't compile at runtime, they don't have to minimize time spent compiling. That means they can spend more time optimizing. Since the context of AOT is the entire application, the AOT compiler also performs cross-module linking and whole-program analysis, which means that all references are followed and a single executable is produced.

See CoreRT and .NET Native.

ASP.NET

The original ASP.NET implementation that ships with the .NET Framework.

Sometimes ASP.NET is an umbrella term that refers to both ASP.NET implementations including ASP.NET Core. The meaning that the term carries in any given instance is determined by context. Refer to ASP.NET 4.x when you want to make it clear that you're not using ASP.NET to mean both implementations.

See ASP.NET documentation.

ASP.NET Core

A cross-platform, high-performance, open source implementation of ASP.NET built on .NET Core.

See ASP.NET Core documentation.

assembly

A *.dll*/*.exe* file that can contain a collection of APIs that can be called by applications or other assemblies.

An assembly may include types such as interfaces, classes, structures, enumerations, and delegates. Assemblies in a project's *bin* folder are sometimes referred to as *binaries*. See also library.

CLR

Common Language Runtime.

The exact meaning depends on the context, but this usually refers to the runtime of the .NET Framework. The CLR handles memory allocation and management. The CLR is also a virtual machine that not only executes apps but also generates and compiles code on-the-fly using a JIT compiler. The current Microsoft CLR implementation is Windows only.

CoreCLR

.NET Core Common Language Runtime.

This CLR is built from the same code base as the CLR. Originally, CoreCLR was the runtime of Silverlight and was designed to run on multiple platforms, specifically Windows and OS X. CoreCLR is now part of .NET Core and represents a simplified version of the CLR. It's still a cross-platform runtime, now including support for many Linux distributions. CoreCLR is also a virtual machine with JIT and code execution capabilities.

CoreFX

.NET Core Base Class Library (BCL)

A set of libraries that comprise the System.* (and to a limited extent Microsoft.*) namespaces. The BCL is a general purpose, lower-level framework that higher-level application frameworks, such as ASP.NET Core, build on. The source code of the .NET Core BCL is contained in the CoreFX repository. However, the majority of the .NET Core APIs are also available in the .NET Framework, so you can think of CoreFX as a fork of the .NET Framework BCL.

CoreRT

.NET Core runtime.

In contrast to the CLR/CoreCLR, CoreRT is not a virtual machine, which means it doesn't include the facilities to generate and run code on-the-fly because it doesn't include a JIT. It does, however, include the GC and the ability for runtime type identification (RTTI) and reflection. However, its type system is designed so that metadata for reflection isn't required. This enables having an AOT tool chain that can link away superfluous metadata and (more importantly) identify code that the app doesn't use. CoreRT is in development.

See Intro to .NET Native and CoreRT

cross-platform

The ability to develop and execute an application that can be used on multiple different operating systems, such as Linux, Windows and iOS, without having to re-write specifically for each one. This enables code re-use and consistency between applications on different platforms.

ecosystem

All of the runtime software, development tools, and community resources that are used to build and run applications for a given technology.

The term ".NET ecosystem" differs from similar terms such as ".NET stack" in its inclusion of third-party apps and libraries. Here's an example in a sentence:

- "The motivation behind the .NET Standard is to establish greater uniformity in the .NET ecosystem."

framework

In general, a comprehensive collection of APIs that facilitates development and deployment of applications that are based on a particular technology. In this general sense, ASP.NET Core and Windows Forms are examples of application frameworks. See also library.

The word "framework" has a more specific technical meaning in the following terms:

- .NET Framework
- target framework
- TFM (target framework moniker)

In the existing documentation, "framework" sometimes refers to an implementation of .NET. For example, an

article may call .NET Core a framework. We plan to eliminate this confusing usage from the documentation.

GC

Garbage collector.

The garbage collector is an implementation of automatic memory management. The GC frees memory occupied by objects that are no longer in use.

See Garbage Collection.

IL

Intermediate language.

Higher-level .NET languages, such as C#, compile down to a hardware-agnostic instruction set, which is called Intermediate Language (IL). IL is sometimes referred to as MSIL (Microsoft IL) or CIL (Common IL).

JIT

Just-in-time compiler.

Similar to AOT, this compiler translates IL to machine code that the processor understands. Unlike AOT, JIT compilation happens on demand and is performed on the same machine that the code needs to run on. Since JIT compilation occurs during execution of the application, compile time is part of the run time. Thus, JIT compilers have to balance time spent optimizing code against the savings that the resulting code can produce. But a JIT knows the actual hardware and can free developers from having to ship different implementations.

implementation of .NET

An implementation of .NET includes the following:

- One or more runtimes. Examples: CLR, CoreCLR, CoreRT.
- A class library that implements a version of the .NET Standard and may include additional APIs. Examples: .NET Framework Base Class Library, .NET Core Base Class Library.
- Optionally, one or more application frameworks. Examples: ASP.NET, Windows Forms, and WPF are included in the .NET Framework.
- Optionally, development tools. Some development tools are shared among multiple implementations.

Examples of .NET implementations:

- .NET Framework
- .NET Core
- Universal Windows Platform (UWP)

library

A collection of APIs that can be called by apps or other libraries. A .NET library is composed of one or more assemblies.

The words library and framework are often used synonymously.

metapackage

A NuGet package that has no library of its own but is only a list of dependencies. The included packages can optionally establish the API for a target framework.

See Packages, Metapackages and Frameworks

Mono

Mono is an open source, cross-platform .NET implementation that is mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, Mac, iOS, tvOS and watchOS and is focused primarily on apps that require a small footprint.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of the .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a just-in-time compiler, but it also features a full static compiler (ahead-of-time compilation) that is used on platforms like iOS.

To learn more about Mono, see the Mono documentation.

.NET

The umbrella term for .NET Standard and all .NET implementations and workloads. Always capitalized, never ".Net".

See the .NET Guide

.NET Core

A cross-platform, high-performance, open source implementation of .NET. Includes the Core Common Language Runtime (CoreCLR), the Core AOT Runtime (CoreRT, in development), the Core Base Class Library, and the Core SDK.

See .NET Core.

.NET Core CLI

A cross-platform toolchain for developing .NET Core applications.

See .NET Core command-line interface (CLI) tools.

.NET Core SDK

A set of libraries and tools that allow developers to create .NET Core applications and libraries. Includes the .NET Core CLI for building apps, .NET Core libraries and runtime for building and running apps, and the dotnet executable (*dotnet.exe*) that runs CLI commands and runs applications.

See .NET Core SDK Overview.

.NET Framework

An implementation of .NET that runs only on Windows. Includes the Common Language Runtime (CLR), the Base Class Library, and application framework libraries such as ASP.NET, Windows Forms, and WPF.

See .NET Framework Guide.

.NET Native

A compiler tool chain that produces native code ahead-of-time (AOT), as opposed to just-in-time (JIT).

Compilation happens on the developer's machine similar to the way a C++ compiler and linker works. It removes unused code and spends more time optimizing it. It extracts code from libraries and merges them into the executable. The result is a single module that represents the entire app.

UWP was the first application framework supported by .NET Native. Now, we support building native console apps for Windows, macOS, and Linux.

See Intro to .NET Native and CoreRT

.NET Standard

A formal specification of .NET APIs that are available in each .NET implementation.

The .NET Standard specification is sometimes called a library in the documentation. Because a library includes API implementations, not only specifications (interfaces), it's misleading to call .NET Standard a "library." We plan to eliminate that usage from the documentation, except in reference to the name of the .NET Standard metapackage (`NETStandard.Library`).

See .NET Standard.

NGEN

Native (image) generation.

You can think of this technology as a persistent JIT compiler. It usually compiles code on the machine where the code is executed, but compilation typically occurs at install time.

package

A NuGet package — or just a package — is a *.zip* file with one or more assemblies of the same name along with additional metadata such as the author name.

The *.zip* file has a *.nupkg* extension and may contain assets, such as *.dll* files and *.xml* files, for use with multiple target frameworks and versions. When installed in an app or library, the appropriate assets are selected based on the target framework specified by the app or library. The assets that define the interface are in the *ref* folder, and the assets that define the implementation are in the *lib* folder.

platform

An operating system and the hardware it runs on, such as Windows, macOS, Linux, iOS, and Android.

Here are examples of usage in sentences:

- ".NET Core is a cross-platform implementation of .NET."
- "PCL profiles represent Microsoft platforms, while the .NET Standard is agnostic to platform."

The .NET documentation frequently uses ".NET platform" to mean either an implementation of .NET or the .NET stack including all implementations. Both of these usages tend to get confused with the primary (OS/hardware) meaning, so we plan to eliminate these usages from the documentation.

runtime

The execution environment for a managed program.

The OS is part of the runtime environment but is not part of the .NET runtime. Here are some examples of .NET runtimes:

- Common Language Runtime (CLR)

- Core Common Language Runtime (CoreCLR)
- .NET Native (for UWP)
- Mono runtime

The .NET documentation sometimes uses "runtime" to mean an implementation of .NET. For example, in the following sentences "runtime" should be replaced with "implementation":

- "The various .NET runtimes implement specific versions of .NET Standard."
- "Libraries that are intended to run on multiple runtimes should target this framework." (referring to .NET Standard)
- "The various .NET runtimes implement specific versions of .NET Standard. ... Each .NET runtime version advertises the highest .NET Standard version it supports ..."

We plan to eliminate this inconsistent usage.

stack

A set of programming technologies that are used together to build and run applications.

"The .NET stack" refers to the .NET Standard and all .NET implementations. The phrase "a .NET stack" may refer to one implementation of .NET.

target framework

The collection of APIs that a .NET app or library relies on.

An app or library can target a version of .NET Standard (for example, .NET Standard 2.0), which is specification for a standardized set of APIs across all .NET implementations. An app or library can also target a version of a specific .NET implementation, in which case it gets access to implementation-specific APIs. For example, an app that targets Xamarin.iOS gets access to Xamarin-provided iOS API wrappers.

For some target frameworks (for example, the .NET Framework) the available APIs are defined by the assemblies that a .NET implementation installs on a system, which may include application framework APIs (for example, ASP.NET, WinForms). For package-based target frameworks (such as .NET Standard and .NET Core), the framework APIs are defined by the packages installed in the app or library. In that case, the target framework implicitly specifies a metapackage that references all the packages that together make up the framework.

See Target Frameworks.

TFM

Target framework moniker.

A standardized token format for specifying the target framework of a .NET app or library. Target frameworks are typically referenced by a short name, such as `net462`. Long-form TFMs (such as .NETFramework,Version=4.6.2) exist but are not generally used to specify a target framework.

See Target Frameworks.

UWP

Universal Windows Platform.

An implementation of .NET that is used for building modern, touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phablets, phones, and even the Xbox. UWP provides many services, such as a centralized

app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, VB.NET, and JavaScript. When using C# and VB.NET, the .NET APIs are provided by .NET Core.

See also

- .NET Guide
- .NET Framework Guide
- .NET Core
- ASP.NET Overview
- ASP.NET Core Overview

Open-source library guidance

5/15/2019 • 2 minutes to read • Edit Online

This guidance provides recommendations for developers to create high-quality .NET libraries. This documentation focuses on the *what* and the *why* when building a .NET library, not the *how*.

Aspects of high-quality open-source .NET libraries:

- **Inclusive** - Good .NET libraries strive to support many platforms, programming languages, and applications.
- **Stable** - Good .NET libraries coexist in the .NET ecosystem, running in applications built with many libraries.
- **Designed to evolve** - .NET libraries should improve and evolve over time, while supporting existing users.
- **Debuggable** - .NET libraries should use the latest tools to create a great debugging experience for users.
- **Trusted** - .NET libraries have developers' trust by publishing to NuGet using security best practices.

Get Started

Types of recommendations

Each article presents four types of recommendations: **Do**, **Consider**, **Avoid**, and **Do not**. The type of recommendation indicates how strongly it should be followed.

You should almost always follow a **Do** recommendation. For example:

✔ **DO** distribute your library using a NuGet package.

On the other hand, **Consider** recommendations should generally be followed, but there are legitimate exceptions to the rule and you shouldn't feel bad about not following the guidance:

✔ **CONSIDER** using SemVer 2.0.0 to version your NuGet package.

Avoid recommendations mention things that are generally not a good idea, but breaking the rule sometimes makes sense:

AVOID NuGet package references that demand an exact version.

And finally, **Do not** recommendations indicate something you should almost never do:

DO NOT publish strong-named and non-strong-named versions of your library. For example, `Contoso.Api` and `Contoso.Api.StrongNamed`.

NEXT

Choosing between .NET Core and .NET Framework for server apps

7/30/2019 • 6 minutes to read • Edit Online

There are two supported implementations for building server-side applications with .NET: .NET Framework and .NET Core. Both share many of the same components and you can share code across the two. However, there are fundamental differences between the two and your choice depends on what you want to accomplish. This article provides guidance on when to use each.

Use .NET Core for your server application when:

- You have cross-platform needs.
- You are targeting microservices.
- You are using Docker containers.
- You need high-performance and scalable systems.
- You need side-by-side .NET versions per application.

Use .NET Framework for your server application when:

- Your app currently uses .NET Framework (recommendation is to extend instead of migrating).
- Your app uses third-party .NET libraries or NuGet packages not available for .NET Core.
- Your app uses .NET technologies that aren't available for .NET Core.
- Your app uses a platform that doesn't support .NET Core.

When to choose .NET Core

The following sections give a more detailed explanation of the previously stated reasons for picking .NET Core.

Cross-platform needs

If your application (web/service) needs to run on multiple platforms (Windows, Linux, and macOS), use .NET Core.

.NET Core supports the previously mentioned operating systems as your development workstation. Visual Studio provides an Integrated Development Environment (IDE) for Windows and macOS. You can also use Visual Studio Code, which runs on macOS, Linux, and Windows. Visual Studio Code supports .NET Core, including IntelliSense and debugging. Most third-party editors, such as Sublime, Emacs, and VI, work with .NET Core. These third-party editors get editor IntelliSense using Omnisharp. You can also avoid any code editor and directly use the .NET Core CLI tools, available for all supported platforms.

Microservices architecture

A microservices architecture allows a mix of technologies across a service boundary. This technology mix enables a gradual embrace of .NET Core for new microservices that work with other microservices or services. For example, you can mix microservices or services developed with .NET Framework, Java, Ruby, or other monolithic technologies.

There are many infrastructure platforms available. Azure Service Fabric is designed for large and complex microservice systems. Azure App Service is a good choice for stateless microservices. Microservices alternatives based on Docker fit any kind of microservices approach, as explained in the Containers section. All these platforms support .NET Core and make them ideal for hosting your microservices.

For more information about microservices architecture, see .NET Microservices. Architecture for Containerized

.NET Applications.

Containers

Containers are commonly used in conjunction with a microservices architecture. Containers can also be used to containerize web apps or services that follow any architectural pattern. .NET Framework can be used on Windows containers, but the modularity and lightweight nature of .NET Core makes it a better choice for containers. When creating and deploying a container, the size of its image is much smaller with .NET Core than with .NET Framework. Because it's cross-platform, you can deploy server apps to Linux Docker containers, for example.

Docker containers can be hosted in your own Linux or Windows infrastructure, or in a cloud service such as Azure Kubernetes Service. Azure Kubernetes Service can manage, orchestrate, and scale container-based applications in the cloud.

A need for high-performance and scalable systems

When your system needs the best possible performance and scalability, .NET Core and ASP.NET Core are your best options. High-performance server runtime for Windows Server and Linux makes .NET a top performing web framework on TechEmpower benchmarks.

Performance and scalability are especially relevant for microservices architectures, where hundreds of microservices may be running. With ASP.NET Core, systems run with a much lower number of servers/Virtual Machines (VM). The reduced servers/VMs save costs in infrastructure and hosting.

A need for side by side of .NET versions per application level

To install applications with dependencies on different versions of .NET, we recommend .NET Core. .NET Core offers side-by-side installation of different versions of the .NET Core runtime on the same machine. This side-by-side installation allows multiple services on the same server, each of them on its own version of .NET Core. It also lowers risks and saves money in application upgrades and IT operations.

When to choose .NET Framework

.NET Core offers significant benefits for new applications and application patterns. However, the .NET Framework continues to be the natural choice for many existing scenarios and as such the .NET Framework isn't replaced by .NET Core for all server applications.

Current .NET Framework applications

In most cases, you don't need to migrate your existing applications to .NET Core. Instead, a recommended approach is to use .NET Core as you extend an existing application, such as writing a new web service in ASP.NET Core.

A need to use third-party .NET libraries or NuGet packages not available for .NET Core

Libraries are quickly embracing .NET Standard. .NET Standard enables sharing code across all .NET implementations including .NET Core. With .NET Standard 2.0, this is even easier:

- The API surface became much larger.
- Introduced a .NET Framework compatibility mode. This compatibility mode allows .NET Standard/.NET Core projects to reference .NET Framework libraries. To learn more about the compatibility mode, see Announcing .NET Standard 2.0.

So only in cases where the libraries or NuGet packages use technologies that aren't available in .NET Standard/.NET Core, you need to use the .NET Framework.

A need to use .NET technologies not available for .NET Core

Some .NET Framework technologies aren't available in .NET Core. Some of them might be available in later .NET Core releases. Others don't apply to the new application patterns targeted by .NET Core and may never be available. The following list shows the most common technologies not found in .NET Core:

- ASP.NET Web Forms applications: ASP.NET Web Forms are only available in the .NET Framework. ASP.NET Core cannot be used for ASP.NET Web Forms. There are no plans to bring ASP.NET Web Forms to .NET Core.
- ASP.NET Web Pages applications: ASP.NET Web Pages aren't included in ASP.NET Core.
- WCF services implementation. Even when there's a WCF-Client library to consume WCF services from .NET Core, WCF server implementation is currently only available in the .NET Framework. This scenario is not part of the current plan for .NET Core but it's being considered for the future.
- Workflow-related services: Windows Workflow Foundation (WF), Workflow Services (WCF + WF in a single service) and WCF Data Services (formerly known as "ADO.NET Data Services") are only available in the .NET Framework. There are no plans to bring WF/WCF+WF/WCF Data Services to .NET Core.
- Language support: Visual Basic and F# are currently supported in .NET Core, but not for all project types. For a list of supported project templates, see Template options for dotnet new.

In addition to the official roadmap, there are other frameworks to be ported to .NET Core. For a full list, see the CoreFX issues marked as port-to-core. This list doesn't represent a commitment from Microsoft to bring those components to .NET Core. They're simply capturing the desire from the community to do so. If you care about any of the components marked as `port-to-core`, participate in the discussions on GitHub. And if you think something is missing, file a new issue in the CoreFX repository.

A need to use a platform that doesn't support .NET Core

Some Microsoft or third-party platforms don't support .NET Core. Some Azure services provide an SDK not yet available for consumption on .NET Core. This is a transitional circumstance, as all of Azure services use .NET Core. In the meantime, you can always use the equivalent REST API instead of the client SDK.

See also

- Choose between ASP.NET and ASP.NET Core
- ASP.NET Core targeting .NET Framework
- Target frameworks
- .NET Core Guide
- Porting from .NET Framework to .NET Core
- Introduction to .NET and Docker
- .NET Components Overview
- .NET Microservices. Architecture for Containerized .NET Applications

What is "managed code"?

3/8/2019 • 2 minutes to read • Edit Online

When working with .NET Framework, you will often encounter the term "managed code". This document will explain what this term means and additional information around it.

To put it very simply, managed code is just that: code whose execution is managed by a runtime. In this case, the runtime in question is called the **Common Language Runtime** or CLR, regardless of the implementation (Mono or .NET Framework or .NET Core). CLR is in charge of taking the managed code, compiling it into machine code and then executing it. On top of that, runtime provides several important services such as automatic memory management, security boundaries, type safety etc.

Contrast this to the way you would run a C/C++ program, also called "unmanaged code". In the unmanaged world, the programmer is in charge of pretty much everything. The actual program is, essentially, a binary that the operating system (OS) loads into memory and starts. Everything else, from memory management to security considerations are a burden of the programmer.

Managed code is written in one of the high-level languages that can be run on top of .NET, such as C#, Visual Basic, F# and others. When you compile code written in those languages with their respective compiler, you don't get machine code. You get **Intermediate Language** code which the runtime then compiles and executes. C++ is the one exception to this rule, as it can also produce native, unmanaged binaries that run on Windows.

Intermediate Language & execution

What is "Intermediate Language" (or IL for short)? It is a product of compilation of code written in high-level .NET languages. Once you compile your code written in one of these languages, you will get a binary that is made out of IL. It is important to note that the IL is independent from any specific language that runs on top of the runtime; there is even a separate specification for it that you can read if you're so inclined.

Once you produce IL from your high-level code, you will most likely want to run it. This is where the CLR takes over and starts the process of **Just-In-Time** compiling, or **JIT-ing** your code from IL to machine code that can actually be run on a CPU. In this way, the CLR knows exactly what your code is doing and can effectively *manage* it.

Intermediate Language is sometimes also called Common Intermediate Language (CIL) or Microsoft Intermediate Language (MSIL).

Unmanaged code interoperability

Of course, the CLR allows passing the boundaries between managed and unmanaged world, and there is a lot of code that does that, even in the Base Class Libraries. This is called **interoperability** or just **interop** for short. These provisions would allow you to, for example, wrap up an unmanaged library and call into it. However, it is important to note that once you do this, when the code passes the boundaries of the runtime, the actual management of the execution is again in the hand of unmanaged code, and thus falls under the same restrictions.

Similar to this, C# is one language that allows you to use unmanaged constructs such as pointers directly in code by utilizing what is known as **unsafe context** which designates a piece of code for which the execution is not managed by the CLR.

More resources

- Overview of the .NET Framework

- Unsafe Code and Pointers
- Native interoperability

Automatic Memory Management

1/23/2019 • 7 minutes to read • Edit Online

Automatic memory management is one of the services that the Common Language Runtime provides during Managed Execution. The Common Language Runtime's garbage collector manages the allocation and release of memory for an application. For developers, this means that you do not have to write code to perform memory management tasks when you develop managed applications. Automatic memory management can eliminate common problems, such as forgetting to free an object and causing a memory leak, or attempting to access memory for an object that has already been freed. This section describes how the garbage collector allocates and releases memory.

Allocating Memory

When you initialize a new process, the runtime reserves a contiguous region of address space for the process. This reserved address space is called the managed heap. The managed heap maintains a pointer to the address where the next object in the heap will be allocated. Initially, this pointer is set to the managed heap's base address. All reference types are allocated on the managed heap. When an application creates the first reference type, memory is allocated for the type at the base address of the managed heap. When the application creates the next object, the garbage collector allocates memory for it in the address space immediately following the first object. As long as address space is available, the garbage collector continues to allocate space for new objects in this manner.

Allocating memory from the managed heap is faster than unmanaged memory allocation. Because the runtime allocates memory for an object by adding a value to a pointer, it is almost as fast as allocating memory from the stack. In addition, because new objects that are allocated consecutively are stored contiguously in the managed heap, an application can access the objects very quickly.

Releasing Memory

The garbage collector's optimizing engine determines the best time to perform a collection based on the allocations being made. When the garbage collector performs a collection, it releases the memory for objects that are no longer being used by the application. It determines which objects are no longer being used by examining the application's roots. Every application has a set of roots. Each root either refers to an object on the managed heap or is set to null. An application's roots include static fields, local variables and parameters on a thread's stack, and CPU registers. The garbage collector has access to the list of active roots that the just-in-time (JIT) compiler and the runtime maintain. Using this list, it examines an application's roots, and in the process creates a graph that contains all the objects that are reachable from the roots.

Objects that are not in the graph are unreachable from the application's roots. The garbage collector considers unreachable objects garbage and will release the memory allocated for them. During a collection, the garbage collector examines the managed heap, looking for the blocks of address space occupied by unreachable objects. As it discovers each unreachable object, it uses a memory-copying function to compact the reachable objects in memory, freeing up the blocks of address spaces allocated to unreachable objects. Once the memory for the reachable objects has been compacted, the garbage collector makes the necessary pointer corrections so that the application's roots point to the objects in their new locations. It also positions the managed heap's pointer after the last reachable object. Note that memory is compacted only if a collection discovers a significant number of unreachable objects. If all the objects in the managed heap survive a collection, then there is no need for memory compaction.

To improve performance, the runtime allocates memory for large objects in a separate heap. The garbage collector automatically releases the memory for large objects. However, to avoid moving large objects in memory, this

memory is not compacted.

Generations and Performance

To optimize the performance of the garbage collector, the managed heap is divided into three generations: 0, 1, and 2. The runtime's garbage collection algorithm is based on several generalizations that the computer software industry has discovered to be true by experimenting with garbage collection schemes. First, it is faster to compact the memory for a portion of the managed heap than for the entire managed heap. Secondly, newer objects will have shorter lifetimes and older objects will have longer lifetimes. Lastly, newer objects tend to be related to each other and accessed by the application around the same time.

The runtime's garbage collector stores new objects in generation 0. Objects created early in the application's lifetime that survive collections are promoted and stored in generations 1 and 2. The process of object promotion is described later in this topic. Because it is faster to compact a portion of the managed heap than the entire heap, this scheme allows the garbage collector to release the memory in a specific generation rather than release the memory for the entire managed heap each time it performs a collection.

In reality, the garbage collector performs a collection when generation 0 is full. If an application attempts to create a new object when generation 0 is full, the garbage collector discovers that there is no address space remaining in generation 0 to allocate for the object. The garbage collector performs a collection in an attempt to free address space in generation 0 for the object. The garbage collector starts by examining the objects in generation 0 rather than all objects in the managed heap. This is the most efficient approach, because new objects tend to have short lifetimes, and it is expected that many of the objects in generation 0 will no longer be in use by the application when a collection is performed. In addition, a collection of generation 0 alone often reclaims enough memory to allow the application to continue creating new objects.

After the garbage collector performs a collection of generation 0, it compacts the memory for the reachable objects as explained in Releasing Memory earlier in this topic. The garbage collector then promotes these objects and considers this portion of the managed heap generation 1. Because objects that survive collections tend to have longer lifetimes, it makes sense to promote them to a higher generation. As a result, the garbage collector does not have to reexamine the objects in generations 1 and 2 each time it performs a collection of generation 0.

After the garbage collector performs its first collection of generation 0 and promotes the reachable objects to generation 1, it considers the remainder of the managed heap generation 0. It continues to allocate memory for new objects in generation 0 until generation 0 is full and it is necessary to perform another collection. At this point, the garbage collector's optimizing engine determines whether it is necessary to examine the objects in older generations. For example, if a collection of generation 0 does not reclaim enough memory for the application to successfully complete its attempt to create a new object, the garbage collector can perform a collection of generation 1, then generation 2. If this does not reclaim enough memory, the garbage collector can perform a collection of generations 2, 1, and 0. After each collection, the garbage collector compacts the reachable objects in generation 0 and promotes them to generation 1. Objects in generation 1 that survive collections are promoted to generation 2. Because the garbage collector supports only three generations, objects in generation 2 that survive a collection remain in generation 2 until they are determined to be unreachable in a future collection.

Releasing Memory for Unmanaged Resources

For the majority of the objects that your application creates, you can rely on the garbage collector to automatically perform the necessary memory management tasks. However, unmanaged resources require explicit cleanup. The most common type of unmanaged resource is an object that wraps an operating system resource, such as a file handle, window handle, or network connection. Although the garbage collector is able to track the lifetime of a managed object that encapsulates an unmanaged resource, it does not have specific knowledge about how to clean up the resource. When you create an object that encapsulates an unmanaged resource, it is recommended that you provide the necessary code to clean up the unmanaged resource in a public **Dispose** method. By providing a **Dispose** method, you enable users of your object to explicitly free its memory when they are finished with the

object. When you use an object that encapsulates an unmanaged resource, you should be aware of **Dispose** and call it as necessary. For more information about cleaning up unmanaged resources and an example of a design pattern for implementing **Dispose**, see Garbage Collection.

See also

- GC
- Garbage Collection
- Managed Execution Process

Common Language Runtime (CLR) overview

5/15/2019 • 4 minutes to read • Edit Online

The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier.

Compilers and tools expose the common language runtime's functionality and enable you to write code that benefits from this managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code; it benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

> **NOTE**
>
> Compilers and tools are able to produce output that the common language runtime can consume because the type system, the format of metadata, and the runtime environment (the virtual execution system) are all defined by a public standard, the ECMA Common Language Infrastructure specification. For more information, see ECMA C# and Common Language Infrastructure Specifications.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks as well as some other common programming errors. If your code is managed, you can use managed data, unmanaged data, or both managed and unmanaged data in your .NET Framework application. Because language compilers supply their own types, such as primitive types, you might not always know (or need to know) whether your data is being managed.

The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For example, you can define a class and then use a different language to derive a class from your original class or call a method on the original class. You can also pass an instance of a class to a method of a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime, and they follow the runtime's rules for defining new types, as well as for creating, using, persisting, and binding to types.

As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs, which makes your code less likely to break because of some unmet dependency. Registration information and state data are no longer stored in the registry where they can be difficult to establish and maintain. Instead, information about the types you define (and their dependencies) is stored with the code as metadata, making the tasks of component replication and removal much less complicated.

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers. This means that some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. For example, if you are a Visual Basic developer, you might notice that with the common language runtime, the

Visual Basic language has more object-oriented features than before. The runtime provides the following benefits:

- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- Language features such as inheritance, interfaces, and overloading for object-oriented programming.
- Support for explicit free threading that allows creation of multithreaded, scalable applications.
- Support for structured exception handling.
- Support for custom attributes.
- Garbage collection.
- Use of delegates instead of function pointers for increased type safety and security. For more information about delegates, see Common Type System.

CLR versions

The version number of the .NET Framework doesn't necessarily correspond to the version number of the CLR it includes. The following table shows how the two version numbers correlate:

.NET FRAMEWORK VERSION	INCLUDES CLR VERSION
1.0	1.0
1.1	1.1
2.0	2.0
3.0	2.0
3.5	2.0
4	4
4.5 (including 4.5.1 and 4.5.2)	4
4.6 (including 4.6.1 and 4.6.2)	4
4.7 (including 4.7.1 and 4.7.2)	4
4.8	4

Related topics

TITLE	DESCRIPTION
Managed Execution Process	Describes the steps required to take advantage of the common language runtime.

TITLE	DESCRIPTION
Automatic Memory Management	Describes how the garbage collector allocates and releases memory.
Overview of the .NET Framework	Describes key .NET Framework concepts such as the common type system, cross-language interoperability, managed execution, application domains, and assemblies.
Common Type System	Describes how types are declared, used, and managed in the runtime in support of cross-language integration.

See also

- Versions and Dependencies

Language independence and language-independent components

7/9/2019 • 69 minutes to read • Edit Online

.NET is language independent. This means that, as a developer, you can develop in one of the many languages that target .NET implementations, such as C#, F#, and Visual Basic. You can access the types and members of class libraries developed for .NET implementations without having to know the language in which they were originally written and without having to follow any of the original language's conventions. If you are a component developer, your component can be accessed by any .NET app regardless of its language.

> **NOTE**
>
> This first part of this article discusses creating language-independent components - that is, components that can be consumed by apps that are written in any language. You can also create a single component or app from source code written in multiple languages; see Cross-Language Interoperability in the second part of this article.

To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the Common Language Specification (CLS), which is a set of rules that apply to generated assemblies. The Common Language Specification is defined in Partition I, Clauses 7 through 11 of the ECMA-335 Standard: Common Language Infrastructure.

If your component conforms to the Common Language Specification, it is guaranteed to be CLS-compliant and can be accessed from code in assemblies written in any programming language that supports the CLS. You can determine whether your component conforms to the Common Language Specification at compile time by applying the CLSCompliantAttribute attribute to your source code. For more information, see The CLSCompliantAttribute attribute.

In this article:

- CLS compliance rules
  - Types and type member signatures
  - Naming conventions
  - Type conversion
  - Arrays
  - Interfaces
  - Enumerations
  - Type members in general
  - Member accessibility
  - Generic types and members
  - Constructors
  - Properties
  - Events

- Overloads
- Exceptions
- Attributes

- CLSCompliantAttribute attribute
- Cross-Language Interoperability

CLS compliance rules

This section discusses the rules for creating a CLS-compliant component. For a complete list of rules, see Partition I, Clause 11 of the ECMA-335 Standard: Common Language Infrastructure.

> **NOTE**
> The Common Language Specification discusses each rule for CLS compliance as it applies to consumers (developers who are programmatically accessing a component that is CLS-compliant), frameworks (developers who are using a language compiler to create CLS-compliant libraries), and extenders (developers who are creating a tool such as a language compiler or a code parser that creates CLS-compliant components). This article focuses on the rules as they apply to frameworks. Note, though, that some of the rules that apply to extenders may also apply to assemblies that are created using Reflection.Emit.

To design a component that is language independent, you only need to apply the rules for CLS compliance to your component's public interface. Your private implementation does not have to conform to the specification.

> **IMPORTANT**
> The rules for CLS compliance apply only to a component's public interface, not to its private implementation.

For example, unsigned integers other than Byte are not CLS-compliant. Because the `Person` class in the following example exposes an `Age` property of type UInt16, the following code displays a compiler warning.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As UInt16
      Get
         Return personAge
      End Get
   End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

You can make the Person class CLS-compliant by changing the type of `Age` property from `UInt16` to Int16, which is a CLS-compliant, 16-bit signed integer. You do not have to change the type of the private `personAge` field.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
   { get { return personAge; } }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As Int16
      Get
         Return CType(personAge, Int16)
      End Get
   End Property
End Class
```

A library's public interface consists of the following:

- Definitions of public classes.
- Definitions of the public members of public classes, and definitions of members accessible to derived classes (that is, protected members).
- Parameters and return types of public methods of public classes, and parameters and return types of methods accessible to derived classes.

The rules for CLS compliance are listed in the following table. The text of the rules is taken verbatim from the ECMA-335 Standard: Common Language Infrastructure, which is Copyright 2012 by Ecma International. More detailed information about these rules is found in the following sections.

CATEGORY	SEE	RULE	RULE NUMBER
Accessibility	Member accessibility	Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility `family-or-assembly`. In this case, the override shall have accessibility `family`.	10
Accessibility	Member accessibility	The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic argument whose type is visible only within the assembly.	12
Arrays	Arrays	Arrays shall have elements with a CLS-compliant type, and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types.	16
Attributes	Attributes	Attributes shall be of type System.Attribute, or a type inheriting from it.	41

CATEGORY	SEE	RULE	RULE NUMBER
Attributes	Attributes	The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are (see Partition IV): System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double, and any enumeration type based on a CLS-compliant base integer type.	34
Attributes	Attributes	The CLS does not allow publicly visible required modifiers (`modreq` , see Partition II), but does allow optional modifiers (`modopt` , see Partition II) it does not understand.	35
Constructors	Constructors	An object constructor shall call some instance constructor of its base class before any access occurs to inherited instance data. (This does not apply to value types, which need not have constructors.)	21
Constructors	Constructors	An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice.	22
Enumerations	Enumerations	The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be "value__", and that field shall be marked `RTSpecialName` .	7

CATEGORY	SEE	RULE	RULE NUMBER
Enumerations	Enumerations	There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute (see Partition IV Library) custom attribute. One represents named integer values; the other represents named bit flags that can be combined to generate an unnamed value. The value of an `enum` is not limited to the specified values.	8
Enumerations	Enumerations	Literal static fields of an enum shall have the type of the enum itself.	9
Events	Events	The methods that implement an event shall be marked `SpecialName` in the metadata.	29
Events	Events	The accessibility of an event and of its accessors shall be identical.	30
Events	Events	The `add` and `remove` methods for an event shall both either be present or absent.	31
Events	Events	The `add` and `remove` methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from System.Delegate.	32
Events	Events	Events shall adhere to a specific naming pattern. The SpecialName attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.	33
Exceptions	Exceptions	Objects that are thrown shall be of type System.Exception or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions.	40

CATEGORY	SEE	RULE	RULE NUMBER
General	CLS compliance rules	CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly.	1
General	CLS compliance rules	Members of non-CLS compliant types shall not be marked CLS-compliant.	2
Generics	Generic types and members	Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type.	42
Generics	Generic types and members	The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above.	43
Generics	Generic types and members	A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints.	44
Generics	Generic types and members	Types used as constraints on generic parameters shall themselves be CLS-compliant.	45
Generics	Generic types and members	The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply.	46
Generics	Generic types and members	For each abstract or virtual generic method, there shall be a default concrete (nonabstract) implementation	47

CATEGORY	SEE	RULE	RULE NUMBER
Interfaces	Interfaces	CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them.	18
Interfaces	Interfaces	CLS-compliant interfaces shall not define static methods, nor shall they define fields.	19
Members	Type members in general	Global static fields and methods are not CLS-compliant.	36
Members	--	The value of a literal static is specified through the use of field initialization metadata. A CLS-compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an `enum`).	13
Members	Type members in general	The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.	15
Naming conventions	Naming conventions	Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard3.0 governing the set of characters permitted to start and be included in identifiers, available online at Unicode Normalization Forms. Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-to-one lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used.	4

CATEGORY	SEE	RULE	RULE NUMBER
Overloading	Naming conventions	All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not.	5
Overloading	Naming conventions	Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type,except as specified in CLS Rule 39	6
Overloading	Overloads	Only properties and methods can be overloaded.	37

CATEGORY	SEE	RULE	RULE NUMBER
Overloading	Overloads	Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit`, which can also be overloaded based on their return type.	38
Overloading	--	If two or more CLS-compliant methods declared in a type have the same name and, for a specific set of type instantiations, they have the same parameter and return types, then all these methods shall be semantically equivalent at those type instantiations.	48
Properties	Properties	The methods that implement the getter and setter methods of a property shall be marked `SpecialName` in the metadata.	24
Properties	Properties	A property's accessors shall all be static, all be virtual, or all be instance.	26
Properties	Properties	The type of a property shall be the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property shall be the types of the parameters to the getter and the types of all but the final parameter of the setter. All of these types shall be CLS-compliant, and shall not be managed pointers (that is, shall not be passed by reference).	27
Properties	Properties	Properties shall adhere to a specific naming pattern. The `SpecialName` attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both.	28

CATEGORY	SEE	RULE	RULE NUMBER
Type conversion	Type conversion	If either op_Implicit or op_Explicit is provided, an alternate means of providing the coercion shall be provided.	39
Types	Types and type member signatures	Boxed value types are not CLS-compliant.	3
Types	Types and type member signatures	All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant.	11
Types	Types and type member signatures	Typed references are not CLS-compliant.	14
Types	Types and type member signatures	Unmanaged pointer types are not CLS-compliant.	17
Types	Types and type member signatures	CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members	20
Types	Types and type member signatures	System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.	23

Types and type member signatures

The System.Object type is CLS-compliant and is the base type of all object types in the .NET Framework type system. Inheritance in the .NET Framework is either implicit (for example, the String class implicitly inherits from the `Object` class) or explicit (for example, the CultureNotFoundException class explicitly inherits from the ArgumentException class, which explicitly inherits from the Exception class. For a derived type to be CLS compliant, its base type must also be CLS-compliant.

The following example shows a derived type whose base type is not CLS-compliant. It defines a base `Counter` class that uses an unsigned 32-bit integer as a counter. Because the class provides counter functionality by wrapping an unsigned integer, the class is marked as non-CLS-compliant. As a result, a derived class, `NonZeroCounter`, is also not CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return String.Format("{0}). ", ctr);
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
   {
   }

   private NonZeroCounter(UInt32 startIndex) : base(startIndex)
   {
   }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter' is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
   Dim ctr As UInt32

   Public Sub New
      ctr = 0
   End Sub

   Protected Sub New(ctr As UInt32)
      ctr = ctr
   End Sub

   Public Overrides Function ToString() As String
      Return String.Format("{0}). ", ctr)
   End Function

   Public ReadOnly Property Value As UInt32
      Get
         Return ctr
      End Get
   End Property

   Public Sub Increment()
      ctr += CType(1, UInt32)
   End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
   Public Sub New(startIndex As Integer)
      MyClass.New(CType(startIndex, UInt32))
   End Sub

   Private Sub New(startIndex As UInt32)
      MyBase.New(CType(startIndex, UInt32))
   End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

All types that appear in member signatures, including a method's return type or a property type, must be CLS-compliant. In addition, for generic types:

- All types that compose an instantiated generic type must be CLS-compliant.
- All types used as constraints on generic parameters must be CLS-compliant.

The .NET common type system includes a number of built-in types that are supported directly by the common language runtime and are specially encoded in an assembly's metadata. Of these intrinsic types, the types listed in the following table are CLS-compliant.

CLS-COMPLIANT TYPE	DESCRIPTION
Byte	8-bit unsigned integer
Int16	16-bit signed integer

CLS-COMPLIANT TYPE	DESCRIPTION
Int32	32-bit signed integer
Int64	64-bit signed integer
Single	Single-precision floating-point value
Double	Double-precision floating-point value
Boolean	true or false value type
Char	UTF-16 encoded code unit
Decimal	Non-floating-point decimal number
IntPtr	Pointer or handle of a platform-defined size
String	Collection of zero, one, or more Char objects

The intrinsic types listed in the following table are not CLS-Compliant.

NON-COMPLIANT TYPE	DESCRIPTION	CLS-COMPLIANT ALTERNATIVE
SByte	8-bit signed integer data type	Int16
UInt16	16-bit unsigned integer	Int32
UInt32	32-bit unsigned integer	Int64
UInt64	64-bit unsigned integer	Int64 (may overflow), BigInteger, or Double
UIntPtr	Unsigned pointer or handle	IntPtr

The .NET Framework Class Library or any other class library may include other types that aren't CLS-compliant; for example:

- Boxed value types. The following C# example creates a class that has a public property of type `int*` named `Value`. Because an `int*` is a boxed value type, the compiler flags it as non-CLS-compliant.

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- Typed references, which are special constructs that contain a reference to an object and a reference to a type.

If a type is not CLS-compliant, you should apply the CLSCompliantAttribute attribute with an *isCompliant* parameter with a value of `false` to it. For more information, see the CLSCompliantAttribute attribute section.

The following example illustrates the problem of CLS compliance in a method signature and in generic type instantiation. It defines an `InvoiceItem` class with a property of type UInt32, a property of type Nullable(Of UInt32), and a constructor with parameters of type `UInt32` and `Nullable(Of UInt32)`. You get four compiler warnings when you try to compile this example.

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of UInteger)

   Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
      itemId = sku
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of UInteger)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As UInteger
      Get
         Return invId
      End Get
      Set
         invId = value
      End Set
   End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                             ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not CLS-compliant.
'
'       Public Property InvoiceId As UInteger
```

To eliminate the compiler warnings, replace the non-CLS-compliant types in the `InvoiceItem` public interface with compliant types:

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is zero or negative.");
         invId = (uint) value; }
   }
}
```

```
Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of Integer)

   Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
      If sku <= 0 Then
         Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
      End If
      itemId = CUInt(sku)
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of Integer)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As Integer
      Get
         Return CInt(invId)
      End Get
      Set
         invId = CUInt(value)
      End Set
   End Property
End Class
```

In addition to the specific types listed, some categories of types are not CLS compliant. These include unmanaged pointer types and function pointer types. The following example generates a compiler warning because it uses a pointer to an integer to create an array of integers.

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

For CLS-compliant abstract classes (that is, classes marked as `abstract` in C#), all members of the class must also be CLS-compliant.

Naming conventions

Because some programming languages are case-insensitive, identifiers (such as the names of namespaces, types, and members) must differ by more than case. Two identifiers are considered equivalent if their lowercase mappings are the same. The following C# example defines two public classes, `Person` and `person`. Because they differ only by case, the C# compiler flags them as not CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{

}

public class person
{

}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Programming language identifiers, such as the names of namespaces, types, and members, must conform to the Unicode Standard 3.0, Technical Report 15, Annex 7. This means that:

- The first character of an identifier can be any Unicode uppercase letter, lowercase letter, title case letter, modifier letter, other letter, or letter number. For information on Unicode character categories, see the System.Globalization.UnicodeCategory enumeration.
- Subsequent characters can be from any of the categories as the first character, and can also include non-spacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes.

Before you compare identifiers, you should filter out formatting codes and convert the identifiers to Unicode Normalization Form C, because a single character can be represented by multiple UTF-16-encoded code units.

Character sequences that produce the same code units in Unicode Normalization Form C are not CLS-compliant. The following example defines a property named `Å`, which consists of the character ANGSTROM SIGN (U+212B), and a second property named `Å` which consists of the character LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5). The C# compiler flags the source code as non-CLS-compliant.

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>
Public Class Size
   Private a1 As Double
   Private a2 As Double

   Public Property Å As Double
       Get
          Return a1
       End Get
       Set
          a1 = value
       End Set
   End Property

   Public Property Å As Double
       Get
          Return a2
       End Get
       Set
          a2 = value
       End Set
   End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

Member names within a particular scope (such as the namespaces within an assembly, the types within a

namespace, or the members within a type) must be unique except for names that are resolved through overloading. This requirement is more stringent than that of the common type system, which allows multiple members within a scope to have identical names as long as they are different kinds of members (for example, one is a method and one is a field). In particular, for type members:

- Fields and nested types are distinguished by name alone.
- Methods, properties, and events that have the same name must differ by more than just return type.

The following example illustrates the requirement that member names must be unique within their scope. It defines a class named `Converter` that includes four members named `Conversion`. Three are methods, and one is a property. The method that includes an `Int64` parameter is uniquely named, but the two methods with an `Int32` parameter are not, because return value is not considered a part of a member's signature. The `Conversion` property also violates this requirement, because properties cannot have the same name as overloaded methods.

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a member called
//             'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains a definition for
//             'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Converter
   Public Function Conversion(number As Integer) As Double
      Return CDbl(number)
   End Function

   Public Function Conversion(number As Integer) As Single
      Return CSng(number)
   End Function

   Public Function Conversion(number As Long) As Double
      Return CDbl(number)
   End Function

   Public ReadOnly Property Conversion As Boolean
      Get
         Return True
      End Get
   End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As Double'
'                    and 'Public Function Conversion(number As Integer) As Single' cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

Individual languages include unique keywords, so languages that target the common language runtime must also provide some mechanism for referencing identifiers (such as type names) that coincide with keywords. For example, `case` is a keyword in both C# and Visual Basic. However, the following Visual Basic example is able to disambiguate a class named `case` from the `case` keyword by using opening and closing braces. Otherwise, the example would produce the error message, "Keyword is not valid as an identifier," and fail to compile.

```
Public Class [case]
   Private _id As Guid
   Private name As String

   Public Sub New(name As String)
      _id = Guid.NewGuid()
      Me.name = name
   End Sub

   Public ReadOnly Property ClientName As String
      Get
         Return name
      End Get
   End Property
End Class
```

The following C# example is able to instantiate the `case` class by using the @ symbol to disambiguate the identifier from the language keyword. Without it, the C# compiler would display two error messages, "Type expected" and "Invalid expression term 'case'."

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

Type conversion

The Common Language Specification defines two conversion operators:

- `op_Implicit`, which is used for widening conversions that do not result in loss of data or precision. For example, the Decimal structure includes an overloaded `op_Implicit` operator to convert values of integral types and Char values to `Decimal` values.
- `op_Explicit`, which is used for narrowing conversions that can result in loss of magnitude (a value is converted to a value that has a smaller range) or precision. For example, the `Decimal` structure includes an overloaded `op_Explicit` operator to convert Double and Single values to `Decimal` and to convert `Decimal` values to integral values, `Double`, `Single`, and `Char`.

However, not all languages support operator overloading or the definition of custom operators. If you choose to implement these conversion operators, you should also provide an alternate way to perform the conversion. We recommend that you provide `From`Xxx and `To`Xxx methods.

The following example defines CLS-compliant implicit and explicit conversions. It creates a `UDouble` class that represents an signed double-precision, floating-point number. It provides for implicit conversions from `UDouble` to `Double` and for explicit conversions from `UDouble` to `Single`, `Double` to `UDouble`, and `Single` to `UDouble`. It also defines a `ToDouble` method as an alternative to the implicit conversion operator and the `ToSingle`, `FromDouble`, and `FromSingle` methods as alternatives to the explicit conversion operators.

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public static readonly UDouble MinValue = (UDouble) 0.0;
   public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

   public static explicit operator Double(UDouble value)
   {
      return value.number;
```

```
    }

    public static implicit operator Single(UDouble value)
    {
       if (value.number > (double) Single.MaxValue)
          throw new InvalidCastException("A UDouble value is out of range of the Single type.");

       return (float) value.number;
    }

    public static explicit operator UDouble(double value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static implicit operator UDouble(float value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static Double ToDouble(UDouble value)
    {
       return (Double) value;
    }

    public static float ToSingle(UDouble value)
    {
       return (float) value;
    }

    public static UDouble FromDouble(double value)
    {
       return new UDouble(value);
    }

    public static UDouble FromSingle(float value)
    {
       return new UDouble(value);
    }
 }
```

```
Public Structure UDouble
   Private number As Double

   Public Sub New(value As Double)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Sub New(value As Single)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
   Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

   Public Shared Widening Operator CType(value As UDouble) As Double
      Return value.number
   End Operator

   Public Shared Narrowing Operator CType(value As UDouble) As Single
      If value.number > CDbl(Single.MaxValue) Then
         Throw New InvalidCastException("A UDouble value is out of range of the Single type.")
      End If
      Return CSng(value.number)
   End Operator

   Public Shared Narrowing Operator CType(value As Double) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Narrowing Operator CType(value As Single) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Function ToDouble(value As UDouble) As Double
      Return CType(value, Double)
   End Function

   Public Shared Function ToSingle(value As UDouble) As Single
      Return CType(value, Single)
   End Function

   Public Shared Function FromDouble(value As Double) As UDouble
      Return New UDouble(value)
   End Function

   Public Shared Function FromSingle(value As Single) As UDouble
      Return New UDouble(value)
   End Function
End Structure
```

Arrays

CLS-compliant arrays conform to the following rules:

- All dimensions of an array must have a lower bound of zero. The following example creates a non-CLS-

compliant array with a lower bound of one. Note that, despite the presence of the CLSCompliantAttribute attribute, the compiler does not detect that the array returned by the `Numbers.GetTenPrimes` method is not CLS-compliant.

```
[assembly: CLSCompliant(true)]

public class Numbers
{
  public static Array GetTenPrimes()
  {
      Array arr = Array.CreateInstance(typeof(Int32), new int[] {10}, new int[] {1});
      arr.SetValue(1, 1);
      arr.SetValue(2, 2);
      arr.SetValue(3, 3);
      arr.SetValue(5, 4);
      arr.SetValue(7, 5);
      arr.SetValue(11, 6);
      arr.SetValue(13, 7);
      arr.SetValue(17, 8);
      arr.SetValue(19, 9);
      arr.SetValue(23, 10);

      return arr;
  }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As Array
      Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
      arr.SetValue(1, 1)
      arr.SetValue(2, 2)
      arr.SetValue(3, 3)
      arr.SetValue(5, 4)
      arr.SetValue(7, 5)
      arr.SetValue(11, 6)
      arr.SetValue(13, 7)
      arr.SetValue(17, 8)
      arr.SetValue(19, 9)
      arr.SetValue(23, 10)
      Return arr
   End Function
End Class
```

- All array elements must consist of CLS-compliant types. The following example defines two methods that return non-CLS-compliant arrays. The first returns an array of UInt32 values. The second returns an Object array that includes Int32 and `UInt32` values. Although the compiler identifies the first array as non-compliant because of its `UInt32` type, it fails to recognize that the second array includes non-CLS-compliant elements.

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
  public static UInt32[] GetTenPrimes()
  {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
  }

  public static Object[] GetFivePrimes()
  {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
  }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of 'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As UInt32()
      Return { 1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui }
   End Function
   Public Shared Function GetFivePrimes() As Object()
      Dim arr() As Object = { 1, 2, 3, 5ui, 7ui }
      Return arr
   End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not CLS-compliant.
```

- Overload resolution for methods that have array parameters is based on the fact that they are arrays and on their element type. For this reason, the following definition of an overloaded `GetSquares` method is CLS-compliant.

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
  public static byte[] GetSquares(byte[] numbers)
  {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
          int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
          if (square <= Byte.MaxValue)
              numbersOut[ctr] = (byte) square;
          // If there's an overflow, assign MaxValue to the corresponding
          // element.
          else
              numbersOut[ctr] = Byte.MaxValue;

      }
      return numbersOut;
  }

  public static BigInteger[] GetSquares(BigInteger[] numbers)
{
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
          numbersOut[ctr] = numbers[ctr] * numbers[ctr];

     return numbersOut;
  }
}
```

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
   Public Function GetSquares(numbers As Byte()) As Byte()
      Dim numbersOut(numbers.Length - 1) As Byte
      For ctr As Integer = 0 To numbers.Length - 1
         Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
         If square <= Byte.MaxValue Then
            numbersOut(ctr) = CByte(square)
         ' If there's an overflow, assign MaxValue to the corresponding
         ' element.
         Else
            numbersOut(ctr) = Byte.MaxValue
         End If
      Next
      Return numbersOut
   End Function

   Public Function GetSquares(numbers As BigInteger()) As BigInteger()
       Dim numbersOut(numbers.Length - 1) As BigInteger
       For ctr As Integer = 0 To numbers.Length - 1
          numbersOut(ctr) = numbers(ctr) * numbers(ctr)
       Next
       Return numbersOut
   End Function
End Module
```

Interfaces

CLS-compliant interfaces can define properties, events, and virtual methods (methods with no implementation). A

CLS-compliant interface cannot have any of the following:

- Static methods or static fields. The C# compiler generates compiler errors if you define a static member in an interface.
- Fields. The C# a compiler generates compiler errors if you define a field in an interface.
- Methods that are not CLS-compliant. For example, the following interface definition includes a method, `INumber.GetUnsigned`, that is marked as non-CLS-compliant. This example generates a compiler warning.

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
    int Length();
    [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()': CLS-compliant interfaces
//            must have only CLS-compliant members
```

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
  Function Length As Integer
    <CLSCompliant(False)> Function GetUnsigned As ULong
  End Interface
  ' Attempting to compile the example displays output like the following:
  '    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not allowed in a
  '    CLS-compliant interface.
  '
  '       <CLSCompliant(False)> Function GetUnsigned As ULong
  '                                      ~~~~~~~~~~~
```

  Because of this rule, CLS-compliant types are not required to implement non-CLS-compliant members. If a CLS-compliant framework does expose a class that implements a non-CLS compliant interface, it should also provide concrete implementations of all non-CLS-compliant members.

CLS-compliant language compilers must also allow a class to provide separate implementations of members that have the same name and signature in multiple interfaces. C# supports explicit interface implementations to provide different implementations of identically named methods. The following example illustrates this scenario by defining a `Temperature` class that implements the `ICelsius` and `IFahrenheit` interfaces as explicit interface implementations.

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        cTemp.GetTemperature());
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        fTemp.GetTemperature());
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

```
Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
   Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
   Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
   Private _value As Decimal

   Public Sub New(value As Decimal)
      ' We assume that this is the Celsius value.
      _value = value
   End Sub

   Public Function GetFahrenheit() As Decimal _
          Implements IFahrenheit.GetTemperature
      Return _value * 9 / 5 + 32
   End Function

   Public Function GetCelsius() As Decimal _
          Implements ICelsius.GetTemperature
      Return _value
   End Function
End Class

Module Example
   Public Sub Main()
      Dim temp As New Temperature(100.0d)
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        temp.GetCelsius())
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        temp.GetFahrenheit())
   End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

Enumerations

CLS-compliant enumerations must follow these rules:

- The underlying type of the enumeration must be an intrinsic CLS-compliant integer (Byte, Int16, Int32, or Int64). For example, the following code tries to define an enumeration whose underlying type is UInt32 and generates a compiler warning.

```
using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
    Unspecified = 0,
    XSmall = 1,
    Small = 2,
    Medium = 3,
    Large = 4,
    XLarge = 5
};

public class Clothing
{
    public string Name;
    public string Type;
    public string Size;
}
// The attempt to compile the example displays a compiler warning like the following:
//    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
   Unspecified = 0
   XSmall = 1
   Small = 2
   Medium = 3
   Large = 4
   XLarge = 5
End Enum

Public Class Clothing
   Public Name As String
   Public Type As String
   Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- An enumeration type must have a single instance field named `Value__` that is marked with the `FieldAttributes.RTSpecialName` attribute. This enables you to reference the field value implicitly.
- An enumeration includes literal static fields whose types match the type of the enumeration itself. For example, if you define a `State` enumeration with values of `State.On` and `State.Off`, `State.On` and `State.Off` are both literal static fields whose type is `State`.
- There are two kinds of enumerations:
  - An enumeration that represents a set of mutually exclusive, named integer values. This type of enumeration is indicated by the absence of the System.FlagsAttribute custom attribute.
  - An enumeration that represents a set of bit flags that can combine to generate an unnamed value. This type of enumeration is indicated by the presence of the System.FlagsAttribute custom attribute.

For more information, see the documentation for the Enum structure.

- The value of an enumeration is not limited to the range of its specified values. In other words, the range of

values in an enumeration is the range of its underlying value. You can use the `Enum.IsDefined` method to determine whether a specified value is a member of an enumeration.

Type members in general

The Common Language Specification requires all fields and methods to be accessed as members of a particular class. Therefore, global static fields and methods (that is, static fields or methods that are defined apart from a type) are not CLS-compliant. If you try to include a global field or method in your source code, the C# compiler generates a compiler error.

The Common Language Specification supports only the standard managed calling convention. It doesn't support unmanaged calling conventions and methods with variable argument lists marked with the `varargs` keyword. For variable argument lists that are compatible with the standard managed calling convention, use the ParamArrayAttribute attribute or the individual language's implementation, such as the `params` keyword in C# and the `ParamArray` keyword in Visual Basic.

Member accessibility

Overriding an inherited member cannot change the accessibility of that member. For example, a public method in a base class cannot be overridden by a private method in a derived class. There is one exception: a `protected internal` (in C#) or `Protected Friend` (in Visual Basic) member in one assembly that is overridden by a type in a different assembly. In that case, the accessibility of the override is `Protected`.

The following example illustrates the error that is generated when the CLSCompliantAttribute attribute is set to `true`, and `Person`, which is a class derived from `Animal`, tries to change the accessibility of the `Species` property from public to private. The example compiles successfully if its accessibility is changed to public.

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be private
```

```
<Assembly: CLSCompliant(True)>

Public Class Animal
   Private _species As String

   Public Sub New(species As String)
      _species = species
   End Sub

   Public Overridable ReadOnly Property Species As String
      Get
         Return _species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _species
   End Function
End Class

Public Class Human : Inherits Animal
   Private _name As String

   Public Sub New(name As String)
      MyBase.New("Homo Sapiens")
      _name = name
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return _name
      End Get
   End Property

   Private Overrides ReadOnly Property Species As String
      Get
         Return MyBase.Species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _name
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim p As New Human("John")
      Console.WriteLine(p.Species)
      Console.WriteLine(p.ToString())
   End Sub
End Module
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

Types in the signature of a member must be accessible whenever that member is accessible. For example, this means that a public member cannot include a parameter whose type is private, protected, or internal. The following example illustrates the compiler error that results when a `StringWrapper` class constructor exposes an internal `StringOperationType` enumeration value that determines how a string value should be wrapped.

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//            'StringOperationType' is less accessible than method
//            'StringWrapper.StringWrapper(StringOperationType)'
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class StringWrapper

   Dim internalString As String
   Dim internalSB As StringBuilder = Nothing
   Dim useSB As Boolean = False

   Public Sub New(type As StringOperationType)
      If type = StringOperationType.Normal Then
         useSB = False
      Else
         internalSB = New StringBuilder()
         useSB = True
      End If
   End Sub

   ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
   Normal = 0
   Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                              ~~~~~~~~~~~~~~~~~~~~
```

Generic types and members

Nested types always have at least as many generic parameters as their enclosing type. These correspond by

position to the generic parameters in the enclosing type. The generic type can also include new generic parameters.

The relationship between the generic type parameters of a containing type and its nested types may be hidden by the syntax of individual languages. In the following example, a generic type `Outer<T>` contains two nested classes, `Inner1A` and `Inner1B<U>`. The calls to the `ToString` method, which each class inherits from `Object.ToString`, show that each nested class includes the type parameters of its containing class.

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);

      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

```
<Assembly:CLSCompliant(True)>

Public Class Outer(Of T)
   Dim value As T

   Public Sub New(value As T)
      Me.value = value
   End Sub

   Public Class Inner1A : Inherits Outer(Of T)
      Public Sub New(value As T)
         MyBase.New(value)
      End Sub
   End Class

   Public Class Inner1B(Of U) : Inherits Outer(Of T)
      Dim value2 As U

      Public Sub New(value1 As T, value2 As U)
         MyBase.New(value1)
         Me.value2 = value2
      End Sub
   End Class
End Class

Public Module Example
   Public Sub Main()
      Dim inst1 As New Outer(Of String)("This")
      Console.WriteLine(inst1)

      Dim inst2 As New Outer(Of String).Inner1A("Another")
      Console.WriteLine(inst2)

      Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
      Console.WriteLine(inst3)
   End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

Generic type names are encoded in the form *name*'*n*, where *name* is the type name, ` is a character literal, and *n* is the number of parameters declared on the type, or, for nested generic types, the number of newly introduced type parameters. This encoding of generic type names is primarily of interest to developers who use reflection to access CLS-complaint generic types in a library.

If constraints are applied to a generic type, any types used as constraints must also be CLS-compliant. The following example defines a class named `BaseClass` that is not CLS-compliant and a generic class named `BaseCollection` whose type parameter must derive from `BaseClass`. But because `BaseClass` is not CLS-compliant, the compiler emits a warning.

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}

public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

```
Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class

Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~~
```

If a generic type is derived from a generic base type, it must redeclare any constraints so that it can guarantee that constraints on the base type are also satisfied. The following example defines a `Number<T>` that can represent any numeric type. It also defines a `FloatingPoint<T>` class that represents a floating point value. However, the source code fails to compile, because it does not apply the constraint on `Number<T>` (that T must be a value type) to `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
// The attempt to compile the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//               order to use it as parameter 'T' in the generic type or method 'Number<T>'
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
' The attempt to compile the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                    ~
```

The example compiles successfully if the constraint is added to the `FloatingPoint<T>` class.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
```

The Common Language Specification imposes a conservative per-instantiation model for nested types and protected members. Open generic types cannot expose fields or members with signatures that contain a specific instantiation of a nested, protected generic type. Non-generic types that extend a specific instantiation of a generic base class or interface cannot expose fields or members with signatures that contain a different instantiation of a nested, protected generic type.

The following example defines a generic type, `C1<T>`, and a protected class, `C1<T>.N`. `C1<T>` has two methods, `M1` and `M2`. However, `M1` is not CLS-compliant because it tries to return a `C1<int>.N` object from `C1<T>`. A second class, `C2`, is derived from `C1<long>`. It has two methods, `M3` and `M4`. `M3` is not CLS-compliant because it tries to return a `C1<int>.N` object from a subclass of `C1<long>`. Note that language compilers can be even more restrictive. In this example, Visual Basic displays an error when it tries to compile `M4`.

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                     // accessible from within C1<T> in all
                                     // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                     // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is not
                                       // accessible in C2 (extends C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
```

```
<Assembly:CLSCompliant(True)>

Public Class C1(Of T)
   Protected Class N
   End Class

   Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
                                             ' accessible from within C1(Of T) in all
   End Sub                                   ' languages

   Protected Sub M2(n As C1(Of T).N)     ' CLS-compliant – C1(Of T).N accessible
   End Sub                               ' inside C1(Of T)
End Class

Public Class C2 : Inherits C1(Of Long)
   Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant – C1(Of Integer).N is not
   End Sub                                   ' accessible in C2 (extends C1(Of Long))

   Protected Sub M4(n As C1(Of Long).N)
   End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
'                                 ~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant - C1(Of Integer).N is not
'
'                                 ~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is 'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                                 ~~~~~~~~~~~~~
```

Constructors

Constructors in CLS-compliant classes and structures must follow these rules:

- A constructor of a derived class must call the instance constructor of its base class before it accesses inherited instance data. This requirement is due to the fact that base class constructors are not inherited by their derived classes. This rule does not apply to structures, which do not support direct inheritance.

  Typically, compilers enforce this rule independently of CLS compliance, as the following example shows. It creates a `Doctor` class that is derived from a `Person` class, but the `Doctor` class fails to call the `Person` class constructor to initialize inherited instance fields.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
private string fName, lName, _id;

public Person(string firstName, string lastName, string id)
{
    if (String.IsNullOrEmpty(firstName + lastName))
        throw new ArgumentNullException("Either a first name or a last name must be provided.");

    fName = firstName;
    lName = lastName;
    _id = id;
}

public string FirstName
{
    get { return fName; }
}

public string LastName
{
    get { return lName; }
}

public string Id
{
    get { return _id; }
}

public override string ToString()
{
    return String.Format("{0}{1}{2}", fName,
                         String.IsNullOrEmpty(fName) ?  "" : " ",
                         lName);
}
}

public class Doctor : Person
{
public Doctor(string firstName, string lastName, string id)
{
}

public override string ToString()
{
    return "Dr. " + base.ToString();
}
}
// Attempting to compile the example displays output like the following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a constructor that takes 0
//            arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private fName, lName, _id As String

   Public Sub New(firstName As String, lastName As String, id As String)
      If String.IsNullOrEmpty(firstName + lastName) Then
         Throw New ArgumentNullException("Either a first name or a last name must be provided.")
      End If

      fName = firstName
      lName = lastName
      _id = id
   End Sub

   Public ReadOnly Property FirstName As String
      Get
         Return fName
      End Get
   End Property

   Public ReadOnly Property LastName As String
      Get
         Return lName
      End Get
   End Property

   Public ReadOnly Property Id As String
      Get
         Return _id
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return String.Format("{0}{1}{2}", fName,
                           If(String.IsNullOrEmpty(fName), "", " "),
                           lName)
   End Function
End Class

Public Class Doctor : Inherits Person
   Public Sub New(firstName As String, lastName As String, id As String)
   End Sub

   Public Overrides Function ToString() As String
      Return "Dr. " + MyBase.ToString()
   End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor' does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- An object constructor cannot be called except to create an object. In addition, an object cannot be initialized twice. For example, this means that `Object.MemberwiseClone` must not call constructors.

Properties

Properties in CLS-compliant types must follow these rules:

- A property must have a setter, a getter, or both. In an assembly, these are implemented as special methods, which means that they will appear as separate methods (the getter is named `get` *_propertyname* and the setter is `set` *_propertyname*) marked as `SpecialName` in the assembly's metadata. The C# compiler enforces

this rule automatically without the need to apply the CLSCompliantAttribute attribute.

- A property's type is the return type of the property getter and the last argument of the setter. These types must be CLS compliant, and arguments cannot be assigned to the property by reference (that is, they cannot be managed pointers).
- If a property has both a getter and a setter, they must both be virtual, both static, or both instance. The C# compiler automatically enforces this rule through property definition syntax.

Events

An event is defined by its name and its type. The event type is a delegate that is used to indicate the event. For example, the `DbConnection.StateChange` event is of type `StateChangeEventHandler`. In addition to the event itself, three methods with names based on the event name provide the event's implementation and are marked as `SpecialName` in the assembly's metadata:

- A method for adding an event handler, named `add` _*EventName*. For example, the event subscription method for the `DbConnection.StateChange` event is named `add_StateChange`.
- A method for removing an event handler, named `remove` _*EventName*. For example, the removal method for the `DbConnection.StateChange` event is named `remove_StateChange`.
- A method for indicating that the event has occurred, named `raise` _*EventName*.

> **NOTE**
> Most of the Common Language Specification's rules regarding events are implemented by language compilers and are transparent to component developers.

The methods for adding, removing, and raising the event must have the same accessibility. They must also all be static, instance, or virtual. The methods for adding and removing an event have one parameter whose type is the event delegate type. The add and remove methods must both be present or both be absent.

The following example defines a CLS-compliant class named `Temperature` that raises a `TemperatureChanged` event if the change in temperature between two readings equals or exceeds a threshold value. The `Temperature` class explicitly defines a `raise_TemperatureChanged` method so that it can selectively execute event handlers.

```
using System;
using System.Collections;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new, DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
      when = time;
   }

   public Decimal OldTemperature
   {
      get { return originalTemp; }
   }
```

```
   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender, TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not executed.");
         else
            d.Invoke(this, eventArgs);
      }
   }
}

public class Example
{
   public Temperature temp;
```

```
   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
   {
      temp = t;
      RecordTemperatures();
   }

   public void RecordTemperatures()
   {
      temp.TemperatureChanged += this.DuplicateTemperatureNotification;
      temp.CurrentTemperature = 66;
      temp.CurrentTemperature = 63;
   }

   internal void TemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }

   public void DuplicateTemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }
}
```

```
Imports System.Collections
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs   : Inherits EventArgs
   Private originalTemp As Decimal
   Private newTemp As Decimal
   Private [when] As DateTimeOffset

   Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
      originalTemp = original
      newTemp = [new]
      [when] = [time]
   End Sub

   Public ReadOnly Property OldTemperature As Decimal
      Get
         Return originalTemp
      End Get
   End Property

   Public ReadOnly Property CurrentTemperature As Decimal
      Get
         Return newTemp
      End Get
```

```
   End Property

   Public ReadOnly Property [Time] As DateTimeOffset
      Get
         Return [when]
      End Get
   End Property
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As TemperatureChangedEventArgs)

Public Class Temperature
   Private Structure TemperatureInfo
      Dim Temperature As Decimal
      Dim Recorded As DateTimeOffset
   End Structure

   Public Event TemperatureChanged As TemperatureChanged

   Private previous As Decimal
   Private current As Decimal
   Private tolerance As Decimal
   Private tis As New List(Of TemperatureInfo)

   Public Sub New(temperature As Decimal, tolerance As Decimal)
      current = temperature
      Dim ti As New TemperatureInfo()
      ti.Temperature = temperature
      ti.Recorded = DateTimeOffset.UtcNow
      tis.Add(ti)
      Me.tolerance = tolerance
   End Sub

   Public Property CurrentTemperature As Decimal
      Get
         Return current
      End Get
      Set
         Dim ti As New TemperatureInfo
         ti.Temperature = value
         ti.Recorded = DateTimeOffset.UtcNow
         previous = current
         current = value
         If Math.Abs(current - previous) >= tolerance Then
            raise_TemperatureChanged(New TemperatureChangedEventArgs(previous, current, ti.Recorded))
         End If
      End Set
   End Property

   Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
      If TemperatureChangedEvent Is Nothing Then Exit Sub

      Dim ListenerList() As System.Delegate = TemperatureChangedEvent.GetInvocationList()
      For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
         If d.Method.Name.Contains("Duplicate") Then
            Console.WriteLine("Duplicate event handler; event handler not executed.")
         Else
            d.Invoke(Me, eventArgs)
         End If
      Next
   End Sub
End Class

Public Class Example
   Public WithEvents temp As Temperature

   Public Shared Sub Main()
      Dim ex As New Example()
   End Sub
```

```
    Public Sub New()
       temp = New Temperature(65, 3)
       RecordTemperatures()
       Dim ex As New Example(temp)
       ex.RecordTemperatures()
    End Sub

    Public Sub New(t As Temperature)
       temp = t
       RecordTemperatures()
    End Sub

    Public Sub RecordTemperatures()
       temp.CurrentTemperature = 66
       temp.CurrentTemperature = 63

    End Sub

    Friend Shared Sub TemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
       Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub

    Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
       Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub
End Class
```

Overloads

The Common Language Specification imposes the following requirements on overloaded members:

- Members can be overloaded based on the number of parameters and the type of any parameter. Calling convention, return type, custom modifiers applied to the method or its parameter, and whether parameters are passed by value or by reference are not considered when differentiating between overloads. For an example, see the code for the requirement that names must be unique within a scope in the Naming conventions section.
- Only properties and methods can be overloaded. Fields and events cannot be overloaded.
- Generic methods can be overloaded based on the number of their generic parameters.

> **NOTE**
>
> The `op_Explicit` and `op_Implicit` operators are exceptions to the rule that return value is not considered part of a method signature for overload resolution. These two operators can be overloaded based on both their parameters and their return value.

Exceptions

Exception objects must derive from System.Exception or from another type derived from `System.Exception`. The following example illustrates the compiler error that results when a custom class named `ErrorClass` is used for exception handling.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be derived from
//            System.Exception
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from 'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

To correct this error, the `ErrorClass` class must inherit from `System.Exception`. In addition, the Message property must be overridden. The following example corrects these errors to define an `ErrorClass` class that is CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public Overrides ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
```

Attributes

In.NET Framework assemblies, custom attributes provide an extensible mechanism for storing custom attributes and retrieving metadata about programming objects, such as assemblies, types, members, and method parameters. Custom attributes must derive from System.Attribute or from a type derived from `System.Attribute`.

The following example violates this rule. It defines a `NumericAttribute` class that does not derive from `System.Attribute`. Note that a compiler error results only when the non-CLS-compliant attribute is applied, not when the class is defined.

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
   Private _isNumeric As Boolean

   Public Sub New(isNumeric As Boolean)
      _isNumeric = isNumeric
   End Sub

   Public ReadOnly Property IsNumeric As Boolean
      Get
         Return _isNumeric
      End Get
   End Property
End Class

<Numeric(True)> Public Structure UDouble
   Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~
```

The constructor or the properties of a CLS-compliant attribute can expose only the following types:

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- Any enumeration type whose underlying type is `Byte`, `Int16`, `Int32`, or `Int64`.

The following example defines a `DescriptionAttribute` class that derives from Attribute. The class constructor has a parameter of type `Descriptor`, so the class is not CLS-compliant. Note that the C# compiler emits a warning but compiles successfully.

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//               constructors which use only CLS-compliant types
```

```
<Assembly:CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
   Type = 0
   Member = 1
End Enum

Public Class Descriptor
   Public Type As DescriptorType
   Public Description As String
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
   Private desc As Descriptor

   Public Sub New(d As Descriptor)
      desc = d
   End Sub

   Public ReadOnly Property Descriptor As Descriptor
      Get
         Return desc
      End Get
   End Property
End Class
```

The CLSCompliantAttribute attribute

The CLSCompliantAttribute attribute is used to indicate whether a program element complies with the Common Language Specification. The `CLSCompliantAttribute.CLSCompliantAttribute(Boolean)` constructor includes a single required parameter, *isCompliant*, that indicates whether the program element is CLS-compliant.

At compile time, the compiler detects non-compliant elements that are presumed to be CLS-compliant and emits a warning. The compiler does not emit warnings for types or members that are explicitly declared to be non-compliant.

Component developers can use the `CLSCompliantAttribute` attribute in two ways:

- To define the parts of the public interface exposed by a component that are CLS-compliant and the parts that are not CLS-compliant. When the attribute is used to mark particular program elements as CLS-compliant, its use guarantees that those elements are accessible from all languages and tools that target the .NET Framework.
- To ensure that the component library's public interface exposes only program elements that are CLS-compliant. If elements are not CLS-compliant, compilers will generally issue a warning.

> **WARNING**
> In some cases, language compilers enforce CLS-compliant rules regardless of whether the `CLSCompliantAttribute` attribute is used. For example, defining a `*static` member in an interface violates a CLS rule. However, if you define a `*static` member in an interface, the C# compiler displays an error message and fails to compile the app.

The `CLSCompliantAttribute` attribute is marked with an AttributeUsageAttribute attribute that has a value of `AttributeTargets.All`. This value allows you to apply the `CLSCompliantAttribute` attribute to any program element, including assemblies, modules, types (classes, structures, enumerations, interfaces, and delegates), type members (constructors, methods, properties, fields, and events), parameters, generic parameters, and return values. However, in practice, you should apply the attribute only to assemblies, types, and type members. Otherwise, compilers

ignore the attribute and continue to generate compiler warnings whenever they encounter a non-compliant parameter, generic parameter, or return value in your library's public interface.

The value of the `CLSCompliantAttribute` attribute is inherited by contained program elements. For example, if an assembly is marked as CLS-compliant, its types are also CLS-compliant. If a type is marked as CLS-compliant, its nested types and members are also CLS-compliant.

You can explicitly override the inherited compliance by applying the `CLSCompliantAttribute` attribute to a contained program element. For example, you can use the `CLSCompliantAttribute` attribute with an *isCompliant* value of `false` to define a non-compliant type in a compliant assembly, and you can use the attribute with an *isCompliant* value of `true` to define a compliant type in a non-compliant assembly. You can also define non-compliant members in a compliant type. However, a non-compliant type cannot have compliant members, so you cannot use the attribute with an *isCompliant* value of `true` to override inheritance from a non-compliant type.

When you are developing components, you should always use the `CLSCompliantAttribute` attribute to indicate whether your assembly, its types, and its members are CLS-compliant.

To create CLS-compliant components:

1. Use the `CLSCompliantAttribute` to mark you assembly as CLS-compliant.
2. Mark any publicly exposed types in the assembly that are not CLS-compliant as non-compliant.
3. Mark any publicly exposed members in CLS-compliant types as non-compliant.
4. Provide a CLS-compliant alternative for non-CLS-compliant members.

If you've successfully marked all your non-compliant types and members, your compiler should not emit any non-compliance warnings. However, you should indicate which members are not CLS-compliant and list their CLS-compliant alternatives in your product documentation.

The following example uses the `CLSCompliantAttribute` attribute to define a CLS-compliant assembly and a type, `CharacterUtilities`, that has two non-CLS-compliant members. Because both members are tagged with the `CLSCompliant(false)` attribute, the compiler produces no warnings. The class also provides a CLS-compliant alternative for both methods. Ordinarily, we would just add two overloads to the `ToUTF16` method to provide CLS-compliant alternatives. However, because methods cannot be overloaded based on return value, the names of the CLS-compliant methods are different from the names of the non-compliant methods.

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class CharacterUtilities
   <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
      s = s.Normalize(NormalizationForm.FormC)
      Return Convert.ToUInt16(s(0))
   End Function

   <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
      Return Convert.ToUInt16(ch)
   End Function

   ' CLS-compliant alternative for ToUTF16(String).
   Public Shared Function ToUTF16CodeUnit(s As String) As Integer
      s = s.Normalize(NormalizationForm.FormC)
      Return CInt(Convert.ToInt16(s(0)))
   End Function

   ' CLS-compliant alternative for ToUTF16(Char).
   Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
      Return Convert.ToInt32(ch)
   End Function

   Public Function HasMultipleRepresentations(s As String) As Boolean
      Dim s1 As String = s.Normalize(NormalizationForm.FormC)
      Return s.Equals(s1)
   End Function

   Public Function GetUnicodeCodePoint(ch As Char) As Integer
      If Char.IsSurrogate(ch) Then
         Throw New ArgumentException("ch cannot be a high or low surrogate.")
      End If
      Return Char.ConvertToUtf32(ch.ToString(), 0)
   End Function

   Public Function GetUnicodeCodePoint(chars() As Char) As Integer
      If chars.Length > 2 Then
         Throw New ArgumentException("The array has too many characters.")
      End If
      If chars.Length = 2 Then
         If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
            Throw New ArgumentException("The array must contain a low and a high surrogate.")
         Else
            Return Char.ConvertToUtf32(chars(0), chars(1))
         End If
      Else
         Return Char.ConvertToUtf32(chars.ToString(), 0)
      End If
   End Function
End Class
```

If you are developing an app rather than a library (that is, if you aren't exposing types or members that can be consumed by other app developers), the CLS compliance of the program elements that your app consumes are of interest only if your language does not support them. In that case, your language compiler will generate an error when you try to use a non-CLS-compliant element.

Cross-Language Interoperability

Language independence has a number of possible meanings. One meaning involves seamlessly consuming types written in one language from an app written in another language. A second meaning, which is the focus of this article, involves combining code written in multiple languages into a single .NET Framework assembly.

The following example illustrates cross-language interoperability by creating a class library named Utilities.dll that includes two classes, `NumericLib` and `StringLib`. The `NumericLib` class is written in C#, and the `StringLib` class is written in Visual Basic. Here's the source code for `StringUtil.vb`, which includes a single member, `ToTitleCase`, in its `StringLib` class.

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
   Private exclusions As List(Of String)

   Sub New()
      Dim words() As String = { "a", "an", "and", "of", "the" }
      exclusions = New List(Of String)
      exclusions.AddRange(words)
   End Sub

   <Extension()> _
   Public Function ToTitleCase(title As String) As String
      Dim words() As String = title.Split()
      Dim result As String = String.Empty

      For ctr As Integer = 0 To words.Length - 1
         Dim word As String = words(ctr)
         If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
            result += word.Substring(0, 1).ToUpper() + _
                      word.Substring(1).ToLower()
         Else
            result += word.ToLower()
         End If
         If ctr <= words.Length - 1 Then
            result += " "
         End If
      Next
      Return result
   End Function
End Module
```

Here's the source code for NumberUtil.cs, which defines a `NumericLib` class that has two members, `IsEven` and `NearZero`.

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return ((long) number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) %2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer value.");
   }

   public static bool NearZero(double number)
   {
      return number < .00001;
   }
}
```

To package the two classes in a single assembly, you must compile them into modules. To compile the Visual Basic source code file into a module, use this command:

```
vbc /t:module StringUtil.vb
```

To compile the C# source code file into a module, use this command:

```
csc /t:module NumberUtil.cs
```

You then use the Link tool (Link.exe) to compile the two modules into an assembly:

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

The following example then calls the `NumericLib.NearZero` and `StringLib.ToTitleCase` methods. Note that both the Visual Basic code and the C# code are able to access the methods in both classes.

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

```
Module Example
   Public Sub Main()
      Dim dbl As Double = 0.0 - Double.Epsilon
      Console.WriteLine(NumericLib.NearZero(dbl))

      Dim s As String = "war and peace"
      Console.WriteLine(s.ToTitleCase())
   End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

To compile the Visual Basic code, use this command:

```
vbc example.vb /r:UtilityLib.dll
```

To compile with C#, change the name of the compiler from vbc to csc, and change the file extension from .vb to .cs:

```
csc example.cs /r:UtilityLib.dll
```

Language Independence and Language-Independent Components

7/9/2019 • 68 minutes to read • Edit Online

The .NET Framework is language independent. This means that, as a developer, you can develop in one of the many languages that target the .NET Framework, such as C#, C++/CLI, Eiffel, F#, IronPython, IronRuby, PowerBuilder, Visual Basic, Visual COBOL, and Windows PowerShell. You can access the types and members of class libraries developed for the .NET Framework without having to know the language in which they were originally written and without having to follow any of the original language's conventions. If you are a component developer, your component can be accessed by any .NET Framework app regardless of its language.

> **NOTE**
> This first part of this article discusses creating language-independent components—that is, components that can be consumed by apps that are written in any language. You can also create a single component or app from source code written in multiple languages; see Cross-Language Interoperability in the second part of this article.

To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the Common Language Specification (CLS), which is a set of rules that apply to generated assemblies. The Common Language Specification is defined in Partition I, Clauses 7 through 11 of the ECMA-335 Standard: Common Language Infrastructure.

If your component conforms to the Common Language Specification, it is guaranteed to be CLS-compliant and can be accessed from code in assemblies written in any programming language that supports the CLS. You can determine whether your component conforms to the Common Language Specification at compile time by applying the CLSCompliantAttribute attribute to your source code. For more information, see The CLSCompliantAttribute attribute.

In this article:

- CLS compliance rules
  - Types and type member signatures
  - Naming conventions
  - Type conversion
  - Arrays
  - Interfaces
  - Enumerations
  - Type members in general
  - Member accessibility
  - Generic types and members
  - Constructors
  - Properties

    - Events
    - Overloads
    - Exceptions
    - Attributes
- The CLSCompliantAttribute attribute
- Cross-Language Interoperability

CLS compliance rules

This section discusses the rules for creating a CLS-compliant component. For a complete list of rules, see Partition I, Clause 11 of the ECMA-335 Standard: Common Language Infrastructure.

> **NOTE**
> The Common Language Specification discusses each rule for CLS compliance as it applies to consumers (developers who are programmatically accessing a component that is CLS-compliant), frameworks (developers who are using a language compiler to create CLS-compliant libraries), and extenders (developers who are creating a tool such as a language compiler or a code parser that creates CLS-compliant components). This article focuses on the rules as they apply to frameworks. Note, though, that some of the rules that apply to extenders may also apply to assemblies that are created using Reflection.Emit.

To design a component that is language independent, you only need to apply the rules for CLS compliance to your component's public interface. Your private implementation does not have to conform to the specification.

> **IMPORTANT**
> The rules for CLS compliance apply only to a component's public interface, not to its private implementation.

For example, unsigned integers other than Byte are not CLS-compliant. Because the `Person` class in the following example exposes an `Age` property of type UInt16, the following code displays a compiler warning.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As UInt16
      Get
         Return personAge
      End Get
   End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

You can make the `Person` class CLS-compliant by changing the type of `Age` property from UInt16 to Int16, which is a CLS-compliant, 16-bit signed integer. You do not have to change the type of the private `personAge` field.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
   { get { return personAge; } }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private personAge As UInt16

   Public ReadOnly Property Age As Int16
      Get
         Return CType(personAge, Int16)
      End Get
   End Property
End Class
```

A library's public interface consists of the following:

- Definitions of public classes.
- Definitions of the public members of public classes, and definitions of members accessible to derived classes (that is, protected members).
- Parameters and return types of public methods of public classes, and parameters and return types of methods accessible to derived classes.

The rules for CLS compliance are listed in the following table. The text of the rules is taken verbatim from the ECMA-335 Standard: Common Language Infrastructure, which is Copyright 2012 by Ecma International. More detailed information about these rules is found in the following sections.

CATEGORY	SEE	RULE	RULE NUMBER
Accessibility	Member accessibility	Accessibility shall not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with accessibility `family-or-assembly`. In this case, the override shall have accessibility `family`.	10
Accessibility	Member accessibility	The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly. The visibility and accessibility of types composing an instantiated generic type used in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, an instantiated generic type present in the signature of a member that is visible outside its assembly shall not have a generic argument whose type is visible only within the assembly.	12
Arrays	Arrays	Arrays shall have elements with a CLS-compliant type, and all dimensions of the array shall have lower bounds of zero. Only the fact that an item is an array and the element type of the array shall be required to distinguish between overloads. When overloading is based on two or more array types the element types shall be named types.	16
Attributes	Attributes	Attributes shall be of type System.Attribute, or a type inheriting from it.	41

CATEGORY	SEE	RULE	RULE NUMBER
Attributes	Attributes	The CLS only allows a subset of the encodings of custom attributes. The only types that shall appear in these encodings are (see Partition IV): System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double, and any enumeration type based on a CLS-compliant base integer type.	34
Attributes	Attributes	The CLS does not allow publicly visible required modifiers (`modreq` , see Partition II), but does allow optional modifiers (`modopt` , see Partition II) it does not understand.	35
Constructors	Constructors	An object constructor shall call some instance constructor of its base class before any access occurs to inherited instance data. (This does not apply to value types, which need not have constructors.)	21
Constructors	Constructors	An object constructor shall not be called except as part of the creation of an object, and an object shall not be initialized twice.	22
Enumerations	Enumerations	The underlying type of an enum shall be a built-in CLS integer type, the name of the field shall be "value__", and that field shall be marked `RTSpecialName` .	7
Enumerations	Enumerations	There are two distinct kinds of enums, indicated by the presence or absence of the System.FlagsAttribute (see Partition IV Library) custom attribute. One represents named integer values; the other represents named bit flags that can be combined to generate an unnamed value. The value of an `enum` is not limited to the specified values.	8

CATEGORY	SEE	RULE	RULE NUMBER
Enumerations	Enumerations	Literal static fields of an enum shall have the type of the enum itself.	9
Events	Events	The methods that implement an event shall be marked `SpecialName` in the metadata.	29
Events	Events	The accessibility of an event and of its accessors shall be identical.	30
Events	Events	The `add` and `remove` methods for an event shall both either be present or absent.	31
Events	Events	The `add` and `remove` methods for an event shall each take one parameter whose type defines the type of the event and that shall be derived from System.Delegate.	32
Events	Events	Events shall adhere to a specific naming pattern. The `SpecialName` attribute referred to in CLS rule 29 shall be ignored in appropriate name comparisons and shall adhere to identifier rules.	33
Exceptions	Exceptions	Objects that are thrown shall be of type System.Exception or a type inheriting from it. Nonetheless, CLS-compliant methods are not required to block the propagation of other types of exceptions.	40
General	CLS compliance: the Rules	CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly.	1
General	CLS compliance: the Rules	Members of non-CLS compliant types shall not be marked CLS-compliant.	2

CATEGORY	SEE	RULE	RULE NUMBER
Generics	Generic types and members	Nested types shall have at least as many generic parameters as the enclosing type. Generic parameters in a nested type correspond by position to the generic parameters in its enclosing type.	42
Generics	Generic types and members	The name of a generic type shall encode the number of type parameters declared on the non-nested type, or newly introduced to the type if nested, according to the rules defined above.	43
Generics	Generic types and members	A generic type shall redeclare sufficient constraints to guarantee that any constraints on the base type, or interfaces would be satisfied by the generic type constraints.	4444
Generics	Generic types and members	Types used as constraints on generic parameters shall themselves be CLS-compliant.	45
Generics	Generic types and members	The visibility and accessibility of members (including nested types) in an instantiated generic type shall be considered to be scoped to the specific instantiation rather than the generic type declaration as a whole. Assuming this, the visibility and accessibility rules of CLS rule 12 still apply.	46
Generics	Generic types and members	For each abstract or virtual generic method, there shall be a default concrete (non-abstract) implementation.	47
Interfaces	Interfaces	CLS-compliant interfaces shall not require the definition of non-CLS compliant methods in order to implement them.	18
Interfaces	Interfaces	CLS-compliant interfaces shall not define static methods, nor shall they define fields.	19

CATEGORY	SEE	RULE	RULE NUMBER
Members	Type members in general	Global static fields and methods are not CLS-compliant.	36
Members	--	The value of a literal static is specified through the use of field initialization metadata. A CLS-compliant literal must have a value specified in field initialization metadata that is of exactly the same type as the literal (or of the underlying type, if that literal is an `enum`).	13
Members	Type members in general	The vararg constraint is not part of the CLS, and the only calling convention supported by the CLS is the standard managed calling convention.	15
Naming conventions	Naming conventions	Assemblies shall follow Annex 7 of Technical Report 15 of the Unicode Standard3.0 governing the set of characters permitted to start and be included in identifiers, available online at https://www.unicode.org/unicode/reports/tr15/tr15-18.html. Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-to-one lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case. However, in order to override an inherited definition the CLI requires the precise encoding of the original declaration be used.	4

CATEGORY	SEE	RULE	RULE NUMBER
Overloading	Naming conventions	All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTSallows a single type to use the same name for a method and a field, the CLS does not.	5
Overloading	Naming conventions	Fields and nested types shall be distinct by identifier comparison alone, even though the CTS allows distinct signatures to be distinguished. Methods, properties, and events that have the same name (by identifier comparison) shall differ by more than just the return type,except as specified in CLS Rule 39.	6
Overloading	Overloads	Only properties and methods can be overloaded.	37
Overloading	Overloads	Properties and methods can be overloaded based only on the number and types of their parameters, except the conversion operators named `op_Implicit` and `op_Explicit`, which can also be overloaded based on their return type.	38
Overloading	--	If two or more CLS-compliant methods declared in a type have the same name and, for a specific set of type instantiations, they have the same parameter and return types, then all these methods shall be semantically equivalent at those type instantiations.	48
Types	Type and type member signatures	System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class.	23

CATEGORY	SEE	RULE	RULE NUMBER
Properties	Properties	The methods that implement the getter and setter methods of a property shall be marked `SpecialName` in the metadata.	24
Properties	Properties	A property's accessors shall all be static, all be virtual, or all be instance.	26
Properties	Properties	The type of a property shall be the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property shall be the types of the parameters to the getter and the types of all but the final parameter of the setter. All of these types shall be CLS-compliant, and shall not be managed pointers (i.e., shall not be passed by reference).	27
Properties	Properties	Properties shall adhere to a specific naming pattern. The `SpecialName` attribute referred to in CLS rule 24 shall be ignored in appropriate name comparisons and shall adhere to identifier rules. A property shall have a getter method, a setter method, or both.	28
Type conversion	Type conversion	If either `op_Implicit` or `op_Explicit` is provided, an alternate means of providing the coercion shall be provided.	39
Types	Type and type member signatures	Boxed value types are not CLS-compliant.	3
Types	Type and type member signatures	All types appearing in a signature shall be CLS-compliant. All types composing an instantiated generic type shall be CLS-compliant.	11
Types	Type and type member signatures	Typed references are not CLS-compliant.	14

CATEGORY	SEE	RULE	RULE NUMBER
Types	Type and type member signatures	Unmanaged pointer types are not CLS-compliant.	17
Types	Type and type member signatures	CLS-compliant classes, value types, and interfaces shall not require the implementation of non-CLS-compliant members.	20

Types and type member signatures

The System.Object type is CLS-compliant and is the base type of all object types in the .NET Framework type system. Inheritance in the .NET Framework is either implicit (for example, the String class implicitly inherits from the Object class) or explicit (for example, the CultureNotFoundException class explicitly inherits from the ArgumentException class, which explicitly inherits from the SystemException class, which explicitly inherits from the Exception class). For a derived type to be CLS compliant, its base type must also be CLS-compliant.

The following example shows a derived type whose base type is not CLS-compliant. It defines a base `Counter` class that uses an unsigned 32-bit integer as a counter. Because the class provides counter functionality by wrapping an unsigned integer, the class is marked as non-CLS-compliant. As a result, a derived class, `NonZeroCounter`, is also not CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return String.Format("{0}). ", ctr);
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
   {
   }

   private NonZeroCounter(UInt32 startIndex) : base(startIndex)
   {
   }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter' is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
   Dim ctr As UInt32

   Public Sub New
      ctr = 0
   End Sub

   Protected Sub New(ctr As UInt32)
      ctr = ctr
   End Sub

   Public Overrides Function ToString() As String
      Return String.Format("{0}). ", ctr)
   End Function

   Public ReadOnly Property Value As UInt32
      Get
         Return ctr
      End Get
   End Property

   Public Sub Increment()
      ctr += CType(1, UInt32)
   End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
   Public Sub New(startIndex As Integer)
      MyClass.New(CType(startIndex, UInt32))
   End Sub

   Private Sub New(startIndex As UInt32)
      MyBase.New(CType(startIndex, UInt32))
   End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

All types that appear in member signatures, including a method's return type or a property type, must be CLS-compliant. In addition, for generic types:

- All types that compose an instantiated generic type must be CLS-compliant.
- All types used as constraints on generic parameters must be CLS-compliant.

The .NET Framework common type system includes a number of built-in types that are supported directly by the common language runtime and are specially encoded in an assembly's metadata. Of these intrinsic types, the types listed in the following table are CLS-compliant.

CLS-COMPLIANT TYPE	DESCRIPTION
Byte	8-bit unsigned integer
Int16	16-bit signed integer

CLS-COMPLIANT TYPE	DESCRIPTION
Int32	32-bit signed integer
Int64	64-bit signed integer
Single	Single-precision floating-point value
Double	Double-precision floating-point value
Boolean	`true` or `false` value type
Char	UTF-16 encoded code unit
Decimal	Non-floating-point decimal number
IntPtr	Pointer or handle of a platform-defined size
String	Collection of zero, one, or more Char objects

The intrinsic types listed in the following table are not CLS-Compliant.

NON-COMPLIANT TYPE	DESCRIPTION	CLS-COMPLIANT ALTERNATIVE
SByte	8-bit signed integer data type	Int16
TypedReference	Pointer to an object and its runtime type	None
UInt16	16-bit unsigned integer	Int32
UInt32	32-bit unsigned integer	Int64
UInt64	64-bit unsigned integer	Int64 (may overflow), BigInteger, or Double
UIntPtr	Unsigned pointer or handle	IntPtr

The .NET Framework Class Library or any other class library may include other types that aren't CLS-compliant; for example:

- Boxed value types. The following C# example creates a class that has a public property of type `int*` named `Value`. Because an `int*` is a boxed value type, the compiler flags it as non-CLS-compliant.

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- Typed references, which are special constructs that contain a reference to an object and a reference to a type. Typed references are represented in the .NET Framework by the TypedReference class.

If a type is not CLS-compliant, you should apply the CLSCompliantAttribute attribute with an `isCompliant` value of `false` to it. For more information, see The CLSCompliantAttribute attribute section.

The following example illustrates the problem of CLS compliance in a method signature and in generic type instantiation. It defines an `InvoiceItem` class with a property of type UInt32, a property of type `Nullable(Of UInt32)`, and a constructor with parameters of type UInt32 and `Nullable(Of UInt32)`. You get four compiler warnings when you try to compile this example.

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of UInteger)

   Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
      itemId = sku
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of UInteger)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As UInteger
      Get
         Return invId
      End Get
      Set
         invId = value
      End Set
   End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                             ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not CLS-compliant.
'
'       Public Property InvoiceId As UInteger
'                       ~~~~~~~~~~
```

To eliminate the compiler warnings, replace the non-CLS-compliant types in the `InvoiceItem` public interface with compliant types:

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is zero or negative.");
         invId = (uint) value; }
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

   Private invId As UInteger = 0
   Private itemId As UInteger = 0
   Private qty AS Nullable(Of Integer)

   Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
      If sku <= 0 Then
         Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
      End If
      itemId = CUInt(sku)
      qty = quantity
   End Sub

   Public Property Quantity As Nullable(Of Integer)
      Get
         Return qty
      End Get
      Set
         qty = value
      End Set
   End Property

   Public Property InvoiceId As Integer
      Get
         Return CInt(invId)
      End Get
      Set
         invId = CUInt(value)
      End Set
   End Property
End Class
```

In addition to the specific types listed, some categories of types are not CLS compliant. These include unmanaged pointer types and function pointer types. The following example generates a compiler warning because it uses a pointer to an integer to create an array of integers.

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

For CLS-compliant abstract classes (that is, classes marked as `abstract` in C# or as `MustInherit` in Visual Basic), all members of the class must also be CLS-compliant.

Naming conventions

Because some programming languages are case-insensitive, identifiers (such as the names of namespaces, types, and members) must differ by more than case. Two identifiers are considered equivalent if their lowercase mappings are the same. The following C# example defines two public classes, `Person` and `person`. Because they differ only by case, the C# compiler flags them as not CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{

}

public class person
{

}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

Programming language identifiers, such as the names of namespaces, types, and members, must conform to the Unicode Standard 3.0, Technical Report 15, Annex 7. This means that:

- The first character of an identifier can be any Unicode uppercase letter, lowercase letter, title case letter, modifier letter, other letter, or letter number. For information on Unicode character categories, see the System.Globalization.UnicodeCategory enumeration.
- Subsequent characters can be from any of the categories as the first character, and can also include non-spacing marks, spacing combining marks, decimal numbers, connector punctuations, and formatting codes.

Before you compare identifiers, you should filter out formatting codes and convert the identifiers to Unicode Normalization Form C, because a single character can be represented by multiple UTF-16-encoded code units. Character sequences that produce the same code units in Unicode Normalization Form C are not CLS-compliant. The following example defines a property named `Å`, which consists of the character ANGSTROM SIGN (U+212B), and a second property named `Å`, which consists of the character LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5). Both the C# and Visual Basic compilers flag the source code as non-CLS-compliant.

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing only in case is not
//            CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>
Public Class Size
   Private a1 As Double
   Private a2 As Double

   Public Property Å As Double
       Get
          Return a1
       End Get
       Set
          a1 = value
       End Set
   End Property

   Public Property Å As Double
       Get
          Return a2
       End Get
       Set
          a2 = value
       End Set
   End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

Member names within a particular scope (such as the namespaces within an assembly, the types within a namespace, or the members within a type) must be unique except for names that are resolved through overloading. This requirement is more stringent than that of the common type system, which allows multiple members within a scope to have identical names as long as they are different kinds of members (for example, one is a method and one is a field). In particular, for type members:

- Fields and nested types are distinguished by name alone.
- Methods, properties, and events that have the same name must differ by more than just return type.

The following example illustrates the requirement that member names must be unique within their scope. It defines a class named `Converter` that includes four members named `Conversion`. Three are methods, and one is a property. The method that includes an Int64 parameter is uniquely named, but the two methods with an Int32 parameter are not, because return value is not considered a part of a member's signature. The `Conversion` property also violates this requirement, because properties cannot have the same name as overloaded methods.

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a member called
//             'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains a definition for
//             'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Converter
   Public Function Conversion(number As Integer) As Double
      Return CDbl(number)
   End Function

   Public Function Conversion(number As Integer) As Single
      Return CSng(number)
   End Function

   Public Function Conversion(number As Long) As Double
      Return CDbl(number)
   End Function

   Public ReadOnly Property Conversion As Boolean
      Get
         Return True
      End Get
   End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As Double'
'                    and 'Public Function Conversion(number As Integer) As Single' cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

Individual languages include unique keywords, so languages that target the common language runtime must also provide some mechanism for referencing identifiers (such as type names) that coincide with keywords. For example, `case` is a keyword in both C# and Visual Basic. However, the following Visual Basic example is able to disambiguate a class named `case` from the `case` keyword by using opening and closing braces. Otherwise, the example would produce the error message, "Keyword is not valid as an identifier," and fail to compile.

```
Public Class [case]
   Private _id As Guid
   Private name As String

   Public Sub New(name As String)
      _id = Guid.NewGuid()
      Me.name = name
   End Sub

   Public ReadOnly Property ClientName As String
      Get
         Return name
      End Get
   End Property
End Class
```

The following C# example is able to instantiate the `case` class by using the `@` symbol to disambiguate the identifier from the language keyword. Without it, the C# compiler would display two error messages, "Type expected" and "Invalid expression term 'case'."

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

Type conversion

The Common Language Specification defines two conversion operators:

- `op_Implicit`, which is used for widening conversions that do not result in loss of data or precision. For example, the Decimal structure includes an overloaded `op_Implicit` operator to convert values of integral types and Char values to Decimal values.
- `op_Explicit`, which is used for narrowing conversions that can result in loss of magnitude (a value is converted to a value that has a smaller range) or precision. For example, the Decimal structure includes an overloaded `op_Explicit` operator to convert Double and Single values to Decimal and to convert Decimal values to integral values, Double, Single, and Char.

However, not all languages support operator overloading or the definition of custom operators. If you choose to implement these conversion operators, you should also provide an alternate way to perform the conversion. We recommend that you provide `From`*Xxx* and `To`*Xxx* methods.

The following example defines CLS-compliant implicit and explicit conversions. It creates a `UDouble` class that represents an signed double-precision, floating-point number. It provides for implicit conversions from `UDouble` to Double and for explicit conversions from `UDouble` to Single, Double to `UDouble`, and Single to `UDouble`. It also defines a `ToDouble` method as an alternative to the implicit conversion operator and the `ToSingle`, `FromDouble`, and `FromSingle` methods as alternatives to the explicit conversion operators.

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public static readonly UDouble MinValue = (UDouble) 0.0;
   public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

   public static explicit operator Double(UDouble value)
   {
      return value.number;
```

```
    }

    public static implicit operator Single(UDouble value)
    {
       if (value.number > (double) Single.MaxValue)
          throw new InvalidCastException("A UDouble value is out of range of the Single type.");

       return (float) value.number;
    }

    public static explicit operator UDouble(double value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static implicit operator UDouble(float value)
    {
       if (value < 0)
          throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

       return new UDouble(value);
    }

    public static Double ToDouble(UDouble value)
    {
       return (Double) value;
    }

    public static float ToSingle(UDouble value)
    {
       return (float) value;
    }

    public static UDouble FromDouble(double value)
    {
       return new UDouble(value);
    }

    public static UDouble FromSingle(float value)
    {
       return new UDouble(value);
    }
}
```

```
Public Structure UDouble
   Private number As Double

   Public Sub New(value As Double)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Sub New(value As Single)
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      number = value
   End Sub

   Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
   Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

   Public Shared Widening Operator CType(value As UDouble) As Double
      Return value.number
   End Operator

   Public Shared Narrowing Operator CType(value As UDouble) As Single
      If value.number > CDbl(Single.MaxValue) Then
         Throw New InvalidCastException("A UDouble value is out of range of the Single type.")
      End If
      Return CSng(value.number)
   End Operator

   Public Shared Narrowing Operator CType(value As Double) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Narrowing Operator CType(value As Single) As UDouble
      If value < 0 Then
         Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
      End If
      Return New UDouble(value)
   End Operator

   Public Shared Function ToDouble(value As UDouble) As Double
      Return CType(value, Double)
   End Function

   Public Shared Function ToSingle(value As UDouble) As Single
      Return CType(value, Single)
   End Function

   Public Shared Function FromDouble(value As Double) As UDouble
      Return New UDouble(value)
   End Function

   Public Shared Function FromSingle(value As Single) As UDouble
      Return New UDouble(value)
   End Function
End Structure
```

Arrays

CLS-compliant arrays conform to the following rules:

- All dimensions of an array must have a lower bound of zero. The following example creates a non-CLS-

compliant array with a lower bound of one. Note that, despite the presence of the CLSCompliantAttribute attribute, the compiler does not detect that the array returned by the `Numbers.GetTenPrimes` method is not CLS-compliant.

```
[assembly: CLSCompliant(true)]

public class Numbers
{
   public static Array GetTenPrimes()
   {
      Array arr = Array.CreateInstance(typeof(Int32), new int[] {10}, new int[] {1});
      arr.SetValue(1, 1);
      arr.SetValue(2, 2);
      arr.SetValue(3, 3);
      arr.SetValue(5, 4);
      arr.SetValue(7, 5);
      arr.SetValue(11, 6);
      arr.SetValue(13, 7);
      arr.SetValue(17, 8);
      arr.SetValue(19, 9);
      arr.SetValue(23, 10);

      return arr;
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As Array
      Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
      arr.SetValue(1, 1)
      arr.SetValue(2, 2)
      arr.SetValue(3, 3)
      arr.SetValue(5, 4)
      arr.SetValue(7, 5)
      arr.SetValue(11, 6)
      arr.SetValue(13, 7)
      arr.SetValue(17, 8)
      arr.SetValue(19, 9)
      arr.SetValue(23, 10)

      Return arr
   End Function
End Class
```

- All array elements must consist of CLS-compliant types. The following example defines two methods that return non-CLS-compliant arrays. The first returns an array of UInt32 values. The second returns an Object array that includes Int32 and UInt32 values. Although the compiler identifies the first array as non-compliant because of its UInt32 type, it fails to recognize that the second array includes non-CLS-compliant elements.

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static UInt32[] GetTenPrimes()
   {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
   }

   public static Object[] GetFivePrimes()
   {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
   }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of 'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
   Public Shared Function GetTenPrimes() As UInt32()
      Return { 1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui }
   End Function

   Public Shared Function GetFivePrimes() As Object()
      Dim arr() As Object = { 1, 2, 3, 5ui, 7ui }
      Return arr
   End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not CLS-compliant.
'
'       Public Shared Function GetTenPrimes() As UInt32()
'                              ~~~~~~~~~~~~
```

- Overload resolution for methods that have array parameters is based on the fact that they are arrays and on their element type. For this reason, the following definition of an overloaded `GetSquares` method is CLS-compliant.

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static byte[] GetSquares(byte[] numbers)
   {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
         int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
         if (square <= Byte.MaxValue)
            numbersOut[ctr] = (byte) square;
         // If there's an overflow, assign MaxValue to the corresponding
         // element.
         else
            numbersOut[ctr] = Byte.MaxValue;

      }
      return numbersOut;
   }

   public static BigInteger[] GetSquares(BigInteger[] numbers)
   {
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         numbersOut[ctr] = numbers[ctr] * numbers[ctr];

      return numbersOut;
   }
}
```

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
   Public Function GetSquares(numbers As Byte()) As Byte()
      Dim numbersOut(numbers.Length - 1) As Byte
      For ctr As Integer = 0 To numbers.Length - 1
         Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
         If square <= Byte.MaxValue Then
            numbersOut(ctr) = CByte(square)
         ' If there's an overflow, assign MaxValue to the corresponding
         ' element.
         Else
            numbersOut(ctr) = Byte.MaxValue
         End If
      Next
      Return numbersOut
   End Function

   Public Function GetSquares(numbers As BigInteger()) As BigInteger()
      Dim numbersOut(numbers.Length - 1) As BigInteger
      For ctr As Integer = 0 To numbers.Length - 1
         numbersOut(ctr) = numbers(ctr) * numbers(ctr)
      Next
      Return numbersOut
   End Function
End Module
```

Interfaces

CLS-compliant interfaces can define properties, events, and virtual methods (methods with no implementation).

A CLS-compliant interface cannot have any of the following:

- Static methods or static fields. Both the C# and Visual Basic compilers generate compiler errors if you define a static member in an interface.
- Fields. Both the C# and Visual Basic compilers generate compiler errors if you define a field in an interface.
- Methods that are not CLS-compliant. For example, the following interface definition includes a method, `INumber.GetUnsigned`, that is marked as non-CLS-compliant. This example generates a compiler warning.

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
   int Length();
   [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()': CLS-compliant interfaces
//            must have only CLS-compliant members
```

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
   Function Length As Integer

   <CLSCompliant(False)> Function GetUnsigned As ULong
End Interface
' Attempting to compile the example displays output like the following:
'    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not allowed in a
'    CLS-compliant interface.
'
'       <CLSCompliant(False)> Function GetUnsigned As ULong
'                                      ~~~~~~~~~~~
```

  Because of this rule, CLS-compliant types are not required to implement non-CLS-compliant members. If a CLS-compliant framework does expose a class that implements a non-CLS compliant interface, it should also provide concrete implementations of all non-CLS-compliant members.

CLS-compliant language compilers must also allow a class to provide separate implementations of members that have the same name and signature in multiple interfaces. Both C# and Visual Basic support explicit interface implementations to provide different implementations of identically named methods. Visual Basic also supports the `Implements` keyword, which enables you to explicitly designate which interface and member a particular member implements. The following example illustrates this scenario by defining a `Temperature` class that implements the `ICelsius` and `IFahrenheit` interfaces as explicit interface implementations.

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        cTemp.GetTemperature());
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        fTemp.GetTemperature());
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

```
<Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
   Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
   Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
   Private _value As Decimal

   Public Sub New(value As Decimal)
      ' We assume that this is the Celsius value.
      _value = value
   End Sub

   Public Function GetFahrenheit() As Decimal _
          Implements IFahrenheit.GetTemperature
      Return _value * 9 / 5 + 32
   End Function

   Public Function GetCelsius() As Decimal _
          Implements ICelsius.GetTemperature
      Return _value
   End Function
End Class

Module Example
   Public Sub Main()
      Dim temp As New Temperature(100.0d)
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        temp.GetCelsius())
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        temp.GetFahrenheit())
   End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

Enumerations

CLS-compliant enumerations must follow these rules:

- The underlying type of the enumeration must be an intrinsic CLS-compliant integer (Byte, Int16, Int32, or Int64). For example, the following code tries to define an enumeration whose underlying type is UInt32 and generates a compiler warning.

```
using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
   Unspecified = 0,
   XSmall = 1,
   Small = 2,
   Medium = 3,
   Large = 4,
   XLarge = 5
};

public class Clothing
{
   public string Name;
   public string Type;
   public string Size;
}
// The attempt to compile the example displays a compiler warning like the following:
//    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
   Unspecified = 0
   XSmall = 1
   Small = 2
   Medium = 3
   Large = 4
   XLarge = 5
End Enum

Public Class Clothing
   Public Name As String
   Public Type As String
   Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- An enumeration type must have a single instance field named `Value__` that is marked with the FieldAttributes.RTSpecialName attribute. This enables you to reference the field value implicitly.
- An enumeration includes literal static fields whose types match the type of the enumeration itself. For example, if you define a `State` enumeration with values of `State.On` and `State.Off`, `State.On` and `State.Off` are both literal static fields whose type is `State`.
- There are two kinds of enumerations:
  - An enumeration that represents a set of mutually exclusive, named integer values. This type of enumeration is indicated by the absence of the System.FlagsAttribute custom attribute.
  - An enumeration that represents a set of bit flags that can combine to generate an unnamed value. This type of enumeration is indicated by the presence of the System.FlagsAttribute custom attribute.

For more information, see the documentation for the Enum structure.

- The value of an enumeration is not limited to the range of its specified values. In other words, the range of values in an enumeration is the range of its underlying value. You can use the Enum.IsDefined method to determine whether a specified value is a member of an enumeration.

Type members in general

The Common Language Specification requires all fields and methods to be accessed as members of a particular class. Therefore, global static fields and methods (that is, static fields or methods that are defined apart from a type) are not CLS-compliant. If you try to include a global field or method in your source code, both the C# and Visual Basic compilers generate a compiler error.

The Common Language Specification supports only the standard managed calling convention. It doesn't support unmanaged calling conventions and methods with variable argument lists marked with the `varargs` keyword. For variable argument lists that are compatible with the standard managed calling convention, use the ParamArrayAttribute attribute or the individual language's implementation, such as the `params` keyword in C# and the `ParamArray` keyword in Visual Basic.

Member accessibility

Overriding an inherited member cannot change the accessibility of that member. For example, a public method in a base class cannot be overridden by a private method in a derived class. There is one exception: a `protected internal` (in C#) or `Protected Friend` (in Visual Basic) member in one assembly that is overridden by a type in a different assembly. In that case, the accessibility of the override is `Protected`.

The following example illustrates the error that is generated when the CLSCompliantAttribute attribute is set to `true`, and `Human`, which is a class derived from `Animal`, tries to change the accessibility of the `Species` property from public to private. The example compiles successfully if its accessibility is changed to public.

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be private
```

```
<Assembly: CLSCompliant(True)>

Public Class Animal
   Private _species As String

   Public Sub New(species As String)
      _species = species
   End Sub

   Public Overridable ReadOnly Property Species As String
      Get
         Return _species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _species
   End Function
End Class

Public Class Human : Inherits Animal
   Private _name As String

   Public Sub New(name As String)
      MyBase.New("Homo Sapiens")
      _name = name
   End Sub

   Public ReadOnly Property Name As String
      Get
         Return _name
      End Get
   End Property

   Private Overrides ReadOnly Property Species As String
      Get
         Return MyBase.Species
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return _name
   End Function
End Class

Public Module Example
   Public Sub Main()
      Dim p As New Human("John")
      Console.WriteLine(p.Species)
      Console.WriteLine(p.ToString())
   End Sub
End Module
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

Types in the signature of a member must be accessible whenever that member is accessible. For example, this means that a public member cannot include a parameter whose type is private, protected, or internal. The following example illustrates the compiler error that results when a `StringWrapper` class constructor exposes an internal `StringOperationType` enumeration value that determines how a string value should be wrapped.

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//            'StringOperationType' is less accessible than method
//            'StringWrapper.StringWrapper(StringOperationType)'
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class StringWrapper

   Dim internalString As String
   Dim internalSB As StringBuilder = Nothing
   Dim useSB As Boolean = False

   Public Sub New(type As StringOperationType)
      If type = StringOperationType.Normal Then
         useSB = False
      Else
         internalSB = New StringBuilder()
         useSB = True
      End If
   End Sub

   ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
   Normal = 0
   Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                                ~~~~~~~~~~~~~~~~~~~
```

Generic types and members

Nested types always have at least as many generic parameters as their enclosing type. These correspond by

position to the generic parameters in the enclosing type. The generic type can also include new generic parameters.

The relationship between the generic type parameters of a containing type and its nested types may be hidden by the syntax of individual languages. In the following example, a generic type `Outer<T>` contains two nested classes, `Inner1A` and `Inner1B<U>`. The calls to the `ToString` method, which each class inherits from Object.ToString, show that each nested class includes the type parameters of its containing class.

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);

      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

```
<Assembly:CLSCompliant(True)>

Public Class Outer(Of T)
   Dim value As T

   Public Sub New(value As T)
      Me.value = value
   End Sub

   Public Class Inner1A : Inherits Outer(Of T)
      Public Sub New(value As T)
         MyBase.New(value)
      End Sub
   End Class

   Public Class Inner1B(Of U) : Inherits Outer(Of T)
      Dim value2 As U

      Public Sub New(value1 As T, value2 As U)
         MyBase.New(value1)
         Me.value2 = value2
      End Sub
   End Class
End Class

Public Module Example
   Public Sub Main()
      Dim inst1 As New Outer(Of String)("This")
      Console.WriteLine(inst1)

      Dim inst2 As New Outer(Of String).Inner1A("Another")
      Console.WriteLine(inst2)

      Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
      Console.WriteLine(inst3)
   End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

Generic type names are encoded in the form *name*`*n*, where *name* is the type name, ` is a character literal, and *n* is the number of parameters declared on the type, or, for nested generic types, the number of newly introduced type parameters. This encoding of generic type names is primarily of interest to developers who use reflection to access CLS-complaint generic types in a library.

If constraints are applied to a generic type, any types used as constraints must also be CLS-compliant. The following example defines a class named `BaseClass` that is not CLS-compliant and a generic class named `BaseCollection` whose type parameter must derive from `BaseClass`. But because `BaseClass` is not CLS-compliant, the compiler emits a warning.

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}


public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class


Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~~
```

If a generic type is derived from a generic base type, it must redeclare any constraints so that it can guarantee that constraints on the base type are also satisfied. The following example defines a `Number<T>` that can represent any numeric type. It also defines a `FloatingPoint<T>` class that represents a floating point value. However, the source code fails to compile, because it does not apply the constraint on `Number<T>` (that T must be a value type) to `FloatingPoint<T>`.

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
// The attempt to comple the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//               order to use it as parameter 'T' in the generic type or method 'Number<T>'
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
' The attempt to comple the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                    ~
```

The example compiles successfully if the constraint is added to the `FloatingPoint<T>` class.

```
using System;

[assembly:CLSCompliant(true)]


public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
```

```
<Assembly:CLSCompliant(True)>

Public Class Number(Of T As Structure)
   ' Use Double as the underlying type, since its range is a superset of
   ' the ranges of all numeric types except BigInteger.
   Protected number As Double

   Public Sub New(value As T)
      Try
         Me.number = Convert.ToDouble(value)
      Catch e As OverflowException
         Throw New ArgumentException("value is too large.", e)
      Catch e As InvalidCastException
         Throw New ArgumentException("The value parameter is not numeric.", e)
      End Try
   End Sub

   Public Function Add(value As T) As T
      Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
   End Function

   Public Function Subtract(value As T) As T
      Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
   End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
   Public Sub New(number As T)
      MyBase.New(number)
      If TypeOf number Is Single Or
               TypeOf number Is Double Or
               TypeOf number Is Decimal Then
         Me.number = Convert.ToDouble(number)
      Else
         throw new ArgumentException("The number parameter is not a floating-point number.")
      End If
   End Sub
End Class
```

The Common Language Specification imposes a conservative per-instantiation model for nested types and protected members. Open generic types cannot expose fields or members with signatures that contain a specific instantiation of a nested, protected generic type. Non-generic types that extend a specific instantiation of a generic base class or interface cannot expose fields or members with signatures that contain a different instantiation of a nested, protected generic type.

The following example defines a generic type, `C1<T>` (or `C1(Of T)` in Visual Basic), and a protected class, `C1<T>.N` (or `C1(Of T).N` in Visual Basic). `C1<T>` has two methods, `M1` and `M2`. However, `M1` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(Of Integer).N`) object from C1<T> (or `C1(Of T)`). A second class, `C2`, is derived from `C1<long>` (or `C1(Of Long)`). It has two methods, `M3` and `M4`. `M3` is not CLS-compliant because it tries to return a `C1<int>.N` (or `C1(Of Integer).N`) object from a subclass of `C1<long>`. Note that language compilers can be even more restrictive. In this example, Visual Basic displays an error when it tries to compile `M4`.

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                     // accessible from within C1<T> in all
                                     // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                     // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is not
                                       // accessible in C2 (extends C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
```

```
<Assembly:CLSCompliant(True)>

Public Class C1(Of T)
   Protected Class N
   End Class

   Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
                                             ' accessible from within C1(Of T) in all
   End Sub                                   ' languages


   Protected Sub M2(n As C1(Of T).N)     ' CLS-compliant – C1(Of T).N accessible
   End Sub                               ' inside C1(Of T)
End Class

Public Class C2 : Inherits C1(Of Long)
   Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant – C1(Of Integer).N is not
   End Sub                                   ' accessible in C2 (extends C1(Of Long))

   Protected Sub M4(n As C1(Of Long).N)
   End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
'                                 ~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant - C1(Of Integer).N is not
'
'                                 ~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is 'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                                 ~~~~~~~~~~~~
```

Constructors

Constructors in CLS-compliant classes and structures must follow these rules:

- A constructor of a derived class must call the instance constructor of its base class before it accesses inherited instance data. This requirement is due to the fact that base class constructors are not inherited by their derived classes. This rule does not apply to structures, which do not support direct inheritance.

  Typically, compilers enforce this rule independently of CLS compliance, as the following example shows. It creates a `Doctor` class that is derived from a `Person` class, but the `Doctor` class fails to call the `Person` class constructor to initialize inherited instance fields.

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private string fName, lName, _id;

   public Person(string firstName, string lastName, string id)
   {
      if (String.IsNullOrEmpty(firstName + lastName))
         throw new ArgumentNullException("Either a first name or a last name must be provided.");

      fName = firstName;
      lName = lastName;
      _id = id;
   }

   public string FirstName
   {
      get { return fName; }
   }

   public string LastName
   {
      get { return lName; }
   }

   public string Id
   {
      get { return _id; }
   }

   public override string ToString()
   {
      return String.Format("{0}{1}{2}", fName,
                           String.IsNullOrEmpty(fName) ?  "" : " ",
                           lName);
   }
}

public class Doctor : Person
{
   public Doctor(string firstName, string lastName, string id)
   {
   }

   public override string ToString()
   {
      return "Dr. " + base.ToString();
   }
}
// Attempting to compile the example displays output like the following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a constructor that takes 0
//            arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
   Private fName, lName, _id As String

   Public Sub New(firstName As String, lastName As String, id As String)
      If String.IsNullOrEmpty(firstName + lastName) Then
         Throw New ArgumentNullException("Either a first name or a last name must be provided.")
      End If

      fName = firstName
      lName = lastName
      _id = id
   End Sub

   Public ReadOnly Property FirstName As String
      Get
         Return fName
      End Get
   End Property

   Public ReadOnly Property LastName As String
      Get
         Return lName
      End Get
   End Property

   Public ReadOnly Property Id As String
      Get
         Return _id
      End Get
   End Property

   Public Overrides Function ToString() As String
      Return String.Format("{0}{1}{2}", fName,
                           If(String.IsNullOrEmpty(fName), "", " "),
                           lName)
   End Function
End Class

Public Class Doctor : Inherits Person
   Public Sub New(firstName As String, lastName As String, id As String)
   End Sub

   Public Overrides Function ToString() As String
      Return "Dr. " + MyBase.ToString()
   End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor' does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- An object constructor cannot be called except to create an object. In addition, an object cannot be initialized twice. For example, this means that Object.MemberwiseClone and deserialization methods such as BinaryFormatter.Deserialize must not call constructors.

Properties

Properties in CLS-compliant types must follow these rules:

- A property must have a setter, a getter, or both. In an assembly, these are implemented as special methods, which means that they will appear as separate methods (the getter is named `get_`

propertyname and the setter is `set_` *propertyname*) marked as `SpecialName` in the assembly's metadata. The C# and Visual Basic compilers enforce this rule automatically without the need to apply the CLSCompliantAttribute attribute.

- A property's type is the return type of the property getter and the last argument of the setter. These types must be CLS compliant, and arguments cannot be assigned to the property by reference (that is, they cannot be managed pointers).

- If a property has both a getter and a setter, they must both be virtual, both static, or both instance. The C# and Visual Basic compilers automatically enforce this rule through their property definition syntax.

Events

An event is defined by its name and its type. The event type is a delegate that is used to indicate the event. For example, the AppDomain.AssemblyResolve event is of type ResolveEventHandler. In addition to the event itself, three methods with names based on the event name provide the event's implementation and are marked as `SpecialName` in the assembly's metadata:

- A method for adding an event handler, named `add_` *EventName*. For example, the event subscription method for the AppDomain.AssemblyResolve event is named `add_AssemblyResolve`.

- A method for removing an event handler, named `remove_` *EventName*. For example, the removal method for the AppDomain.AssemblyResolve event is named `remove_AssemblyResolve`.

- A method for indicating that the event has occurred, named `raise_` *EventName*.

> **NOTE**
> Most of the Common Language Specification's rules regarding events are implemented by language compilers and are transparent to component developers.

The methods for adding, removing, and raising the event must have the same accessibility. They must also all be static, instance, or virtual. The methods for adding and removing an event have one parameter whose type is the event delegate type. The add and remove methods must both be present or both be absent.

The following example defines a CLS-compliant class named `Temperature` that raises a `TemperatureChanged` event if the change in temperature between two readings equals or exceeds a threshold value. The `Temperature` class explicitly defines a `raise_TemperatureChanged` method so that it can selectively execute event handlers.

```
using System;
using System.Collections;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new, DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
      when = time;
   }

   public Decimal OldTemperature
   {
      get { return originalTemp; }
```

```
      get { return originalTemp; }
   }

   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender, TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not executed.");
         else
            d.Invoke(this, eventArgs);
      }
   }
}

public class Example
```

```
public class Example
{
   public Temperature temp;

   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
   {
      temp = t;
      RecordTemperatures();
   }

   public void RecordTemperatures()
   {
      temp.TemperatureChanged += this.DuplicateTemperatureNotification;
      temp.CurrentTemperature = 66;
      temp.CurrentTemperature = 63;
   }

   internal void TemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }

   public void DuplicateTemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }
}
```

```
Imports System.Collections
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs   : Inherits EventArgs
   Private originalTemp As Decimal
   Private newTemp As Decimal
   Private [when] As DateTimeOffset

   Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
      originalTemp = original
      newTemp = [new]
      [when] = [time]
   End Sub

   Public ReadOnly Property OldTemperature As Decimal
      Get
         Return originalTemp
      End Get
   End Property

   Public ReadOnly Property CurrentTemperature As Decimal
      Get
```

```
      Get
         Return newTemp
      End Get
   End Property

   Public ReadOnly Property [Time] As DateTimeOffset
      Get
         Return [when]
      End Get
   End Property
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As TemperatureChangedEventArgs)

Public Class Temperature
   Private Structure TemperatureInfo
      Dim Temperature As Decimal
      Dim Recorded As DateTimeOffset
   End Structure

   Public Event TemperatureChanged As TemperatureChanged

   Private previous As Decimal
   Private current As Decimal
   Private tolerance As Decimal
   Private tis As New List(Of TemperatureInfo)

   Public Sub New(temperature As Decimal, tolerance As Decimal)
      current = temperature
      Dim ti As New TemperatureInfo()
      ti.Temperature = temperature
      ti.Recorded = DateTimeOffset.UtcNow
      tis.Add(ti)
      Me.tolerance = tolerance
   End Sub

   Public Property CurrentTemperature As Decimal
      Get
         Return current
      End Get
      Set
         Dim ti As New TemperatureInfo
         ti.Temperature = value
         ti.Recorded = DateTimeOffset.UtcNow
         previous = current
         current = value
         If Math.Abs(current - previous) >= tolerance Then
            raise_TemperatureChanged(New TemperatureChangedEventArgs(previous, current, ti.Recorded))
         End If
      End Set
   End Property

   Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
      If TemperatureChangedEvent Is Nothing Then Exit Sub

      Dim ListenerList() As System.Delegate = TemperatureChangedEvent.GetInvocationList()
      For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
         If d.Method.Name.Contains("Duplicate") Then
            Console.WriteLine("Duplicate event handler; event handler not executed.")
         Else
            d.Invoke(Me, eventArgs)
         End If
      Next
   End Sub
End Class

Public Class Example
   Public WithEvents temp As Temperature
```

```
    Public Shared Sub Main()
       Dim ex As New Example()
    End Sub

    Public Sub New()
       temp = New Temperature(65, 3)
       RecordTemperatures()
       Dim ex As New Example(temp)
       ex.RecordTemperatures()
    End Sub

    Public Sub New(t As Temperature)
       temp = t
       RecordTemperatures()
    End Sub

    Public Sub RecordTemperatures()
       temp.CurrentTemperature = 66
       temp.CurrentTemperature = 63

    End Sub

    Friend Shared Sub TemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
       Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub

    Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
       Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub
End Class
```

Overloads

The Common Language Specification imposes the following requirements on overloaded members:

- Members can be overloaded based on the number of parameters and the type of any parameter. Calling convention, return type, custom modifiers applied to the method or its parameter, and whether parameters are passed by value or by reference are not considered when differentiating between overloads. For an example, see the code for the requirement that names must be unique within a scope in the Naming conventions section.
- Only properties and methods can be overloaded. Fields and events cannot be overloaded.
- Generic methods can be overloaded based on the number of their generic parameters.

> **NOTE**
>
> The `op_Explicit` and `op_Implicit` operators are exceptions to the rule that return value is not considered part of a method signature for overload resolution. These two operators can be overloaded based on both their parameters and their return value.

Exceptions

Exception objects must derive from System.Exception or from another type derived from System.Exception. The following example illustrates the compiler error that results when a custom class named `ErrorClass` is used for exception handling.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be derived from
//            System.Exception
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from 'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

To correct this error, the `ErrorClass` class must inherit from System.Exception. In addition, the `Message` property must be overridden. The following example corrects these errors to define an `ErrorClass` class that is CLS-compliant.

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
   Dim msg As String

   Public Sub New(errorMessage As String)
      msg = errorMessage
   End Sub

   Public Overrides ReadOnly Property Message As String
      Get
         Return msg
      End Get
   End Property
End Class

Public Module StringUtilities
   <Extension()> Public Function SplitString(value As String, index As Integer) As String()
      If index < 0 Or index > value.Length Then
         Dim BadIndex As New ErrorClass("The index is not within the string.")
         Throw BadIndex
      End If
      Dim retVal() As String = { value.Substring(0, index - 1),
                                 value.Substring(index) }
      Return retVal
   End Function
End Module
```

Attributes

In.NET Framework assemblies, custom attributes provide an extensible mechanism for storing custom attributes and retrieving metadata about programming objects, such as assemblies, types, members, and method parameters. Custom attributes must derive from System.Attribute or from a type derived from System.Attribute.

The following example violates this rule. It defines a `NumericAttribute` class that does not derive from System.Attribute. Note that a compiler error results only when the non-CLS-compliant attribute is applied, not when the class is defined.

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
   Private _isNumeric As Boolean

   Public Sub New(isNumeric As Boolean)
      _isNumeric = isNumeric
   End Sub

   Public ReadOnly Property IsNumeric As Boolean
      Get
         Return _isNumeric
      End Get
   End Property
End Class

<Numeric(True)> Public Structure UDouble
   Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~
```

The constructor or the properties of a CLS-compliant attribute can expose only the following types:

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- Any enumeration type whose underlying type is Byte, Int16, Int32, or Int64.

The following example defines a `DescriptionAttribute` class that derives from Attribute. The class constructor has a parameter of type `Descriptor`, so the class is not CLS-compliant. Note that the C# compiler emits a warning but compiles successfully, whereas the Visual Basic compiler emits neither a warning nor an error.

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//               constructors which use only CLS-compliant types
```

```
<Assembly:CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
   Type = 0
   Member = 1
End Enum

Public Class Descriptor
   Public Type As DescriptorType
   Public Description As String
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
   Private desc As Descriptor

   Public Sub New(d As Descriptor)
      desc = d
   End Sub

   Public ReadOnly Property Descriptor As Descriptor
      Get
         Return desc
      End Get
   End Property
End Class
```

The CLSCompliantAttribute attribute

The CLSCompliantAttribute attribute is used to indicate whether a program element complies with the Common Language Specification. The CLSCompliantAttribute.CLSCompliantAttribute(Boolean) constructor includes a single required parameter, `isCompliant`, that indicates whether the program element is CLS-compliant.

At compile time, the compiler detects non-compliant elements that are presumed to be CLS-compliant and emits a warning. The compiler does not emit warnings for types or members that are explicitly declared to be non-compliant.

Component developers can use the CLSCompliantAttribute attribute in two ways:

- To define the parts of the public interface exposed by a component that are CLS-compliant and the parts that are not CLS-compliant. When the attribute is used to mark particular program elements as CLS-compliant, its use guarantees that those elements are accessible from all languages and tools that target the .NET Framework.
- To ensure that the component library's public interface exposes only program elements that are CLS-compliant. If elements are not CLS-compliant, compilers will generally issue a warning.

> **WARNING**
> In some cases, language compilers enforce CLS-compliant rules regardless of whether the CLSCompliantAttribute attribute is used. For example, defining a static member in an interface violates a CLS rule. In this regard, if you define a `static` (in C#) or `Shared` (in Visual Basic) member in an interface, both the C# and Visual Basic compilers display an error message and fail to compile the app.

The CLSCompliantAttribute attribute is marked with an AttributeUsageAttribute attribute that has a value of AttributeTargets.All. This value allows you to apply the CLSCompliantAttribute attribute to any program element, including assemblies, modules, types (classes, structures, enumerations, interfaces, and delegates), type

members (constructors, methods, properties, fields, and events), parameters, generic parameters, and return values. However, in practice, you should apply the attribute only to assemblies, types, and type members. Otherwise, compilers ignore the attribute and continue to generate compiler warnings whenever they encounter a non-compliant parameter, generic parameter, or return value in your library's public interface.

The value of the CLSCompliantAttribute attribute is inherited by contained program elements. For example, if an assembly is marked as CLS-compliant, its types are also CLS-compliant. If a type is marked as CLS-compliant, its nested types and members are also CLS-compliant.

You can explicitly override the inherited compliance by applying the CLSCompliantAttribute attribute to a contained program element. For example, you can use the CLSCompliantAttribute attribute with an `isCompliant` value of `false` to define a non-compliant type in a compliant assembly, and you can use the attribute with an `isCompliant` value of `true` to define a compliant type in a non-compliant assembly. You can also define non-compliant members in a compliant type. However, a non-compliant type cannot have compliant members, so you cannot use the attribute with an `isCompliant` value of `true` to override inheritance from a non-compliant type.

When you are developing components, you should always use the CLSCompliantAttribute attribute to indicate whether your assembly, its types, and its members are CLS-compliant.

To create CLS-compliant components:

1. Use the CLSCompliantAttribute to mark you assembly as CLS-compliant.
2. Mark any publicly exposed types in the assembly that are not CLS-compliant as non-compliant.
3. Mark any publicly exposed members in CLS-compliant types as non-compliant.
4. Provide a CLS-compliant alternative for non-CLS-compliant members.

If you've successfully marked all your non-compliant types and members, your compiler should not emit any non-compliance warnings. However, you should indicate which members are not CLS-compliant and list their CLS-compliant alternatives in your product documentation.

The following example uses the CLSCompliantAttribute attribute to define a CLS-compliant assembly and a type, `CharacterUtilities`, that has two non-CLS-compliant members. Because both members are tagged with the `CLSCompliant(false)` attribute, the compiler produces no warnings. The class also provides a CLS-compliant alternative for both methods. Ordinarily, we would just add two overloads to the `ToUTF16` method to provide CLS-compliant alternatives. However, because methods cannot be overloaded based on return value, the names of the CLS-compliant methods are different from the names of the non-compliant methods.

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

```
Imports System.Text

<Assembly:CLSCompliant(True)>

Public Class CharacterUtilities
   <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
      s = s.Normalize(NormalizationForm.FormC)
      Return Convert.ToUInt16(s(0))
   End Function

   <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
      Return Convert.ToUInt16(ch)
   End Function

   ' CLS-compliant alternative for ToUTF16(String).
   Public Shared Function ToUTF16CodeUnit(s As String) As Integer
      s = s.Normalize(NormalizationForm.FormC)
      Return CInt(Convert.ToInt16(s(0)))
   End Function

   ' CLS-compliant alternative for ToUTF16(Char).
   Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
      Return Convert.ToInt32(ch)
   End Function

   Public Function HasMultipleRepresentations(s As String) As Boolean
      Dim s1 As String = s.Normalize(NormalizationForm.FormC)
      Return s.Equals(s1)
   End Function

   Public Function GetUnicodeCodePoint(ch As Char) As Integer
      If Char.IsSurrogate(ch) Then
         Throw New ArgumentException("ch cannot be a high or low surrogate.")
      End If
      Return Char.ConvertToUtf32(ch.ToString(), 0)
   End Function

   Public Function GetUnicodeCodePoint(chars() As Char) As Integer
      If chars.Length > 2 Then
         Throw New ArgumentException("The array has too many characters.")
      End If
      If chars.Length = 2 Then
         If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
            Throw New ArgumentException("The array must contain a low and a high surrogate.")
         Else
            Return Char.ConvertToUtf32(chars(0), chars(1))
         End If
      Else
         Return Char.ConvertToUtf32(chars.ToString(), 0)
      End If
   End Function
End Class
```

If you are developing an app rather than a library (that is, if you aren't exposing types or members that can be consumed by other app developers), the CLS compliance of the program elements that your app consumes are of interest only if your language does not support them. In that case, your language compiler will generate an error when you try to use a non-CLS-compliant element.

Cross-Language Interoperability

Language independence has a number of possible meanings. One meaning, which is discussed in the article Language Independence and Language-Independent Components, involves seamlessly consuming types written in one language from an app written in another language. A second meaning, which is the focus of this article, involves combining code written in multiple languages into a single .NET Framework assembly.

The following example illustrates cross-language interoperability by creating a class library named Utilities.dll that includes two classes, `NumericLib` and `StringLib`. The `NumericLib` class is written in C#, and the `StringLib` class is written in Visual Basic. Here's the source code for StringUtil.vb, which includes a single member, `ToTitleCase`, in its `StringLib` class.

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
   Private exclusions As List(Of String)

   Sub New()
      Dim words() As String = { "a", "an", "and", "of", "the" }
      exclusions = New List(Of String)
      exclusions.AddRange(words)
   End Sub

   <Extension()> _
   Public Function ToTitleCase(title As String) As String
      Dim words() As String = title.Split()
      Dim result As String = String.Empty

      For ctr As Integer = 0 To words.Length - 1
         Dim word As String = words(ctr)
         If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
            result += word.Substring(0, 1).ToUpper() + _
                      word.Substring(1).ToLower()
         Else
            result += word.ToLower()
         End If
         If ctr <= words.Length - 1 Then
            result += " "
         End If
      Next
      Return result
   End Function
End Module
```

Here's the source code for NumberUtil.cs, which defines a `NumericLib` class that has two members, `IsEven` and `NearZero`.

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return Convert.ToInt64(number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) % 2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer value.");
   }

   public static bool NearZero(double number)
   {
      return Math.Abs(number) < .00001;
   }
}
```

To package the two classes in a single assembly, you must compile them into modules. To compile the Visual Basic source code file into a module, use this command:

```
vbc /t:module StringUtil.vb
```

For more information about the command-line syntax of the Visual Basic compiler, see Building from the Command Line.

To compile the C# source code file into a module, use this command:

```
csc /t:module NumberUtil.cs
```

For more information about the command-line syntax of the C# compiler, see Command-line Building With csc.exe.

You then use the Linker options to compile the two modules into an assembly:

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

The following example then calls the `NumericLib.NearZero` and `StringLib.ToTitleCase` methods. Note that both the Visual Basic code and the C# code are able to access the methods in both classes.

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

```
Module Example
   Public Sub Main()
      Dim dbl As Double = 0.0 - Double.Epsilon
      Console.WriteLine(NumericLib.NearZero(dbl))

      Dim s As String = "war and peace"
      Console.WriteLine(s.ToTitleCase())
   End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

To compile the Visual Basic code, use this command:

```
vbc example.vb /r:UtilityLib.dll
```

To compile with C#, change the name of the compiler from **vbc** to **csc**, and change the file extension from .vb to .cs:

```
csc example.cs /r:UtilityLib.dll
```

See also

- CLSCompliantAttribute

Framework Libraries

7/9/2019 • 2 minutes to read • Edit Online

.NET has an expansive standard set of class libraries, referred to as either the base class libraries (core set) or framework class libraries (complete set). These libraries provide implementations for many general and app-specific types, algorithms and utility functionality. Both commercial and community libraries build on top of the framework class libraries, providing easy to use off-the-shelf libraries for a wide set of computing tasks.

A subset of these libraries are provided with each .NET implementation. Base Class Library (BCL) APIs are expected with any .NET implementation, both because developers will want them and because popular libraries will need them to run. App-specific libraries above the BCL, such as ASP.NET, will not be available on all .NET implementations.

Base Class Libraries

The BCL provides the most foundational types and utility functionality and are the base of all other .NET class libraries. They aim to provide very general implementations without any bias to any workload. Performance is always an important consideration, since apps might prefer a particular policy, such as low-latency to high-throughput or low-memory to low-CPU usage. These libraries are intended to be high-performance generally, and take a middle-ground approach according to these various performance concerns. For most apps, this approach has been quite successful.

Primitive Types

.NET includes a set of primitive types, which are used (to varying degrees) in all programs. These types contain data, such as numbers, strings, bytes and arbitrary objects. The C# language includes keywords for these types. A sample set of these types is listed below, with the matching C# keywords.

- System.Object (object) - The ultimate base class in the CLR type system. It is the root of the type hierarchy.
- System.Int16 (short) - A 16-bit signed integer type. The unsigned UInt16 also exists.
- System.Int32 (int) - A 32-bit signed integer type. The unsigned UInt32 also exists.
- System.Single (float) - A 32-bit floating-point type.
- System.Decimal (decimal) - A 128-bit decimal type.
- System.Byte (byte) - An unsigned 8-bit integer that represents a byte of memory.
- System.Boolean (bool) - A boolean type that represents `true` or `false`.
- System.Char (char) - A 16-bit numeric type that represents a Unicode character.
- System.String (string) - Represents a series of characters. Different than a `char[]`, but enables indexing into each individual `char` in the `string`.

Data Structures

.NET includes a set of data structures that are the workhorses of almost any .NET apps. These are mostly collections, but also include other types.

- Array - Represents an array of strongly types objects that can be accessed by index. Has a fixed size, per its construction.
- List<T> - Represents a strongly typed list of objects that can be accessed by index. Is automatically resized as needed.
- Dictionary<TKey,TValue> - Represents a collection of values that are indexed by a key. Values can be accessed

via key. Is automatically resized as needed.

- Uri - Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.
- DateTime - Represents an instant in time, typically expressed as a date and time of day.

Utility APIs

.NET includes a set of utility APIs that provide functionality for many important tasks.

- HttpClient - An API for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.
- XDocument - An API for loading, and querying XML documents with LINQ.
- StreamReader - An API for reading files.
- StreamWriter - An API for writing files.

App-Model APIs

There are many app-models that can be used with .NET, provided by several companies.

- ASP.NET - Provides a web framework for building Web sites and services. Supported on Windows, Linux and macOS (depends on ASP.NET version).

.NET class library overview

5/14/2019 • 4 minutes to read • Edit Online

.NET implementations include classes, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose compiler conforms to the common language specification (CLS).

.NET types are the foundation on which .NET applications, components, and controls are built. .NET implementations include types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET Framework security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

.NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name — up to the rightmost dot — is the namespace name. The last part of the name is the type name. For example, `System.Collections.Generic.List<T>` represents the `List<T>` type, which belongs to the `System.Collections.Generic` namespace. The types in System.Collections.Generic can be used to work with generic collections.

This naming scheme makes it easy for library developers extending the .NET Framework to create hierarchical groups of types and name them in a consistent, informative manner. It also allows types to be unambiguously identified by their full name (that is, by their namespace and type name), which prevents type name collisions. Library developers are expected to use the following convention when creating names for their namespaces:

CompanyName.TechnologyName

For example, the namespace `Microsoft.Word` conforms to this guideline.

The use of naming patterns to group related types into namespaces is a very useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

For more information on namespaces and type names, see Common Type System.

System namespace

The System namespace is the root namespace for fundamental types in .NET. This namespace includes classes that represent the base data types used by all applications: Object (the root of the inheritance hierarchy), Byte, Char, Array, Int32, String, and so on. Many of these types correspond to the primitive data types that your programming language uses. When you write code using .NET Framework types, you can use your language's corresponding keyword when a .NET Framework base data type is expected.

The following table lists the base types that .NET supplies, briefly describes each type, and indicates the corresponding type in Visual Basic, C#, C++, and F#.

CATEGORY	CLASS NAME	DESCRIPTION	VISUAL BASIC DATA TYPE	C# DATA TYPE	C++/CLI DATA TYPE	F# DATA TYPE
Integer	Byte	An 8-bit unsigned integer.	**Byte**	**byte**	**unsigned char**	**byte**
	SByte	An 8-bit signed integer. Not CLS-compliant.	**SByte**	**sbyte**	**char** -or- **signed char**	**sbyte**
	Int16	A 16-bit signed integer.	**Short**	**short**	**short**	**int16**
	Int32	A 32-bit signed integer.	**Integer**	**int**	**int** -or- **long**	**int**
	Int64	A 64-bit signed integer.	**Long**	**long**	**__int64**	**int64**
	UInt16	A 16-bit unsigned integer. Not CLS-compliant.	**UShort**	**ushort**	**unsigned short**	**uint16**
	UInt32	A 32-bit unsigned integer. Not CLS-compliant.	**UInteger**	**uint**	**unsigned int** -or- **unsigned long**	**uint32**
	UInt64	A 64-bit unsigned integer. Not CLS-compliant.	**ULong**	**ulong**	**unsigned __int64**	**uint64**

CATEGORY	CLASS NAME	DESCRIPTION	VISUAL BASIC DATA TYPE	C# DATA TYPE	C++/CLI DATA TYPE	F# DATA TYPE
Floating point	Single	A single-precision (32-bit) floating-point number.	**Single**	**float**	**float**	**float32** or **single**
	Double	A double-precision (64-bit) floating-point number.	**Double**	**double**	**double**	**float** or **double**
Logical	Boolean	A Boolean value (true or false).	**Boolean**	**bool**	**bool**	**bool**
Other	Char	A Unicode (16-bit) character.	**Char**	**char**	**wchar_t**	**char**
	Decimal	A decimal (128-bit) value.	**Decimal**	**decimal**	**Decimal**	**decimal**
	IntPtr	A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	**IntPtr** No built-in type.	**IntPtr** No built-in type.	**IntPtr** No built-in type.	**unativeint**
	UIntPtr	An unsigned integer whose size depends on the underlying platform (a 32- bit value on a 32-bit platform and a 64-bit value on a 64-bit platform). Not CLS-compliant.	**UIntPtr** No built-in type.	**UIntPtr** No built-in type.	**UIntPtr** No built-in type.	**unativeint**
	Object	The root of the object hierarchy.	**Object**	**object**	**Object^**	**obj**

CATEGORY	CLASS NAME	DESCRIPTION	VISUAL BASIC DATA TYPE	C# DATA TYPE	C++/CLI DATA TYPE	F# DATA TYPE
	String	An immutable, fixed-length string of Unicode characters.	**String**	**string**	**String^**	**string**

In addition to the base data types, the System namespace contains over 100 classes, ranging from classes that handle exceptions to classes that deal with core runtime concepts, such as application domains and the garbage collector. The System namespace also contains many second-level namespaces.

For more information about namespaces, use the .NET API Browser to browse the .NET Class Library. The API reference documentation provides documentation on each namespace, its types, and each of their members.

See also

- Common Type System
- .NET API Browser
- Overview

Working with Base Types in .NET

5/2/2018 • 2 minutes to read • Edit Online

This section describes .NET base type operations, including formatting, conversion, and common operations.

In This Section

Type Conversion in the .NET Framework

Describes how to convert from one type to another.

Formatting Types

Describes how to format strings using the string format specifiers.

Manipulating Strings

Describes how to manipulate and format strings.

Parsing Strings

Describes how to convert strings into .NET Framework types.

Related Sections

Common Type System

Describes types used by the .NET Framework.

Dates, Times, and Time Zones

Describes how to work with time zones and time zone conversions in time zone-aware applications.

.NET class libraries

4/28/2019 • 3 minutes to read • Edit Online

Class libraries are the shared library concept for .NET. They enable you to componentize useful functionality into modules that can be used by multiple applications. They can also be used as a means of loading functionality that is not needed or not known at application startup. Class libraries are described using the .NET Assembly file format.

There are three types of class libraries that you can use:

- **Platform-specific** class libraries have access to all the APIs in a given platform (for example, .NET Framework, Xamarin iOS), but can only be used by apps and libraries that target that platform.
- **Portable** class libraries have access to a subset of APIs, and can be used by apps and libraries that target multiple platforms.
- **.NET Standard** class libraries are a merger of the platform-specific and portable library concept into a single model that provides the best of both.

Platform-specific class libraries

Platform-specific libraries are bound to a single .NET implementation (for example, .NET Framework on Windows) and can therefore take significant dependencies on a known execution environment. Such an environment will expose a known set of APIs (.NET and OS APIs) and will maintain and expose expected state (for example, Windows registry).

Developers who create platform specific libraries can fully exploit the underlying platform. The libraries will only ever run on that given platform, making platform checks or other forms of conditional code unnecessary (modulo single sourcing code for multiple platforms).

Platform-specific libraries have been the primary class library type for the .NET Framework. Even as other .NET implementations emerged, platform-specific libraries remained the dominant library type.

Portable class libraries

Portable libraries are supported on multiple .NET implementations. They can still take dependencies on a known execution environment, however, the environment is a synthetic one that is generated by the intersection of a set of concrete .NET implementations. This means that exposed APIs and platform assumptions are a subset of what would be available to a platform-specific library.

You choose a platform configuration when you create a portable library. These are the set of platforms that you need to support (for example, .NET Framework 4.5+, Windows Phone 8.0+). The more platforms you opt to support, the fewer APIs and fewer platform assumptions you can make, the lowest common denominator. This characteristic can be confusing at first, since people often think "more is better", but find that more supported platforms results in fewer available APIs.

Many library developers have switched from producing multiple platform-specific libraries from one source (using conditional compilation directives) to portable libraries. There are several approaches for accessing platform-specific functionality within portable libraries, with bait-and-switch being the most widely accepted technique at this point.

.NET Standard class libraries

.NET Standard libraries are a replacement of the platform-specific and portable libraries concepts. They are

platform-specific in the sense that they expose all functionality from the underlying platform (no synthetic platforms or platform intersections). They are portable in the sense that they work on all supporting platforms.

The .NET Standard exposes a set of library *contracts*. .NET implementations must support each contract fully or not at all. Each implementation, therefore, supports a set of .NET Standard contracts. The corollary is that each .NET Standard class library is supported on the platforms that support its contract dependencies.

The .NET Standard does not expose the entire functionality of the .NET Framework (nor is that a goal), however, they do expose many more APIs than Portable Class Libraries. More APIs will be added over time.

The following platforms support .NET Standard libraries:

- .NET Core
- .NET Framework
- Mono
- Xamarin.iOS, Xamarin.Mac, Xamarin.Android
- Universal Windows Platform (UWP)
- Windows
- Windows Phone
- Windows Phone Silverlight

For more information, see the .NET Standard topic.

Mono class libraries

Class libraries are supported on Mono, including the three types of libraries described above. Mono has often been seen (correctly) as a cross-platform implementation of the Microsoft .NET Framework. In part, this was because platform-specific .NET Framework libraries could run on the Mono runtime without modification or recompilation. This characteristic was in place before the creation of portable class libraries, so was an obvious choice to enable binary portability between the .NET Framework and Mono (although it only worked in one direction).

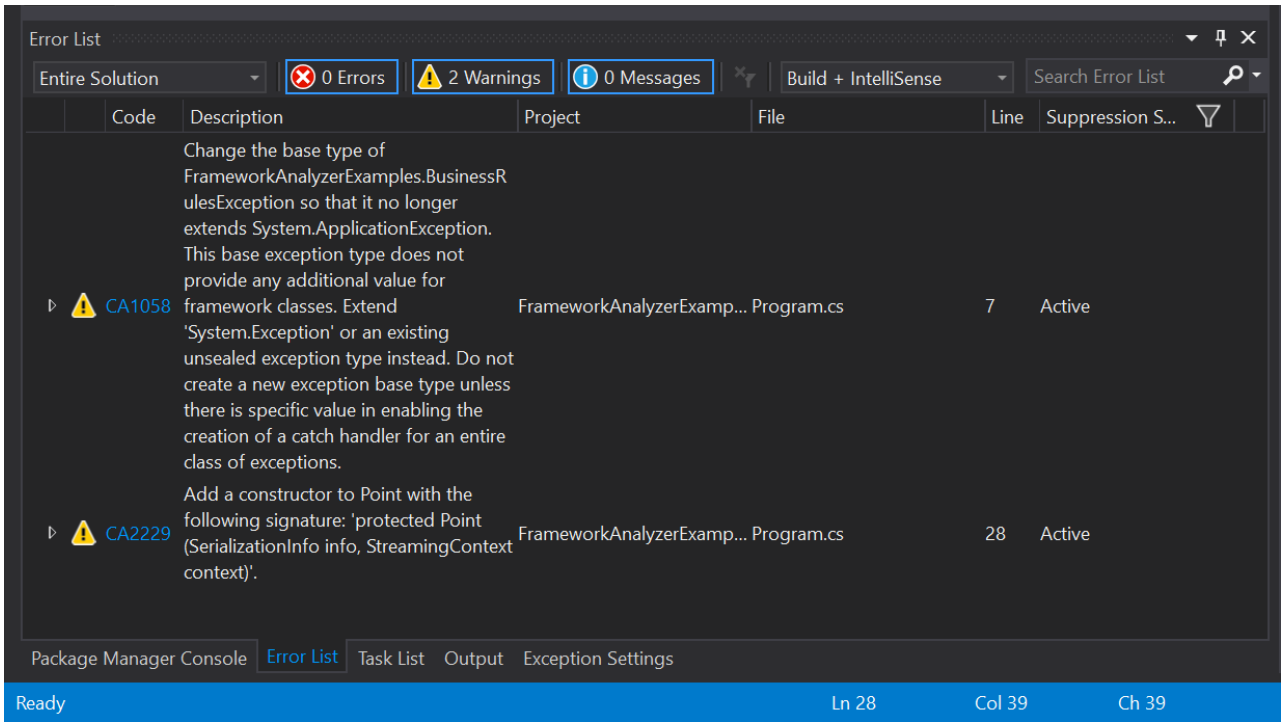
The Roslyn based Analyzers

3/29/2019 • 2 minutes to read • Edit Online

Roslyn-based analyzers use the .NET Compiler SDK (Roslyn APIs) to analyze your project's source code to find issues and suggest corrections. Different analyzers look for different classes of issues, ranging from practices that are likely to cause bugs to security concerns to API compatibility.

Roslyn-based analyzers work both interactively and during builds. The analyzer provides different guidance when in Visual Studio or in command-line builds.

While you edit code in Visual Studio, the analyzers run as you make changes, catching possible issues as soon as you create code that trigger concerns. Any issues are highlighted with squiggles under any issue. Visual Studio displays a light bulb, and when you click on it the analyzer will suggest possible fixes for that issue. When you build the project, either in Visual Studio or from the command line, all the source code is analyzed and the analyzer provides a full list of potential issues. The following figure shows one example.



Roslyn-based analyzers report potential issues as errors, warnings, or information based on the severity of the issue.

You install Roslyn-based analyzers as NuGet packages in your project. The configured analyzers and any settings for each analyzer are restored and run on any developer's machine for that project.

> **NOTE**
>
> The user experience for Roslyn-based analyzers is different than that of the Code Analysis libraries like the older versions of FxCop and the security analysis tools. You don't need to explicitly run the Roslyn-based analyzers. There's no need to use the "Run Code Analysis" menu items on the "Analyze" menu in Visual Studio. Roslyn-based analyzers run asynchronously as you work.

More information on specific analyzers

The following analyzers are covered in this section:

- API Analyzer: This analyzer examines your code for potential compatibility risks or uses of deprecated APIs.
- Framework Analyzer: This analyzer examines your code to ensure it follows the guidelines for .NET Framework applications. These rules include several security-based recommendations.
- .NET Portability Analyzer: This analyzer examines your code to see how much work is required to make your application compatible with other .NET implementations and profiles, including .NET Core, .NET Standard, UWP, and Xamarin for iOS, Android, and Mac.

.NET API analyzer

5/6/2019 • 4 minutes to read • Edit Online

The .NET API Analyzer is a Roslyn analyzer that discovers potential compatibility risks for C# APIs on different platforms and detects calls to deprecated APIs. It can be useful for all C# developers at any stage of development.

API Analyzer comes as a NuGet package Microsoft.DotNet.Analyzers.Compatibility. After you reference it in a project, it automatically monitors the code and indicates problematic API usage. You can also get suggestions on possible fixes by clicking on the light bulb. The drop-down menu includes an option to suppress the warnings.

> **NOTE**
> The .NET API analyzer is still a pre-release version.

Prerequisites

- Visual Studio 2017 and later versions, or Visual Studio for Mac (all versions).

Discovering deprecated APIs

What are deprecated APIs?

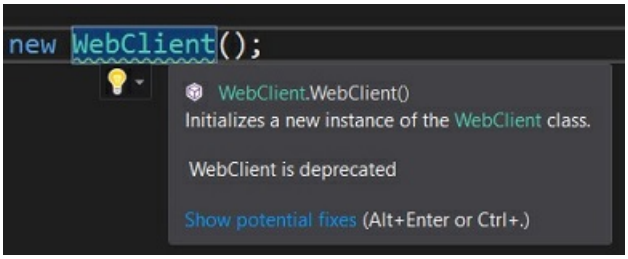
The .NET family is a set of large products that are constantly upgraded to better serve customer needs. It's natural to deprecate some APIs and replace them with new ones. An API is considered deprecated when a better alternative exists. One way to inform that an API is deprecated and shouldn't be used is to mark it with the ObsoleteAttribute attribute. The disadvantage of this approach is that there is only one diagnostic ID for all obsolete APIs (for C#, CS0612). This means that:

- It's impossible to have dedicated documents for each case.
- It's impossible to suppress certain category of warnings. You can suppress either all or none of them.
- To inform users of a new deprecation, a referenced assembly or targeting package has to be updated.

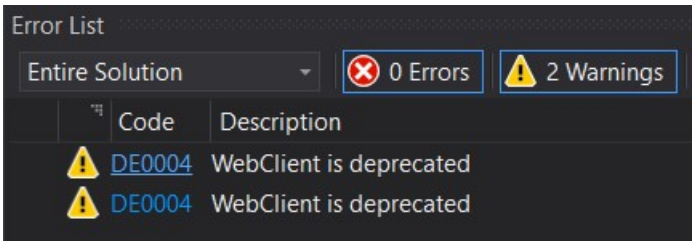
The API Analyzer uses API-specific error codes that begin with DE (which stands for Deprecation Error), which allows control over the display of individual warnings. The deprecated APIs identified by the analyzer are defined in the dotnet/platform-compat repo.

Using the API Analyzer

When a deprecated API, such as WebClient, is used in a code, API Analyzer highlights it with a green squiggly line. When you hover over the API call, a light bulb is displayed with information about the API deprecation, as in the following example:



The **Error List** window contains warnings with a unique ID per deprecated API, as shown in the following example (`DE004`):



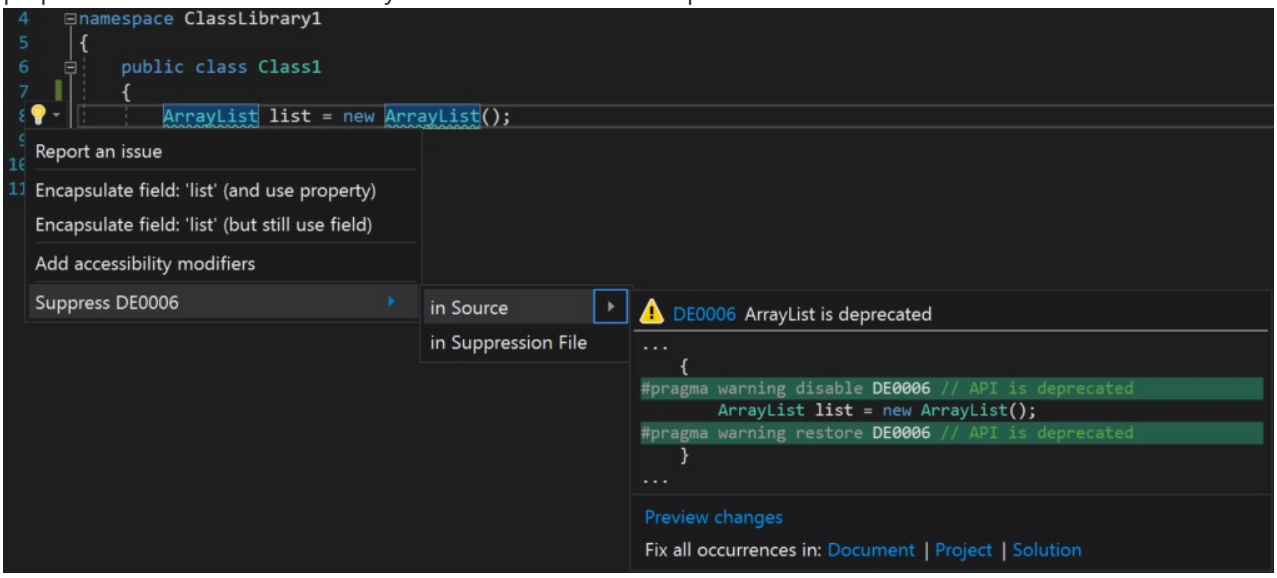
By clicking on the ID, you go to a webpage with detailed information about why the API was deprecated and suggestions regarding alternative APIs that can be used.

Any warnings can be suppressed by right-clicking on the highlighted member and selecting **Suppress <diagnostic ID>**. There are two ways to suppress warnings:

- locally (in source)
- globally (in a suppression file) - recommended

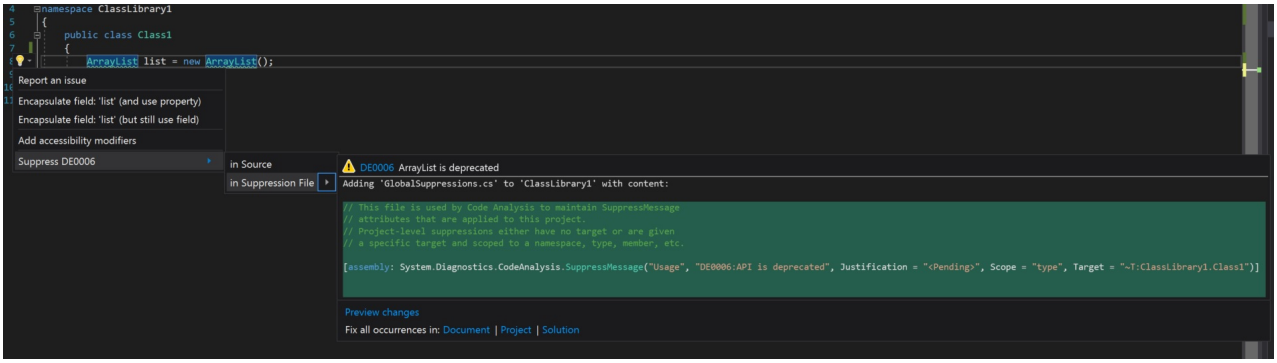
Suppressing warnings locally

To suppress warnings locally, right-click on the member you want to suppress warnings for and then select **Quick Actions and Refactorings** > **Suppress *diagnostic ID*<diagnostic ID>** > **in Source**. The #pragma warning preprocessor directive is added to your source code in the scope defined:

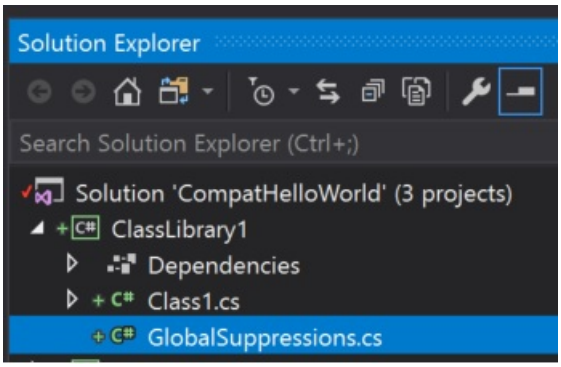


Suppressing warnings globally

To suppress warnings globally, right-click on the member you want to suppress warnings for and then select **Quick Actions and Refactorings** > **Suppress *diagnostic ID*<diagnostic ID>** > **in Suppression File**.



A *GlobalSuppressions.cs* file is added to your project after the first suppression. New global suppressions are appended to this file.



Global suppression is the recommended way to ensure consistency of API usage across projects.

Discovering cross-platform issues

Similar to deprecated APIs, the analyzer identifies all APIs that are not cross-platform. For example, Console.WindowWidth works on Windows but not on Linux and macOS. The diagnostic ID is shown in the **Error List** window. You can suppress that warning by right-clicking and selecting **Quick Actions and Refactorings**. Unlike deprecation cases where you have two options (either keep using the deprecated member and suppress warnings or not use it at all), here if you're developing your code only for certain platforms, you can suppress all warnings for all other platforms you don't plan to run your code on. To do so, you just need to edit your project file and add the `PlatformCompatIgnore` property that lists all platforms to be ignored. The accepted values are: `Linux`, `macOS`, and `Windows`.

```
<PropertyGroup>
    <PlatformCompatIgnore>Linux;macOS</PlatformCompatIgnore>
</PropertyGroup>
```

If your code targets multiple platforms and you want to take advantage of an API not supported on some of them, you can guard that part of the code with an `if` statement:

```
if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
{
     var w = Console.WindowWidth;
     // More code
}
```

You can also conditionally compile per target framework/operating system, but you currently need to do that manually.

Supported diagnostics

Currently, the analyzer handles the following cases:

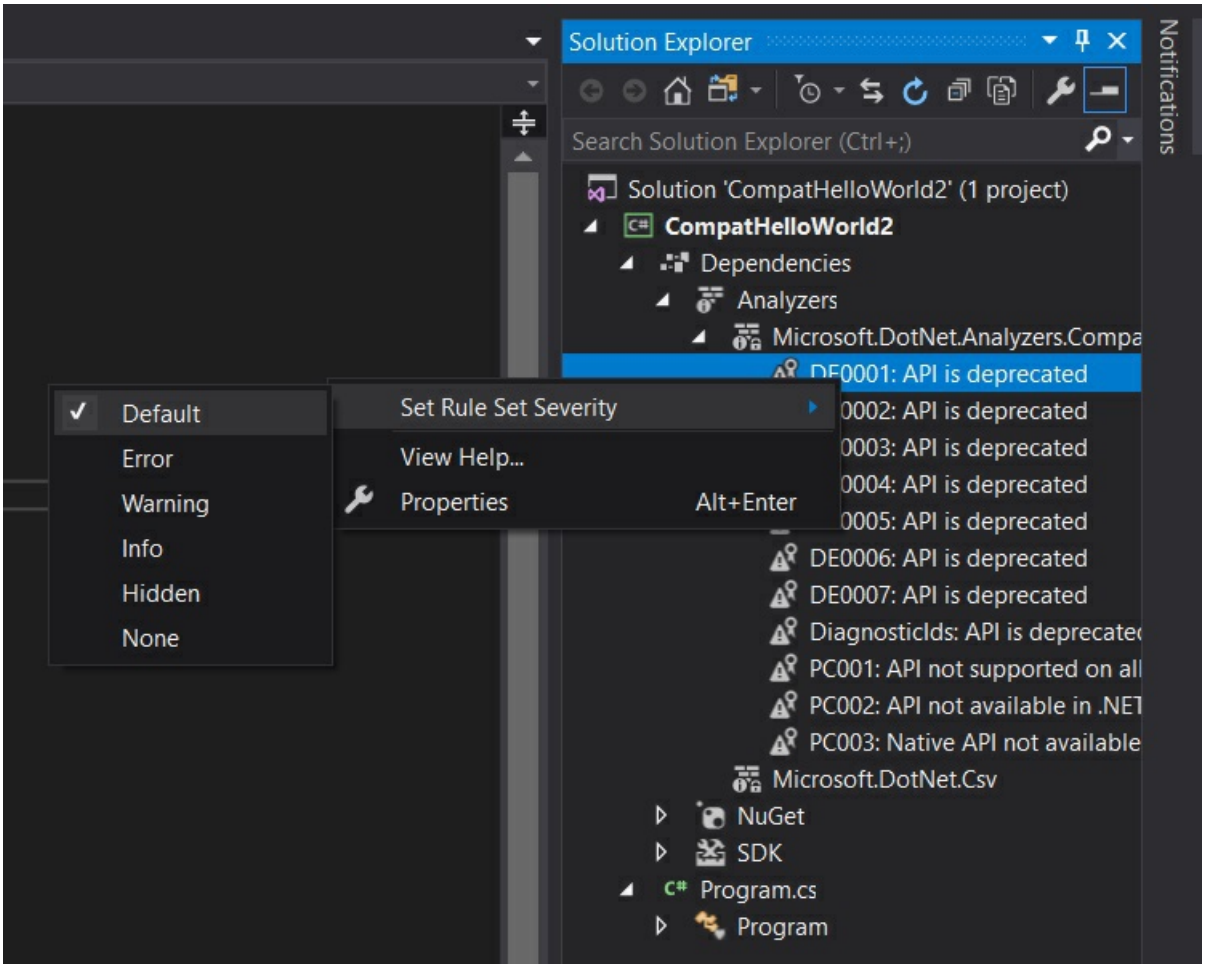
- Usage of a .NET Standard API that throws PlatformNotSupportedException (PC001).
- Usage of a .NET Standard API that isn't available on the .NET Framework 4.6.1 (PC002).
- Usage of a native API that doesn't exist in UWP (PC003).
- Usage of Delegate.BeginInvoke and EndInvoke APIs (PC004).
- Usage of an API that is marked as deprecated (DEXXXX).

CI machine

All these diagnostics are available not only in the IDE, but also on the command line as part of building your project, which includes the CI server.

Configuration

The user decides how the diagnostics should be treated: as warnings, errors, suggestions, or to be turned off. For example, as an architect, you can decide that compatibility issues should be treated as errors, calls to some deprecated APIs generate warnings, while others only generate suggestions. You can configure this separately by diagnostic ID and by project. To do so in **Solution Explorer**, navigate to the **Dependencies** node under your project. Expand the nodes **Dependencies** > **Analyzers** > **Microsoft.DotNet.Analyzers.Compatibility**. Right click on the diagnostic ID, select **Set Rule Set Severity** and pick the desired option.



See also

- Introducing API Analyzer blog post.
- API Analyzer demo video on YouTube.

The .NET Portability Analyzer

7/24/2019 • 4 minutes to read • Edit Online

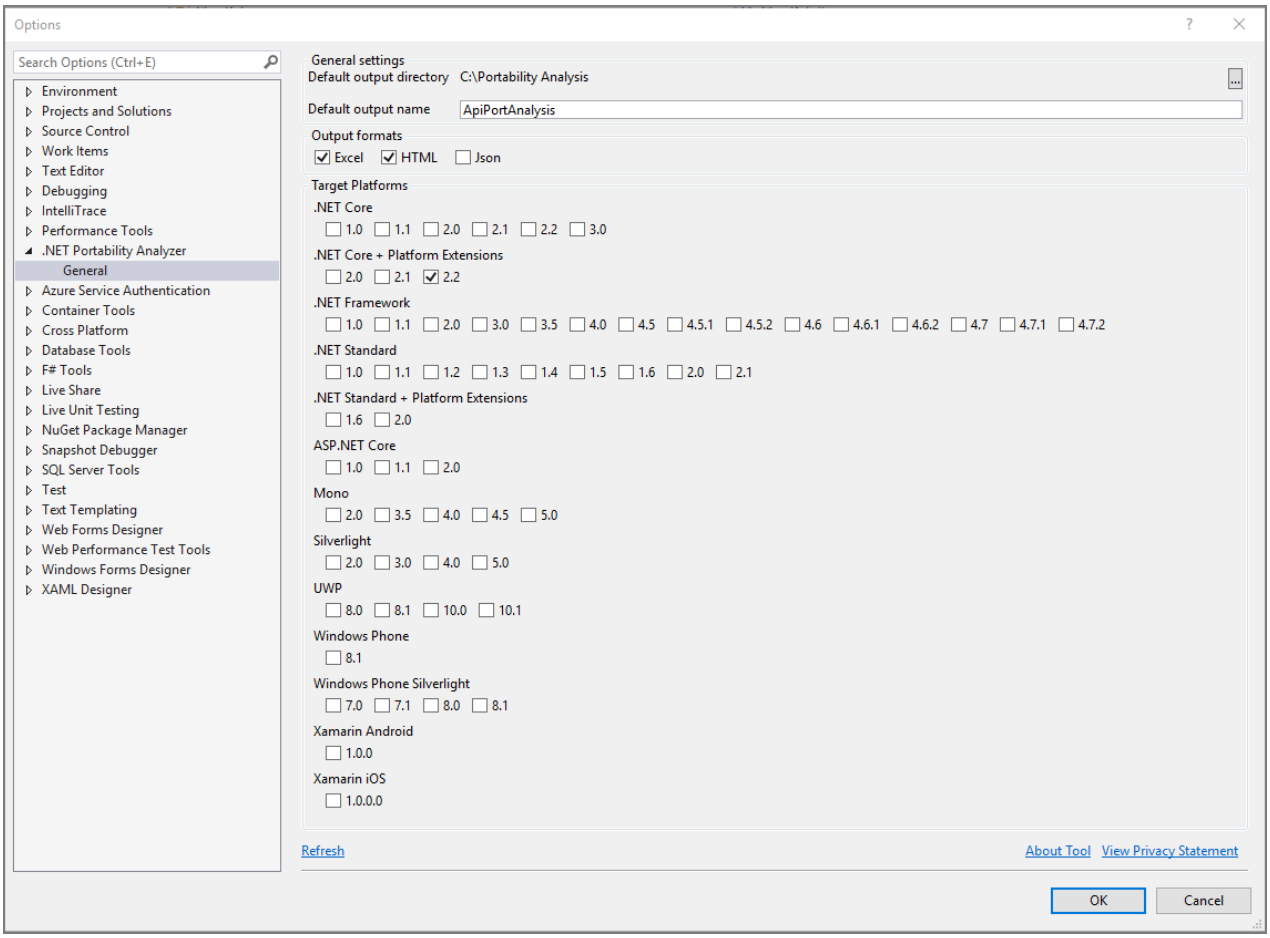
Want to make your libraries support multi-platform? Want to see how much work is required to make your application compatible with other .NET implementations and profiles, including .NET Core, .NET Standard, UWP, and Xamarin for iOS, Android, and Mac? The .NET Portability Analyzer is a tool that provides you with a detailed report on how flexible your program is across .NET implementations by analyzing assemblies. The Portability Analyzer is offered as a Visual Studio Extension, which analyzes assembly per project, and as a ApiPort console app, which analyzes assemblies by specified files or directory.

Common targets

- .NET Core: Has a modular design, employs side-by-side, and targets cross-platform scenarios. Side-by-side allows you to adopt new .NET Core versions without breaking other apps. If your goal is to port your app to .NET Core supporting cross-platforms, this is the recommended target.
- .NET Standard: Includes the .NET Standard APIs available on all .NET implementations. If your goal is to make your library to run on all .NET supported platforms, this is recommended target.
- ASP.NET Core: A modern web-framework built on .NET Core. If your goal is to port your web app to .NET Core to support multiple platforms, this is the recommended target.
- .NET Core + Platform Extensions: Includes the .NET Core APIs in addition to the Windows Compatibility Pack, which provides many of the .NET Framework available technologies. This is a recommended target for porting your app from .NET Framework to .NET Core on Windows.
- .NET Standard + Platform Extensions: Includes the .NET Standard APIs in addition to the Windows Compatibility Pack, which provides many of the .NET Framework available technologies. This is a recommended target for porting your library from .NET Framework to .NET Core on Windows.

How to use the .NET Portability Analyzer

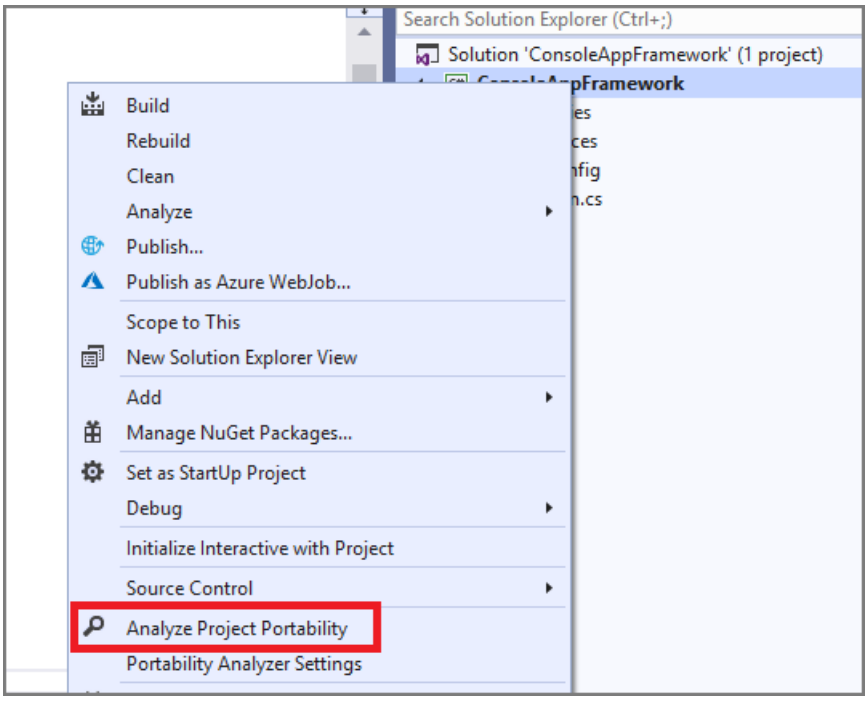
To begin using the .NET Portability Analyzer in Visual Studio, you first need to download and install the extension from the Visual Studio Marketplace. It works on Visual Studio 2017 and later versions. You can configure it in Visual Studio via **Analyze** > **Portability Analyzer Settings** and select your Target Platforms, which is the .NET platforms/versions that you want to evaluate the portability gaps comparing with the platform/version that your current assembly is built with.



You can also use the ApiPort console application, download it from ApiPort repository. You can use `listTargets` command option to display the available target list, then pick target platforms by specifying `-t` or `--target` command option.

Analyze portability

To analyze your entire project in Visual Studio, right-click on your project in **Solution Explorer** and select **Analyze Assembly Portability**. Otherwise, go to the **Analyze** menu and select **Analyze Assembly Portability**. From there, select your project's executable or DLL.



You can also use the ApiPort console app.

- Type the following command to analyze the current directory: `ApiPort.exe analyze -f .`
- To analyze a specific list of .dll files, type the following command:
  `ApiPort.exe analyze -f first.dll -f second.dll -f third.dll`
- Run `ApiPort.exe -?` to get more help

It is recommended that you include all the related exe and dll files that you own and want to port, and exclude the files that your app depends on, but you don't own and can't port. This will give you most relevant portability report.

View and interpret portability result

Only APIs that are unsupported by a Target Platform appear in the report. After running the analysis in Visual Studio, you'll see your .NET Portability report file link pops up. If you used the ApiPort console app, your .NET Portability report is saved as a file in the format you specified. The default is in an Excel file (*.xlsx*) in your current directory.

Portability Summary

Submission Id	5552b6d0-b7ed-4fa8-9a3c-e3d8099c7ab1				
Description					
Targets	.NET Core + Platform Extensions,.NET Core,.NET Framework,.NET Standard + Platform Extensions				
Assembly Name	**Target Framework**	**.NET Core + Platfor**	**.NET Core**	**.NET Framewor**	**.NET Standard**
svcutil	.NETFramework,Version=v4.5	71.24	71.48	100	71.24
API Catalog last updated on	Tuesday, March 5, 2019				
See 'http://go.microsoft.com/fwlink/?LinkId=397652' to learn how to read this table					

Portability Summary | Details

The Portability Summary section of the report shows the portability percentage for each assembly included in the run. In the previous example, 71.24% of the .NET Framework APIs used in the `svcutil` app are available in .NET Core + Platform Extensions. If you run the .NET Portability Analyzer tool against multiple assemblies, each assembly should have a row in the Portability Summary report.

Details

Target type	Target member	Assembly name	.NET Core + Platform Extensions	Recommended
T:System.AppDomain	M:System.AppDomain.get_SetupInformation	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	T:System.AppDomainSetup	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	M:System.AppDomainSetup.get_ConfigurationFile	svcutil	Not supported	Remove usage.
T:System.Data.DataSetSchemaImporterExtension	T:System.Data.DataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtension	T:System.Data.Design.TypedDataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx3!	T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx:	svcutil	Not supported	

Portability Summary | Details

The Details section of the report lists the APIs missing from one of the Target Platforms.

- Target type: the type has missing API from a Target Platform
- Target member: the method is missing from a Target Platform
- Assembly name: the .NET Framework assembly that the missing API lives in.
- Each of the selected Target Platforms is one column, such as ".NET Core": "Not supported" value means the API is not supported on this Target Platform.
- Recommended Changes: recommended API or technology to change to. Currently, this field is empty or out of date for a lot of APIs. Due to the large number of APIs, we have big challenge to keep it up. We are looking at alternate solutions to provide helpful information to customers.

Missing Assemblies

Header for assembly name entries	Used By	Reason
Autofac, Version=4.6.2.0, Culture=neutral, PublicKeyToken=17863af14b0044da	ApiPort	Unresolved assembly
System.CommandLine, Version=0.1.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60	ApiPort	Unresolved assembly

Portability Summary | Details | Missing assemblies

You may find a Missing Assemblies section in your report. It tells you that this list of assemblies are referenced by your analyzed assemblies and were not analyzed. If it's an assembly that you own, include it in the Api portability analyzer run so that you can get API level detailed portability report for it. If it's third party library, looks for if they

have newer version supporting your target platform. If so, consider moving to the newer version. Eventually, you would expect this list includes all the third party assemblies that your app depends on and confirmed that they have a version supporting your target platform.

For more information on the .NET Portability Analyzer, visit the GitHub documentation and A Brief Look at the .NET Portability Analyzer Channel 9 video.

The .NET Framework Analyzer

4/8/2019 • 4 minutes to read • Edit Online

You can use the .NET Framework Analyzer to find potential issues in your .NET Framework-based application code. This analyzer finds potential issues and suggests fixes to them.

The analyzer runs interactively in Visual Studio as you write your code or as part of a CI build. You should add the analyzer to your project as early as possible in your development. The sooner you find any potential issues in your code, the easier they are to fix. However, you can add it at any time in the development cycle. It finds any issues with the existing code and warns about new issues as you keep developing.

Installing and configuring the .NET Framework Analyzer

The .NET Security Analyzers must be installed as a NuGet package on every project where you want them to run. Only one developer needs to add them to the project. The analyzer package is a project dependency and will run on every developer's machine once it has the updated solution.

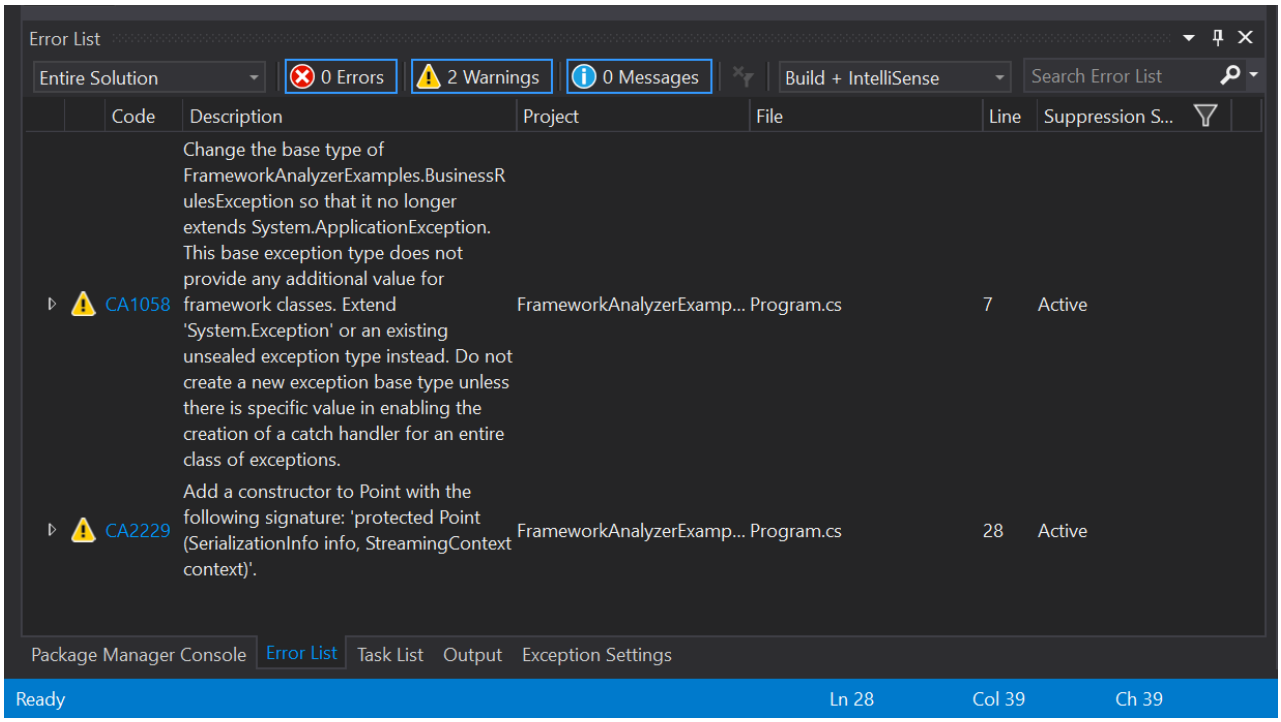
The .NET Framework Analyzer is delivered in the Microsoft.NetFramework.Analyzers NuGet package. This package provides only the analyzers specific to the .NET Framework, which includes security analyzers. In most cases, you'll want the Microsoft.CodeAnalysis.FxCopAnalyzers NuGet package. The FxCopAnalyzers aggregate package contains all the framework analyzers included in the Framework.Analyzers package as well as the following analyzers:

- Microsoft.CodeQuality.Analyzers: Provides general guidance and guidance for .NET Standard APIs
- Microsoft.NetCore.Analyzers: Provides analyzers specific to .NET Core APIs.
- Text.Analyzers: Provides guidance for text included as code, including comments.

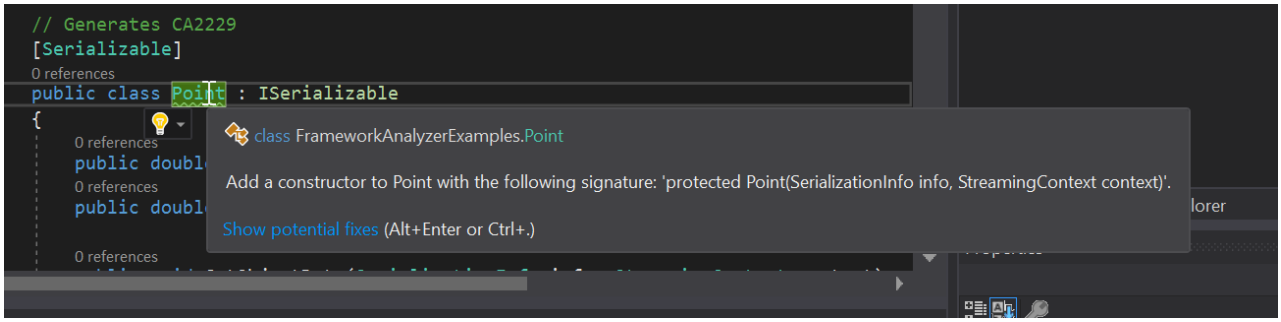
To install it, right-click on the project, and select "Manage Dependencies". From the NuGet explorer, search for "NetFramework Analyzer", or if you prefer, "Fx Cop Analyzer". Install the latest stable version in all projects in your solution.

Using the .NET Framework Analyzer

Once the NuGet package is installed, build your solution. The analyzer will report any issues it locates in your codebase. The issues are reported as warnings in the Visual Studio Error List window, as shown in the following image:



As you write code, you see squiggles underneath any potential issue in your code. Hover over any issue and you see details about the issue, and suggestions for any possible fix, as shown in the following image:



The analyzers examine the code in your solution and provide you with a list of warnings for any of these issues:

CA1058: Types should not extend certain base types

There are a small number of types in the .NET Framework that you should not derived from directly.

Category: Design

Severity: Warning

Additional information: CA:1058: Types should not extend certain base types

CA2153: Do not catch corrupted state exceptions

Catching corrupted state exceptions could mask errors (such as access violations), resulting in an inconsistent state of execution or making it easier for attackers to compromise a system. Instead, catch and handle a more specific set of exception type(s) or re-throw the exception

Category: Security

Severity: Warning

Additional information: ## CA2153: Do not catch corrupted state exceptions

CA2229: Implement serialization constructors

The analyzer generates this warning when you create a type that implements the ISerializable interface but does not define the required serialization constructor. To fix a violation of this rule, implement the serialization constructor. For a sealed class, make the constructor private; otherwise, make it protected. The serialization

constructor has the following signature:

```
public class MyItemType
{
    // The special constructor is used to deserialize values.
    public MyItemType(SerializationInfo info, StreamingContext context)
    {
        // implementation removed.
    }
}
```

Category: Usage

Severity: Warning

Additional information: CA2229: Implement serialization constructors

CA2235: Mark all non-serializable fields

An instance field of a type that is not serializable is declared in a type that is serializable. You must explicitly mark that field with the NonSerializedAttribute to fix this warning.

Category: Usage

Severity: Warning

Additional information: CA2235: Mark all non-serializable fields

CA2237: Mark ISerializable types with serializable

To be recognized by the common language runtime as serializable, types must be marked by using the SerializableAttribute attribute even when the type uses a custom serialization routine by implementing the ISerializable interface.

Category: Usage

Severity: Warning

Additional information: CA2237: Mark ISerializable types with serializable

CA3075: Insecure DTD processing in XML

If you use insecure DtdProcessing instances or reference external entity sources, the parser may accept untrusted input and disclose sensitive information to attackers.

Category: Security

Severity: Warning

Additional information: A3075: Insecure DTD processing in XML

CA5350: Do not use weak cryptographic algorithms

Cryptographic algorithms degrade over time as attacks become more advanced. Depending on the type and application of this cryptographic algorithm, further degradation of its cryptographic strength may allow attackers to read enciphered messages, tamper with enciphered messages, forge digital signatures, tamper with hashed content, or otherwise compromise any cryptosystem based on this algorithm. For encryption, use an AES algorithm (AES-256, AES-192 and AES-128 are acceptable) with a key length greater than or equal to 128 bits. For hashing, use a hashing function in the SHA-2 family, such as SHA-2 512, SHA-2 384, or SHA-2 256.

Category: Security

Severity: Warning

Additional information: CA5350: Do not use weak cryptographic algorithms

CA5351: Do not use broken cryptographic algorithms

An attack making it computationally feasible to break this algorithm exists. This allows attackers to break the cryptographic guarantees it is designed to provide. Depending on the type and application of this cryptographic algorithm, this may allow attackers to read enciphered messages, tamper with enciphered messages, forge digital signatures, tamper with hashed content, or otherwise compromise any cryptosystem based on this algorithm. For encryption, use an AES algorithm (AES-256, AES-192 and AES-128 are acceptable) with a key length greater than or equal to 128 bits. For hashing, use a hashing function in the SHA-2 family, such as SHA512, SHA384, or SHA256. For digital signatures, use RSA with a key length greater than or equal to 2048-bits, or ECDSA with a key length greater than or equal to 256 bits.

Category: Security

Severity: Warning

Additional Information: CA5351: Do not use broken cryptographic algorithms

Handling and throwing exceptions in .NET

2/6/2019 • 2 minutes to read • Edit Online

Applications must be able to handle errors that occur during execution in a consistent manner. .NET provides a model for notifying applications of errors in a uniform way: .NET operations indicate failure by throwing exceptions.

Exceptions

An exception is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be thrown because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions that the runtime encounters (such as code that can't be verified), and so on. Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you can't recover from most runtime exceptions.

In .NET, an exception is an object that inherits from the System.Exception class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

Exceptions vs. traditional error-handling methods

Traditionally, a language's error-handling model relied on either the language's unique way of detecting errors and locating handlers for them, or on the error-handling mechanism provided by the operating system. The way .NET implements exception handling provides the following advantages:

- Exception throwing and handling works the same for .NET programming languages.
- Doesn't require any particular language syntax for handling exceptions, but allows each language to define its own syntax.
- Exceptions can be thrown across process and even machine boundaries.
- Exception-handling code can be added to an application to increase program reliability.

Exceptions offer advantages over other methods of error notification, such as return codes. Failures don't go unnoticed because if an exception is thrown and you don't handle it, the runtime terminates your application. Invalid values don't continue to propagate through the system as a result of code that fails to check for a failure return code.

Common exceptions

The following table lists some common exceptions with examples of what can cause them.

EXCEPTION TYPE	DESCRIPTION	EXAMPLE
Exception	Base class for all exceptions.	None (use a derived class of this exception).
IndexOutOfRangeException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside its valid range: `arr[arr.Length+1]`

EXCEPTION TYPE	DESCRIPTION	EXAMPLE
NullReferenceException	Thrown by the runtime only when a null object is referenced.	`object o = null;` `o.ToString();`
InvalidOperationException	Thrown by methods when in an invalid state.	Calling `Enumerator.MoveNext()` after removing an item from the underlying collection.
ArgumentException	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	Thrown by methods that do not allow an argument to be null.	`String s = null;` `"Calculate".IndexOf(s);`
ArgumentOutOfRangeException	Thrown by methods that verify that arguments are in a given range.	`String s = "string";` `s.Substring(s.Length+1);`

See also

- Exception Class and Properties
- How to: Use the Try-Catch Block to Catch Exceptions
- How to: Use Specific Exceptions in a Catch Block
- How to: Explicitly Throw Exceptions
- How to: Create User-Defined Exceptions
- Using User-Filtered Exception Handlers
- How to: Use Finally Blocks
- Handling COM Interop Exceptions
- Best Practices for Exceptions
- What Every Dev needs to Know About Exceptions in the Runtime

Assemblies in .NET

8/21/2019 • 4 minutes to read • Edit Online

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

In .NET Core and .NET Framework, an assembly can be built from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This allows larger projects to be planned in such a way that several individual developers work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see How to: Build a Multifile Assembly.

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- For libraries that target the .NET Framework, you can share an assembly between applications by putting it in the global assembly cache. Assemblies must be strong-named before they can be included in the global assembly cache. For more information, see Strong-Named Assemblies.
- Assemblies are only loaded into memory if they are required. If they are not used, they are not loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see Reflection (C#) or Reflection (Visual Basic).
- You can load an assembly only to inspect by using the MetadataLoadContext class.

Assembly manifest

Within every assembly is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, such as other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- An *assembly reference list*, which is a list of all external dependencies — .dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects are available to all other applications. In .NET Core, they are coupled with a particular .NET Core runtime. In .NET Framework, they reside in the global assembly cache. The System.IO namespace is an example of an assembly in the global assembly cache. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications that use them need not rely on Windows registry values to function properly. Assemblies reduce .dll conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer. For more information, see Assembly Manifest.

Adding a reference to an assembly

To use an assembly, you must add a reference to it. Next, you can use the using directive for C# or Imports statement for Visual Basic to choose the namespace of the items you want to use. Once an assembly is referenced and imported, all the accessible types, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

> **NOTE**
>
> Most assemblies from the .NET Class Library are referenced automatically. In some cases, though, a system assembly may not automatically be referenced. In .NET Core, you can add a reference to the NuGet package that contains the assembly either by using NuGet Package Manager in Visual Studio or by adding a <PackageReference> element for the assembly to the *.csproj or *.vbproj project. In .NET Framework, you can add a reference to the assembly by using the **Add Reference** dialog in Visual Studio or by using the `-reference` command line option for the C# or Visual Basic compilers.

In C#, you can also use two versions of the same assembly in a single application. For more information, see extern alias.

Creating an assembly

Compile your application by building it in Visual Studio, by building it from the command line by using .NET Core command-line interface (CLI) tools, or by building .NET Framework assemblies with a command-line compiler. For more information about building assemblies using .NET CLI tools, see .NET Core command-line interface (CLI) tools. For building assemblies with the command-line compilers, see Command-line build with csc.exe for C# and Building from the Command Line for Visual Basic.

> **NOTE**
>
> To build an assembly in Visual Studio, on the **Build** menu choose **Build**.

See also

- .NET assembly file format
- Assemblies in the Common Language Runtime
- Friend Assemblies
- How to: Load and Unload Assemblies (C#)
- How to: Load and Unload Assemblies (Visual Basic)
- How to: Use and Debug Assembly Unloadability in .NET Core
- How to: Determine If a File Is an Assembly (C#)
- How to: Determine If a File Is an Assembly (Visual Basic)

Memory Management and Garbage Collection in .NET

5/2/2018 • 2 minutes to read • Edit Online

This section of the documentation provides information about managing memory in .NET.

In This Section

Cleaning Up Unmanaged Resources
Describes how to properly manage and clean up unmanaged resources..

Garbage Collection
Provides information about the .NET garbage collector.

Related Sections

Development Guide

Generic types overview

3/26/2019 • 2 minutes to read • Edit Online

Developers use generics all the time in .NET, whether implicitly or explicitly. When you use LINQ in .NET, did you ever notice that you're working with IEnumerable<T>? Or if you ever saw an online sample of a "generic repository" for talking to databases using Entity Framework, did you see that most methods return IQueryable<T>? You may have wondered what the **T** is in these examples and why it's in there.

First introduced in the .NET Framework 2.0, **generics** are essentially a "code template" that allows developers to define type-safe data structures without committing to an actual data type. For example, List<T> is a generic collection that can be declared and used with any type, such as `List<int>`, `List<string>`, or `List<Person>`.

To understand why generics are useful, let's take a look at a specific class before and after adding generics: ArrayList. In .NET Framework 1.0, the `ArrayList` elements were of type Object. This meant that any element added was silently converted into an `Object`. The same would happen when reading the elements from the list. This process is known as boxing and unboxing, and it impacts performance. More than that, however, there's no way to determine the type of data in the list at compile time. This makes for some fragile code. Generics solve this problem by defining the type of data each instance of list will contain. For example, you can only add integers to `List<int>` and only add Persons to `List<Person>`.

Generics are also available at runtime. This means the runtime knows what type of data structure you're using and can store it in memory more efficiently.

The following example is a small program that illustrates the efficiency of knowing the data structure type at runtime:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace GenericsExample {
  class Program {
    static void Main(string[] args) {
      //generic list
      List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
      //non-generic list
      ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
      // timer for generic list sort
      Stopwatch s = Stopwatch.StartNew();
      ListGeneric.Sort();
      s.Stop();
      Console.WriteLine($"Generic Sort: {ListGeneric}  \n Time taken: {s.Elapsed.TotalMilliseconds}ms");

      //timer for non-generic list sort
      Stopwatch s2 = Stopwatch.StartNew();
      ListNonGeneric.Sort();
      s2.Stop();
      Console.WriteLine($"Non-Generic Sort: {ListNonGeneric}  \n Time taken:
{s2.Elapsed.TotalMilliseconds}ms");
      Console.ReadLine();
    }
  }
}
```

This program produces output similar to the following:

```
Generic Sort: System.Collections.Generic.List`1[System.Int32]
 Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
 Time taken: 0.2592ms
```

The first thing you can notice here is that sorting the generic list is significantly faster than sorting the non-generic list. You might also notice that the type for the generic list is distinct ([System.Int32]), whereas the type for the non-generic list is generalized. Because the runtime knows the generic `List<int>` is of type Int32, it can store the list elements in an underlying integer array in memory while the non-generic `ArrayList` has to cast each list element to an object. As this example shows, the extra casts take up time and slow down the list sort.

An additional advantage of the runtime knowing the type of your generic is a better debugging experience. When you're debugging a generic in C#, you know what type each element is in your data structure. Without generics, you would have no idea what type each element was.

See also

- C# Programming Guide - Generics

Delegates and lambdas

4/28/2019 • 4 minutes to read • Edit Online

Delegates define a type, which specify a particular method signature. A method (static or instance) that satisfies this signature can be assigned to a variable of that type, then called directly (with the appropriate arguments) or passed as an argument itself to another method and then called. The following example demonstrates delegate use.

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- The `public delegate string Reverse(string s);` line creates a delegate type of a certain signature, in this case a method that takes a string parameter and then returns a string parameter.
- The `static string ReverseString(string s)` method, which has the exact same signature as the defined delegate type, implements the delegate.
- The `Reverse rev = ReverseString;` line shows that you can assign a method to a variable of the corresponding delegate type.
- The `Console.WriteLine(rev("a string"));` line demonstrates how to use a variable of a delegate type to invoke the delegate.

In order to streamline the development process, .NET includes a set of delegate types that programmers can reuse and not have to create new types. These are `Func<>`, `Action<>` and `Predicate<>`, and they can be used in various places throughout the .NET APIs without the need to define new delegate types. Of course, there are some differences between the three as you will see in their signatures which mostly have to do with the way they were meant to be used:

- `Action<>` is used when there is a need to perform an action using the arguments of the delegate.
- `Func<>` is used usually when you have a transformation on hand, that is, you need to transform the arguments of the delegate into a different result. Projections are a prime example of this.
- `Predicate<>` is used when you need to determine if the argument satisfies the condition of the delegate. It can also be written as a `Func<T, bool>`.

We can now take our example above and rewrite it using the `Func<>` delegate instead of a custom type. The program will continue running exactly the same.

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

For this simple example, having a method defined outside of the `Main` method seems a bit superfluous. It is because of this that .NET Framework 2.0 introduced the concept of **anonymous delegates**. With their support you are able to create "inline" delegates without having to specify any additional type or method. You simply inline the definition of the delegate where you need it.

For an example, we are going to switch it up and use our anonymous delegate to filter out a list of only even numbers and then print them to the console.

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
          delegate (int no)
          {
              return (no % 2 == 0);
          }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

As you can see, the body of the delegate is just a set of expressions, as any other delegate. But instead of it being a separate definition, we've introduced it *ad hoc* in our call to the List<T>.FindAll method.

However, even with this approach, there is still much code that we can throw away. This is where **lambda expressions** come into play.

Lambda expressions, or just "lambdas" for short, were introduced first in C# 3.0, as one of the core building blocks of Language Integrated Query (LINQ). They are just a more convenient syntax for using delegates. They declare a

signature and a method body, but don't have an formal identity of their own, unless they are assigned to a delegate. Unlike delegates, they can be directly assigned as the left-hand side of event registration or in various LINQ clauses and methods.

Since a lambda expression is just another way of specifying a delegate, we should be able to rewrite the above sample to use a lambda expression instead of an anonymous delegate.

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(i => i % 2 == 0);

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

In the preceding example, the lambda expression used is `i => i % 2 == 0`. Again, it is just a **very** convenient syntax for using delegates, so what happens under the covers is similar to what happens with the anonymous delegate.

Again, lambdas are just delegates, which means that they can be used as an event handler without any problems, as the following code snippet illustrates.

```
public MainWindow()
{
    InitializeComponent();

    Loaded += (o, e) =>
    {
        this.Title = "Loaded";
    };
}
```

The `+=` operator in this context is used to subscribe to an event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Further reading and resources

- Delegates
- Anonymous Functions
- Lambda expressions

LINQ (Language Integrated Query)

7/16/2019 • 8 minutes to read • Edit Online

What is it?

LINQ provides language-level querying capabilities and a higher-order function API to C# and VB as a way to write expressive, declarative code.

Language-level query syntax:

```
var linqExperts = from p in programmers
                  where p.IsNewToLINQ
                  select new LINQExpert(p);
```

```
Dim linqExperts = From p in programmers
                  Where p.IsNewToLINQ
                  Select New LINQExpert(p)
```

Same example using the `IEnumerable<T>` API:

```
var linqExperts = programmers.Where(p => p.IsNewToLINQ)
                             .Select(p => new LINQExpert(p));
```

```
Dim linqExperts = programmers.Where(Function(p) p.IsNewToLINQ).
                             Select(Function(p) New LINQExpert(p))
```

LINQ is Expressive

Imagine you have a list of pets, but want to convert it into a dictionary where you can access a pet directly by its `RFID` value.

Traditional imperative code:

```
var petLookup = new Dictionary<int, Pet>();

foreach (var pet in pets)
{
    petLookup.Add(pet.RFID, pet);
}
```

```
Dim petLookup = New Dictionary(Of Integer, Pet)()

For Each pet in pets
    petLookup.Add(pet.RFID, pet)
Next
```

The intention behind the code is not to create a new `Dictionary<int, Pet>` and add to it via a loop, it is to convert an existing list into a dictionary! LINQ preserves the intention whereas the imperative code does not.

Equivalent LINQ expression:

```
var petLookup = pets.ToDictionary(pet => pet.RFID);
```

```
Dim petLookup = pets.ToDictionary(Function(pet) pet.RFID)
```

The code using LINQ is valuable because it evens the playing field between intent and code when reasoning as a programmer. Another bonus is code brevity. Imagine reducing large portions of a codebase by 1/3 as done above. Pretty sweet deal, right?

LINQ Providers Simplify Data Access

For a significant chunk of software out in the wild, everything revolves around dealing with data from some source (Databases, JSON, XML, etc). Often this involves learning a new API for each data source, which can be annoying. LINQ simplifies this by abstracting common elements of data access into a query syntax which looks the same no matter which data source you pick.

Consider the following: finding all XML elements with a specific attribute value.

```
public static IEnumerable<XElement> FindAllElementsWithAttribute(XElement documentRoot, string elementName,
                                        string attributeName, string value)
{
    return from el in documentRoot.Elements(elementName)
           where (string)el.Element(attributeName) == value
           select el;
}
```

```
Public Shared Function FindAllElementsWithAttribute(documentRoot As XElement, elementName As String,
                                        attributeName As String, value As String) As IEnumerable(Of
XElement)
    Return From el In documentRoot.Elements(elementName)
           Where el.Element(attributeName).ToString() = value
           Select el
End Function
```

Writing code to manually traverse the XML document to perform this task would be far more challenging.

Interacting with XML isn't the only thing you can do with LINQ Providers. Linq to SQL is a fairly bare-bones Object-Relational Mapper (ORM) for an MSSQL Server Database. The JSON.NET library provides efficient JSON Document traversal via LINQ. Furthermore, if there isn't a library which does what you need, you can also write your own LINQ Provider!

Why Use the Query Syntax?

This is a question which often comes up. After all, this,

```
var filteredItems = myItems.Where(item => item.Foo);
```

```
Dim filteredItems = myItems.Where(Function(item) item.Foo)
```

is a lot more concise than this:

```
var filteredItems = from item in myItems
                    where item.Foo
                    select item;
```

```
Dim filteredItems = From item In myItems
                    Where item.Foo
                    Select item
```

Isn't the API syntax just a more concise way to do the query syntax?

No. The query syntax allows for the use of the **let** clause, which allows you to introduce and bind a variable within the scope of the expression, using it in subsequent pieces of the expression. Reproducing the same code with only the API syntax can be done, but will most likely lead to code which is hard to read.

So this begs the question, **should you just use the query syntax?**

The answer to this question is **yes** if...

- Your existing codebase already uses the query syntax
- You need to scope variables within your queries due to complexity
- You prefer the query syntax and it won't distract from your codebase

The answer to this question is **no** if...

- Your existing codebase already uses the API syntax
- You have no need to scope variables within your queries
- You prefer the API syntax and it won't distract from your codebase

Essential Samples

For a truly comprehensive list of LINQ samples, visit 101 LINQ Samples.

The following is a quick demonstration of some of the essential pieces of LINQ. This is in no way comprehensive, as LINQ provides significantly more functionality than what is showcased here.

- The bread and butter - `Where`, `Select`, and `Aggregate`:

```
// Filtering a list.
var germanShepards = dogs.Where(dog => dog.Breed == DogBreed.GermanShepard);

// Using the query syntax.
var queryGermanShepards = from dog in dogs
                          where dog.Breed == DogBreed.GermanShepard
                          select dog;

// Mapping a list from type A to type B.
var cats = dogs.Select(dog => dog.TurnIntoACat());

// Using the query syntax.
var queryCats = from dog in dogs
                select dog.TurnIntoACat();

// Summing the lengths of a set of strings.
int seed = 0;
int sumOfStrings = strings.Aggregate(seed, (s1, s2) => s1.Length + s2.Length);
```

```
' Filtering a list.
Dim germanShepards = dogs.Where(Function(dog) dog.Breed = DogBreed.GermanShepard)

' Using the query syntax.
Dim queryGermanShepards = From dog In dogs
                          Where dog.Breed = DogBreed.GermanShepard
                          Select dog

' Mapping a list from type A to type B.
Dim cats = dogs.Select(Function(dog) dog.TurnIntoACat())

' Using the query syntax.
Dim queryCats = From dog In dogs
                Select dog.TurnIntoACat()

' Summing the lengths of a set of strings.
Dim seed As Integer = 0
Dim sumOfStrings As Integer = strings.Aggregate(seed, Function(s1, s2) s1.Length + s2.Length)
```

- Flattening a list of lists:

```
// Transforms the list of kennels into a list of all their dogs.
var allDogsFromKennels = kennels.SelectMany(kennel => kennel.Dogs);
```

```
' Transforms the list of kennels into a list of all their dogs.
Dim allDogsFromKennels = kennels.SelectMany(Function(kennel) kennel.Dogs)
```

- Union between two sets (with custom comparator):

```
public class DogHairLengthComparer : IEqualityComparer<Dog>
{
    public bool Equals(Dog a, Dog b)
    {
        if (a == null && b == null)
        {
            return true;
        }
        else if ((a == null && b != null) ||
                 (a != null && b == null))
        {
            return false;
        }
        else
        {
            return a.HairLengthType == b.HairLengthType;
        }
    }

    public int GetHashCode(Dog d)
    {
        // Default hashcode is enough here, as these are simple objects.
        return d.GetHashCode();
    }
}

...

// Gets all the short-haired dogs between two different kennels.
var allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, new DogHairLengthComparer());
```

```
Public Class DogHairLengthComparer
  Inherits IEqualityComparer(Of Dog)

  Public Function Equals(a As Dog,b As Dog) As Boolean
      If a Is Nothing AndAlso b Is Nothing Then
          Return True
      ElseIf (a Is Nothing AndAlso b IsNot Nothing) OrElse (a IsNot Nothing AndAlso b Is Nothing) Then
          Return False
      Else
          Return a.HairLengthType = b.HairLengthType
      End If
  End Function

  Public Function GetHashCode(d As Dog) As Integer
      ' Default hashcode is enough here, as these are simple objects.
      Return d.GetHashCode()
  End Function
End Class

...

' Gets all the short-haired dogs between two different kennels.
Dim allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, New DogHairLengthComparer())
```

- Intersection between two sets:

```
// Gets the volunteers who spend share time with two humane societies.
var volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     new VolunteerTimeComparer());
```

```
' Gets the volunteers who spend share time with two humane societies.
Dim volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     New VolunteerTimeComparer())
```

- Ordering:

```
// Get driving directions, ordering by if it's toll-free before estimated driving time.
var results = DirectionsProcessor.GetDirections(start, end)
              .OrderBy(direction => direction.HasNoTolls)
              .ThenBy(direction => direction.EstimatedTime);
```

```
' Get driving directions, ordering by if it's toll-free before estimated driving time.
Dim results = DirectionsProcessor.GetDirections(start, end).
                OrderBy(Function(direction) direction.HasNoTolls).
                ThenBy(Function(direction) direction.EstimatedTime)
```

- Finally, a more advanced sample: determining if the values of the properties of two instances of the same type are equal (Borrowed and modified from this StackOverflow post):

```
public static bool PublicInstancePropertiesEqual<T>(this T self, T to, params string[] ignore) where T : class
{
    if (self == null || to == null)
    {
        return self == to;
    }

    // Selects the properties which have unequal values into a sequence of those properties.
    var unequalProperties = from property in typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance)
                            where !ignore.Contains(property.Name)
                            let selfValue = property.GetValue(self, null)
                            let toValue = property.GetValue(to, null)
                            where !Equals(selfValue, toValue)
                            select property;
    return !unequalProperties.Any();
}
```

```
<System.Runtime.CompilerServices.Extension()>
Public Function PublicInstancePropertiesEqual(Of T As Class)(self As T, [to] As T, ParamArray ignore As
String()) As Boolean
    If self Is Nothing OrElse [to] Is Nothing Then
        Return self Is [to]
    End If

    ' Selects the properties which have unequal values into a sequence of those properties.
    Dim unequalProperties = From [property] In GetType(T).GetProperties(BindingFlags.Public Or
BindingFlags.Instance)
                            Where Not ignore.Contains([property].Name)
                            Let selfValue = [property].GetValue(self, Nothing)
                            Let toValue = [property].GetValue([to], Nothing)
                            Where Not Equals(selfValue, toValue) Select [property]
    Return Not unequalProperties.Any()
End Function
```

PLINQ

PLINQ, or Parallel LINQ, is a parallel execution engine for LINQ expressions. In other words, a regular LINQ expression can be trivially parallelized across any number of threads. This is accomplished via a call to `AsParallel()` preceding the expression.

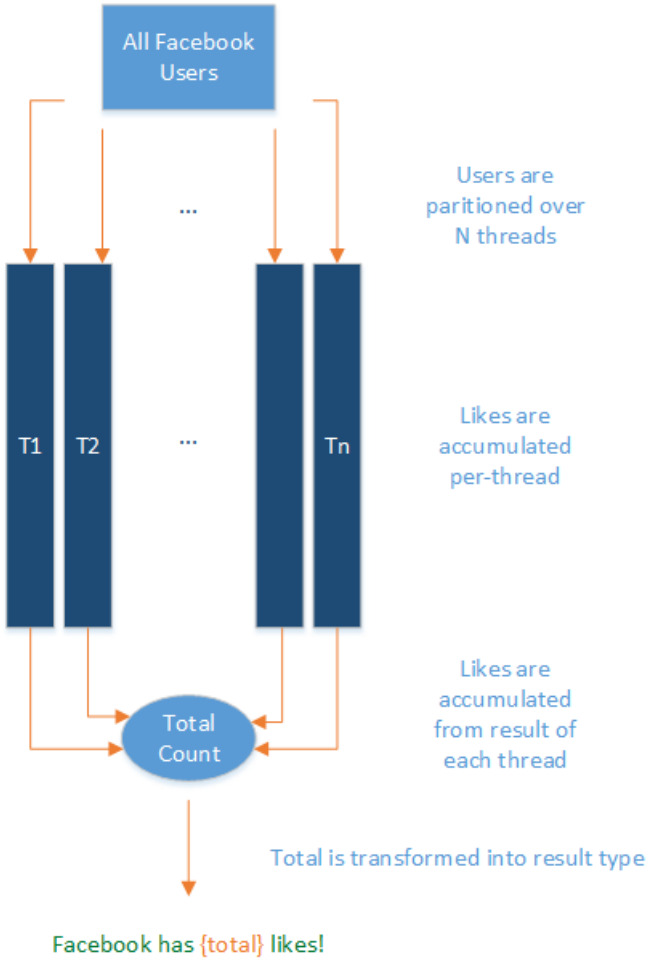
Consider the following:

```
public static string GetAllFacebookUserLikesMessage(IEnumerable<FacebookUser> facebookUsers)
{
    var seed = default(UInt64);

    Func<UInt64, UInt64, UInt64> threadAccumulator = (t1, t2) => t1 + t2;
    Func<UInt64, UInt64, UInt64> threadResultAccumulator = (t1, t2) => t1 + t2;
    Func<Uint64, string> resultSelector = total => $"Facebook has {total} likes!";

    return facebookUsers.AsParallel()
                        .Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector);
}
```

```
Public Shared GetAllFacebookUserLikesMessage(facebookUsers As IEnumerable(Of FacebookUser)) As String
{
    Dim seed As UInt64 = 0

    Dim threadAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim threadResultAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim resultSelector As Func(Of Uint64, string) = Function(total) $"Facebook has {total} likes!"

    Return facebookUsers.AsParallel().
                        Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector)
}
```

This code will partition `facebookUsers` across system threads as necessary, sum up the total likes on each thread in parallel, sum the results computed by each thread, and project that result into a nice string.

In diagram form:



Parallelizable CPU-bound jobs which can be easily expressed via LINQ (in other words, are pure functions and have no side effects) are a great candidate for PLINQ. For jobs which *do* have a side effect, consider using the Task Parallel Library.

Further Resources:

- 101 LINQ Samples
- Linqpad, a playground environment and Database querying engine for C#/F#/VB
- EduLinq, an e-book for learning how LINQ-to-objects is implemented

Common Type System & Common Language Specification

4/28/2019 • 2 minutes to read • Edit Online

Again, two terms that are freely used in the .NET world, they actually are crucial to understand how a .NET implementation enables multi-language development and to understand how it works.

Common Type System

To start from the beginning, remember that a .NET implementation is *language agnostic*. This doesn't just mean that a programmer can write her code in any language that can be compiled to IL. It also means that she needs to be able to interact with code written in other languages that are able to be used on a .NET implementation.

In order to do this transparently, there has to be a common way to describe all supported types. This is what the Common Type System (CTS) is in charge of doing. It was made to do several things:

- Establish a framework for cross-language execution.
- Provide an object-oriented model to support implementing various languages on a .NET implementation.
- Define a set of rules that all languages must follow when it comes to working with types.
- Provide a library that contains the basic primitive types that are used in application development (such as, `Boolean`, `Byte`, `Char` etc.)

CTS defines two main kinds of types that should be supported: reference and value types. Their names point to their definitions.

Reference types' objects are represented by a reference to the object's actual value; a reference here is similar to a pointer in C/C++. It simply refers to a memory location where the objects' values are. This has a profound impact on how these types are used. If you assign a reference type to a variable and then pass that variable into a method, for instance, any changes to the object will be reflected on the main object; there is no copying.

Value types are the opposite, where the objects are represented by their values. If you assign a value type to a variable, you are essentially copying a value of the object.

CTS defines several categories of types, each with their specific semantics and usage:

- Classes
- Structures
- Enums
- Interfaces
- Delegates

CTS also defines all other properties of the types, such as access modifiers, what are valid type members, how inheritance and overloading works and so on. Unfortunately, going deep into any of those is beyond the scope of an introductory article such as this, but you can consult More resources section at the end for links to more in-depth content that covers these topics.

Common Language Specification

To enable full interoperability scenarios, all objects that are created in code must rely on some commonality in the languages that are consuming them (are their *callers*). Since there are numerous different languages, .NET has

specified those commonalities in something called the **Common Language Specification** (CLS). CLS defines a set of features that are needed by many common applications. It also provides a sort of recipe for any language that is implemented on top of .NET on what it needs to support.

CLS is a subset of the CTS. This means that all of the rules in the CTS also apply to the CLS, unless the CLS rules are more strict. If a component is built using only the rules in the CLS, that is, it exposes only the CLS features in its API, it is said to be **CLS-compliant**. For instance, the `<framework-librares>` are CLS-compliant precisely because they need to work across all of the languages that are supported on .NET.

You can consult the documents in the More Resources section below to get an overview of all the features in the CLS.

More resources

- Common Type System
- Common Language Specification

Parallel Processing, Concurrency, and Async Programming in .NET

5/2/2018 • 2 minutes to read • Edit Online

.NET provides several ways for you to write asynchronous code to make your application more responsive to a user and write parallel code that uses multiple threads of execution to maximize the performance of your user's computer.

In This Section

Asynchronous Programming

Describes mechanisms for asynchronous programming provided by .NET.

Parallel Programming

Describes a task-based programming model that simplifies parallel development, enabling you to write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool.

Threading

Describes the basic concurrency and synchronization mechanisms provided by .NET.

Async Overview

4/28/2019 • 2 minutes to read • Edit Online

Not so long ago, apps got faster simply by buying a newer PC or server and then that trend stopped. In fact, it reversed. Mobile phones appeared with 1ghz single core ARM chips and server workloads transitioned to VMs. Users still want responsive UI and business owners want servers that scale with their business. The transition to mobile and cloud and an internet-connected population of >3B users has resulted in a new set of software patterns.

- Client applications are expected to be always-on, always-connected and constantly responsive to user interaction (for example, touch) with high app store ratings!
- Services are expected to handle spikes in traffic by gracefully scaling up and down.

Async programming is a key technique that makes it straightforward to handle blocking I/O and concurrent operations on multiple cores. .NET provides the capability for apps and services to be responsive and elastic with easy-to-use, language-level asynchronous programming models in C#, VB, and F#.

Why Write Async Code?

Modern apps make extensive use of file and networking I/O. I/O APIs traditionally block by default, resulting in poor user experiences and hardware utilization unless you want to learn and use challenging patterns. Task-based async APIs and the language-level asynchronous programming model invert this model, making async execution the default with few new concepts to learn.

Async code has the following characteristics:

- Handles more server requests by yielding threads to handle more requests while waiting for I/O requests to return.
- Enables UIs to be more responsive by yielding threads to UI interaction while waiting for I/O requests and by transitioning long-running work to other CPU cores.
- Many of the newer .NET APIs are asynchronous.
- It's easy to write async code in .NET!

What's next?

For more information, see the Async in depth topic.

The Asynchronous Programming Patterns topic provides an overview of the three asynchronous programming patterns supported in .NET:

- Asynchronous Programming Model (APM) (legacy)
- Event-based Asynchronous Pattern (EAP) (legacy)
- Task-based Asynchronous Pattern (TAP) (recommended for new development)

For more information about recommended task-based programming model, see the Task-based asynchronous programming topic.

Async in depth

8/20/2019 • 9 minutes to read • Edit Online

Writing I/O- and CPU-bound asynchronous code is straightforward using the .NET Task-based async model. The model is exposed by the `Task` and `Task<T>` types and the `async` and `await` keywords in C# and Visual Basic. (Language-specific resources are found in the See also section.) This article explains how to use .NET async and provides insight into the async framework used under the covers.

Task and Task<T>

Tasks are constructs used to implement what is known as the Promise Model of Concurrency. In short, they offer you a "promise" that work will be completed at a later point, letting you coordinate with the promise with a clean API.

- `Task` represents a single operation which does not return a value.
- `Task<T>` represents a single operation which returns a value of type `T`.

It's important to reason about tasks as abstractions of work happening asynchronously, and *not* an abstraction over threading. By default, tasks execute on the current thread and delegate work to the Operating System, as appropriate. Optionally, tasks can be explicitly requested to run on a separate thread via the `Task.Run` API.

Tasks expose an API protocol for monitoring, waiting upon and accessing the result value (in the case of `Task<T>`) of a task. Language integration, with the `await` keyword, provides a higher-level abstraction for using tasks.

Using `await` allows your application or service to perform useful work while a task is running by yielding control to its caller until the task is done. Your code does not need to rely on callbacks or events to continue execution after the task has been completed. The language and task API integration does that for you. If you're using `Task<T>`, the `await` keyword will additionally "unwrap" the value returned when the Task is complete. The details of how this works are explained further below.

You can learn more about tasks and the different ways to interact with them in the Task-based Asynchronous Pattern (TAP) topic.

Deeper Dive into Tasks for an I/O-Bound Operation

The following section describes a 10,000 foot view of what happens with a typical async I/O call. Let's start with a couple examples.

The first example calls an async method and returns an active task, likely yet to complete.

```
public Task<string> GetHtmlAsync()
{
    // Execution is synchronous here
    var client = new HttpClient();

    return client.GetStringAsync("https://www.dotnetfoundation.org");
}
```

The second example adds the use of the `async` and `await` keywords to operate on the task.

```
public async Task<string> GetFirstCharactersCountAsync(string url, int count)
{
    // Execution is synchronous here
    var client = new HttpClient();

    // Execution of GetFirstCharactersCountAsync() is yielded to the caller here
    // GetStringAsync returns a Task<string>, which is *awaited*
    var page = await client.GetStringAsync("https://www.dotnetfoundation.org");

    // Execution resumes when the client.GetStringAsync task completes,
    // becoming synchronous again.

    if (count > page.Length)
    {
        return page;
    }
    else
    {
        return page.Substring(0, count);
    }
}
```

The call to `GetStringAsync()` calls through lower-level .NET libraries (perhaps calling other async methods) until it reaches a P/Invoke interop call into a native networking library. The native library may subsequently call into a System API call (such as `write()` to a socket on Linux). A task object will be created at the native/managed boundary, possibly using TaskCompletionSource. The task object will be passed up through the layers, possibly operated on or directly returned, eventually returned to the initial caller.

In the second example above, a `Task<T>` object will be returned from `GetStringAsync`. The use of the `await` keyword causes the method to return a newly created task object. Control returns to the caller from this location in the `GetFirstCharactersCountAsync` method. The methods and properties of the Task<T> object enable callers to monitor the progress of the task, which will complete when the remaining code in GetFirstCharactersCountAsync has executed.

After the System API call, the request is now in kernel space, making its way to the networking subsystem of the OS (such as `/net` in the Linux Kernel). Here the OS will handle the networking request *asynchronously*. Details may be different depending on the OS used (the device driver call may be scheduled as a signal sent back to the runtime, or a device driver call may be made and *then* a signal sent back), but eventually the runtime will be informed that the networking request is in progress. At this time, the work for the device driver will either be scheduled, in-progress, or already finished (the request is already out "over the wire") - but because this is all happening asynchronously, the device driver is able to immediately handle something else!

For example, in Windows an OS thread makes a call to the network device driver and asks it to perform the networking operation via an Interrupt Request Packet (IRP) which represents the operation. The device driver receives the IRP, makes the call to the network, marks the IRP as "pending", and returns back to the OS. Because the OS thread now knows that the IRP is "pending", it doesn't have any more work to do for this job and "returns" back so that it can be used to perform other work.

When the request is fulfilled and data comes back through the device driver, it notifies the CPU of new data received via an interrupt. How this interrupt gets handled will vary depending on the OS, but eventually the data will be passed through the OS until it reaches a system interop call (for example, in Linux an interrupt handler will schedule the bottom half of the IRQ to pass the data up through the OS asynchronously). Note that this *also* happens asynchronously! The result is queued up until the next available thread is able to execute the async method and "unwrap" the result of the completed task.

Throughout this entire process, a key takeaway is that **no thread is dedicated to running the task**. Although work is executed in some context (that is, the OS does have to pass data to a device driver and respond to an interrupt), there is no thread dedicated to *waiting* for data from the request to come back. This allows the system

to handle a much larger volume of work rather than waiting for some I/O call to finish.

Although the above may seem like a lot of work to be done, when measured in terms of wall clock time, it's miniscule compared to the time it takes to do the actual I/O work. Although not at all precise, a potential timeline for such a call would look like this:

0-1
——2-
3

- Time spent from points `0` to `1` is everything up until an async method yields control to its caller.
- Time spent from points `1` to `2` is the time spent on I/O, with no CPU cost.
- Finally, time spent from points `2` to `3` is passing control back (and potentially a value) to the async method, at which point it is executing again.

What does this mean for a server scenario?

This model works well with a typical server scenario workload. Because there are no threads dedicated to blocking on unfinished tasks, the server threadpool can service a much higher volume of web requests.

Consider two servers: one that runs async code, and one that does not. For the purpose of this example, each server only has 5 threads available to service requests. Note that these numbers are imaginarily small and serve only in a demonstrative context.

Assume both servers receive 6 concurrent requests. Each request performs an I/O operation. The server *without* async code has to queue up the 6th request until one of the 5 threads have finished the I/O-bound work and written a response. At the point that the 20th request comes in, the server might start to slow down, because the queue is getting too long.

The server *with* async code running on it still queues up the 6th request, but because it uses `async` and `await`, each of its threads are freed up when the I/O-bound work starts, rather than when it finishes. By the time the 20th request comes in, the queue for incoming requests will be far smaller (if it has anything in it at all), and the server won't slow down.

Although this is a contrived example, it works in a very similar fashion in the real world. In fact, you can expect a server to be able to handle an order of magnitude more requests using `async` and `await` than if it were dedicating a thread for each request it receives.

What does this mean for client scenario?

The biggest gain for using `async` and `await` for a client app is an increase in responsiveness. Although you can make an app responsive by spawning threads manually, the act of spawning a thread is an expensive operation relative to just using `async` and `await`. Especially for something like a mobile game, impacting the UI thread as little as possible where I/O is concerned is crucial.

More importantly, because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

Additionally, dispatching work to the UI thread (such as updating a UI) is very simple with `async` methods, and does not require extra work (such as calling a thread-safe delegate).

Deeper Dive into Task and Task<T> for a CPU-Bound Operation

CPU-bound `async` code is a bit different than I/O-bound `async` code. Because the work is done on the CPU, there's no way to get around dedicating a thread to the computation. The use of `async` and `await` provides you with a clean way to interact with a background thread and keep the caller of the async method responsive. Note that this does not provide any protection for shared data. If you are using shared data, you will still need to apply an appropriate synchronization strategy.

Here's a 10,000 foot view of a CPU-bound async call:

```
public async Task<int> CalculateResult(InputData data)
{
    // This queues up the work on the threadpool.
    var expensiveResultTask = Task.Run(() => DoExpensiveCalculation(data));

    // Note that at this point, you can do some other work concurrently,
    // as CalculateResult() is still executing!

    // Execution of CalculateResult is yielded here!
    var result = await expensiveResultTask;

    return result;
}
```

`CalculateResult()` executes on the thread it was called on. When it calls `Task.Run`, it queues the expensive CPU-bound operation, `DoExpensiveCalculation()`, on the thread pool and receives a `Task<int>` handle. `DoExpensiveCalculation()` is eventually run concurrently on the next available thread, likely on another CPU core. It's possible to do concurrent work while `DoExpensiveCalculation()` is busy on another thread, because the thread which called `CalculateResult()` is still executing.

Once `await` is encountered, the execution of `CalculateResult()` is yielded to its caller, allowing other work to be done with the current thread while `DoExpensiveCalculation()` is churning out a result. Once it has finished, the result is queued up to run on the main thread. Eventually, the main thread will return to executing `CalculateResult()`, at which point it will have the result of `DoExpensiveCalculation()`.

Why does async help here?

`async` and `await` are the best practice for managing CPU-bound work when you need responsiveness. There are multiple patterns for using async with CPU-bound work. It's important to note that there is a small cost to using async and it's not recommended for tight loops. It's up to you to determine how you write your code around this new capability.

See also

- Asynchronous programming in C#
- Asynchronous programming with async and await (C#)
- Async Programming in F#
- Asynchronous Programming with Async and Await (Visual Basic)

Asynchronous programming patterns

8/20/2019 • 2 minutes to read • Edit Online

.NET provides three patterns for performing asynchronous operations:

- **Task-based Asynchronous Pattern (TAP)**, which uses a single method to represent the initiation and completion of an asynchronous operation. TAP was introduced in the .NET Framework 4. **It's the recommended approach to asynchronous programming in .NET.** The async and await keywords in C# and the Async and Await operators in Visual Basic add language support for TAP. For more information, see Task-based Asynchronous Pattern (TAP).
- **Event-based Asynchronous Pattern (EAP)**, which is the event-based legacy model for providing asynchronous behavior. It requires a method that has the `Async` suffix and one or more events, event handler delegate types, and `EventArg`-derived types. EAP was introduced in the .NET Framework 2.0. It's no longer recommended for new development. For more information, see Event-based Asynchronous Pattern (EAP).
- **Asynchronous Programming Model (APM)** pattern (also called the IAsyncResult pattern), which is the legacy model that uses the IAsyncResult interface to provide asynchronous behavior. In this pattern, synchronous operations require `Begin` and `End` methods (for example, `BeginWrite` and `EndWrite` to implement an asynchronous write operation). This pattern is no longer recommended for new development. For more information, see Asynchronous Programming Model (APM).

Comparison of patterns

For a quick comparison of how the three patterns model asynchronous operations, consider a `Read` method that reads a specified amount of data into a provided buffer starting at a specified offset:

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

The TAP counterpart of this method would expose the following single `ReadAsync` method:

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

The EAP counterpart would expose the following set of types and members:

```
public class MyClass
{
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```

The APM counterpart would expose the `BeginRead` and `EndRead` methods:

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

See also

- Async in depth
- Asynchronous programming in C#
- Async Programming in F#
- Asynchronous Programming with Async and Await (Visual Basic)

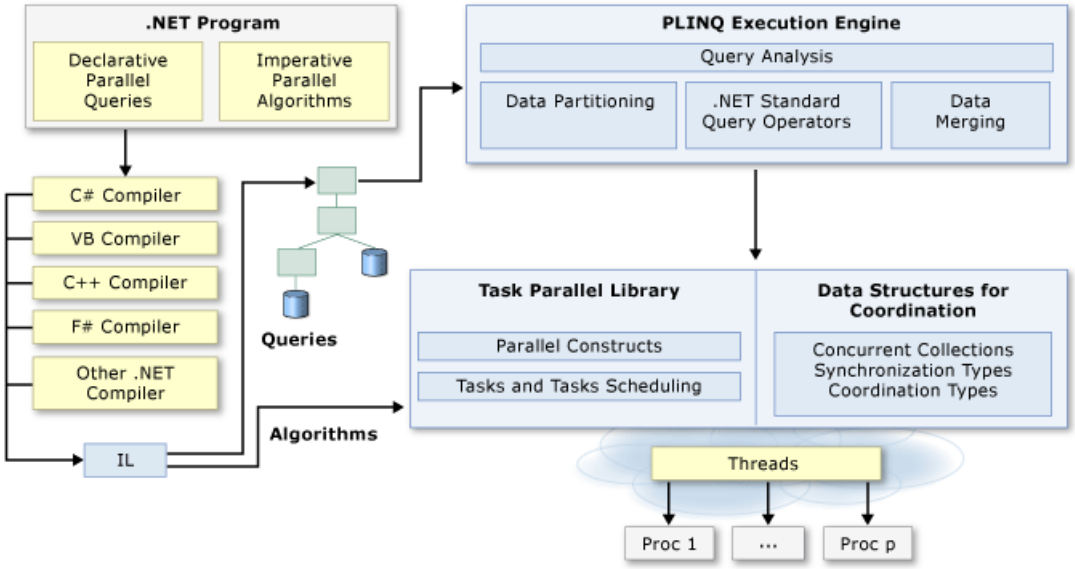
Parallel Programming in .NET

2/28/2019 • 2 minutes to read • Edit Online

Many personal computers and workstations have multiple CPU cores that enable multiple threads to be executed simultaneously. To take advantage of the hardware, you can parallelize your code to distribute work across multiple processors.

In the past, parallelization required low-level manipulation of threads and locks. Visual Studio and the .NET Framework enhance support for parallel programming by providing a runtime, class library types, and diagnostic tools. These features, which were introduced with the .NET Framework 4, simplify parallel development. You can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool.

The following illustration provides a high-level overview of the parallel programming architecture in the .NET Framework:



Related Topics

TECHNOLOGY	DESCRIPTION
Task Parallel Library (TPL)	Provides documentation for the System.Threading.Tasks.Parallel class, which includes parallel versions of `For` and `ForEach` loops, and also for the System.Threading.Tasks.Task class, which represents the preferred way to express asynchronous operations.
Parallel LINQ (PLINQ)	A parallel implementation of LINQ to Objects that significantly improves performance in many scenarios.
Data Structures for Parallel Programming	Provides links to documentation for thread-safe collection classes, lightweight synchronization types, and types for lazy initialization.

TECHNOLOGY	DESCRIPTION
Parallel Diagnostic Tools	Provides links to documentation for Visual Studio debugger windows for tasks and parallel stacks, and for the Concurrency Visualizer.
Custom Partitioners for PLINQ and TPL	Describes how partitioners work and how to configure the default partitioners or create a new partitioner.
Task Schedulers	Describes how schedulers work and how the default schedulers may be configured.
Lambda Expressions in PLINQ and TPL	Provides a brief overview of lambda expressions in C# and Visual Basic, and shows how they are used in PLINQ and the Task Parallel Library.
For Further Reading	Provides links to additional information and sample resources for parallel programming in .NET.

See also

- Async Overview
- Managed Threading

Managed Threading

8/22/2019 • 2 minutes to read • Edit Online

Whether you are developing for computers with one processor or several, you want your application to provide the most responsive interaction with the user, even if the application is currently doing other work. Using multiple threads of execution is one of the most powerful ways to keep your application responsive to the user and at the same time make use of the processor in between or even during user events. While this section introduces the basic concepts of threading, it focuses on managed threading concepts and using managed threading.

> **NOTE**
>
> Starting with the .NET Framework 4, multithreaded programming is greatly simplified with the System.Threading.Tasks.Parallel and System.Threading.Tasks.Task classes, Parallel LINQ (PLINQ), new concurrent collection classes in the System.Collections.Concurrent namespace, and a new programming model that is based on the concept of tasks rather than threads. For more information, see Parallel Programming.

In This Section

Managed Threading Basics
Provides an overview of managed threading and discusses when to use multiple threads.

Using Threads and Threading
Explains how to create, start, pause, resume, and abort threads.

Managed Threading Best Practices
Discusses levels of synchronization, how to avoid deadlocks and race conditions, and other threading issues.

Threading Objects and Features
Describes the managed classes you can use to synchronize the activities of threads and the data of objects accessed on different threads, and provides an overview of thread pool threads.

Reference

System.Threading
Contains classes for using and synchronizing managed threads.

System.Collections.Concurrent
Contains collection classes that are safe for use with multiple threads.

System.Threading.Tasks
Contains classes for creating and scheduling concurrent processing tasks.

Related Sections

Application Domains
Provides an overview of application domains and their use by the Common Language Infrastructure.

Asynchronous File I/O
Describes the performance advantages and basic operation of asynchronous I/O.

Task-based Asynchronous Pattern (TAP)
Provides an overview of the recommended pattern for asynchronous programming in .NET.

Calling Synchronous Methods Asynchronously

Explains how to call methods on thread pool threads using built-in features of delegates.

Parallel Programming

Describes the parallel programming libraries, which simplify the use of multiple threads in applications.

Parallel LINQ (PLINQ)

Describes a system for running queries in parallel, to take advantage of multiple processors.

Memory- and span-related types

8/20/2019 • 2 minutes to read • Edit Online

Starting with .NET Core 2.1, .NET includes a number of interrelated types that represent a contiguous, strongly-typed region of arbitrary memory. These include:

- System.Span<T>, a type that is used to access a contiguous region of memory. A Span<T> instance can be backed by an array of type `T`, a String, a buffer allocated with stackalloc, or a pointer to unmanaged memory. Because it has to be allocated on the stack, it has a number of restrictions. For example, a field in a class cannot be of type Span<T>, nor can span be used in asynchronous operations.
- System.ReadOnlySpan<T>, an immutable version of the Span<T> structure.
- System.Memory<T>, a contiguous region of memory that is allocated on the managed heap rather than the stack. A Memory<T> instance can be backed by an array of type `T` or a String. Because it can be stored on the managed heap, Memory<T> has none of the limitations of Span<T>.
- System.ReadOnlyMemory<T>, an immutable version of the Memory<T> structure.
- System.Buffers.MemoryPool<T>, which allocates strongly-typed blocks of memory from a memory pool to an owner. IMemoryOwner<T> instances can be rented from the pool by calling MemoryPool<T>.Rent and released back to the pool by calling MemoryPool<T>.Dispose().
- System.Buffers.IMemoryOwner<T>, which represents the owner of a block of memory and controls its lifetime management.
- MemoryManager<T>, an abstract base class that can be used to replace the implementation of Memory<T> so that Memory<T> can be backed by additional types, such as safe handles. MemoryManager<T> is intended for advanced scenarios.
- ArraySegment<T>, a wrapper for a particular number of array elements starting at a particular index.
- System.MemoryExtensions, a collection of extension methods for converting strings, arrays, and array segments to Memory<T> blocks.

> **NOTE**
> For earlier frameworks, Span<T> and Memory<T> are available in the System.Memory NuGet package.

For more information, see the System.Buffers namespace.

Working with memory and span

Because the memory- and span-related types are typically used to store data in a processing pipeline, it is important that developers follow a set of best practices when using Span<T>, Memory<T>, and related types. These best practices are documented in Memory<T> and Span<T> usage guidelines.

See also

- System.Memory<T>
- System.ReadOnlyMemory<T>
- System.Span<T>

- System.ReadOnlySpan<T>
- System.Buffers

Memory<T> and Span<T> usage guidelines

8/20/2019 • 14 minutes to read • Edit Online

.NET Core includes a number of types that represent an arbitrary contiguous region of memory. .NET Core 2.0 introduced Span<T> and ReadOnlySpan<T>, which are lightweight memory buffers that can be backed by managed or unmanaged memory. Because these types can only be stored on the stack, they are unsuitable for a number of scenarios, including asynchronous method calls. .NET Core 2.1 adds a number of additional types, including Memory<T>, ReadOnlyMemory<T>, IMemoryOwner<T>, and MemoryPool<T>. Like Span<T>, Memory<T> and its related types can be backed by both managed and unmanaged memory. Unlike Span<T>, Memory<T> can be stored on the managed heap.

Both Span<T> and Memory<T> are buffers of structured data that can be used in pipelines. That is, they are designed so that some or all of the data can be efficiently passed to components in the pipeline, which can process them and optionally modify the buffer. Because Memory<T> and its related types can be accessed by multiple components or by multiple threads, it's important that developers follow some standard usage guidelines to produce robust code.

Owners, consumers, and lifetime management

Since buffers can be passed around between APIs, and since buffers can sometimes be accessed from multiple threads, it's important to consider lifetime management. There are three core concepts:

- **Ownership**. The owner of a buffer instance is responsible for lifetime management, including destroying the buffer when it's no longer in use. All buffers have a single owner. Generally the owner is the component that created the buffer or that received the buffer from a factory. Ownership can also be transferred; **Component-A** can relinquish control of the buffer to **Component-B**, at which point **Component-A** may no longer use the buffer, and **Component-B** becomes responsible for destroying the buffer when it's no longer in use.
- **Consumption**. The consumer of a buffer instance is allowed to use the buffer instance by reading from it and possibly writing to it. Buffers can have one consumer at a time unless some external synchronization mechanism is provided. Note that the active consumer of a buffer isn't necessarily the buffer's owner.
- **Lease**. The lease is the length of time that a particular component is allowed to be the consumer of the buffer.

The following pseudo-code example illustrates these three concepts. It includes a `Main` method that instantiates a Memory<T> buffer of type Char, calls the `WriteInt32ToBuffer` method to write the string representation of an integer to the buffer, and then calls the `DisplayBufferToConsole` method to display the value of the buffer.

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

The `Main` method creates the buffer (in this case an Span<T> instance) and so is its owner. Therefore, `Main` is responsible for destroying the buffer when it's no longer in use. It does this by calling the buffer's Span<T>.Clear() method. (The Clear() method here actually clears the buffer's memory; the Span<T> structure doesn't actually have a method that destroys the buffer.)

The buffer has two consumers, `WriteInt32ToBuffer` and `DisplayBufferToConsole`. There is only one consumer at a time (first `WriteInt32ToBuffer`, then `DisplayBufferToConsole`), and neither of the consumers owns the buffer. Note also that "consumer" in this context doesn't imply a read-only view of the buffer; consumers can modify the buffer's contents, as `WriteInt32ToBuffer` does, if given a read/write view of the buffer.

The `WriteInt32ToBuffer` method has a lease on (can consume) the buffer between the start of the method call and the time the method returns. Similarly, `DisplayBufferToConsole` has a lease on the buffer while it's executing, and the lease is released when the method unwinds. (There is no API for lease management; a "lease" is a conceptual matter.)

Memory<T> and the owner/consumer model

As the Owners, consumers, and lifetime management section notes, a buffer always has an owner. .NET Core supports two ownership models:

- A model that supports single ownership. A buffer has a single owner for its entire lifetime.
- A model that supports ownership transfer. Ownership of a buffer can be transferred from its original owner (its creator) to another component, which then becomes responsible for the buffer's lifetime management. That owner can in turn transfer ownership to another component, and so on.

You use the System.Buffers.IMemoryOwner<T> interface to explicitly manage the ownership of a buffer. IMemoryOwner<T> supports both ownership models. The component that has an IMemoryOwner<T> reference owns the buffer. The following example uses an IMemoryOwner<T> instance to reflect the ownership of an Memory<T> buffer.

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try {
            var value = Int32.Parse(Console.ReadLine());

            var memory = owner.Memory;

            WriteInt32ToBuffer(value, memory);

            DisplayBufferToConsole(owner.Memory.Slice(0, value.ToString().Length));
        }
        catch (FormatException) {
            Console.WriteLine("You did not enter a valid number.");
        }
        catch (OverflowException) {
            Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
        }
        finally {
            owner?.Dispose();
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Span;
        for (int ctr = 0; ctr < strValue.Length; ctr++)
            span[ctr] = strValue[ctr];
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");

}
```

We can also write this example with the `using`:

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try {
                var value = Int32.Parse(Console.ReadLine());

                var memory = owner.Memory;
                WriteInt32ToBuffer(value, memory);
                DisplayBufferToConsole(memory.Slice(0, value.ToString().Length));
            }
            catch (FormatException) {
                Console.WriteLine("You did not enter a valid number.");
            }
            catch (OverflowException) {
                Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
            }
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Slice(0, strValue.Length).Span;
        strValue.AsSpan().CopyTo(span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

In this code:

- The `Main` method holds the reference to the IMemoryOwner<T> instance, so the `Main` method is the owner of the buffer.
- The `WriteInt32ToBuffer` and `DisplayBufferToConsole` methods accept Memory<T> as a public API. Therefore, they are consumers of the buffer. And they only consume it one at a time.

Although the `WriteInt32ToBuffer` method is intended to write a value to the buffer, the `DisplayBufferToConsole` method isn't. To reflect this, it could have accepted an argument of type ReadOnlyMemory<T>. For additional information on ReadOnlyMemory<T>, see Rule #2: Use ReadOnlySpan<T> or ReadOnlyMemory<T> if the buffer should be read-only.

"Ownerless" Memory<T> instances

You can create a Memory<T> instance without using IMemoryOwner<T>. In this case, ownership of the buffer is implicit rather than explicit, and only the single-owner model is supported. You can do this by:

- Calling one of the Memory<T> constructors directly, passing in a `T[]`, as the following example does.
- Calling the String.AsMemory extension method to produce a `ReadOnlyMemory<char>` instance.

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];

        Console.Write("Enter a number: ");
        var value = Int32.Parse(Console.ReadLine());

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

The method that initially creates the Memory<T> instance is the implicit owner of the buffer. Ownership cannot be transferred to any other component because there is no IMemoryOwner<T> instance to facilitate the transfer. (As an alternative, you can also imagine that the runtime's garbage collector owns the buffer, and all methods just consume the buffer.)

Usage guidelines

Because a memory block is owned but is intended to be passed to multiple components, some of which may operate upon a particular memory block simultaneously, it is important to establish guidelines for using both Memory<T> and Span<T>. Guidelines are necessary because:

- It is possible for a component to retain a reference to a memory block after its owner has released it.
- It is possible for a component to operate on a buffer at the same time that another component is operating on it, in the process corrupting the data in the buffer.
- While the stack-allocated nature of Span<T> optimizes performance and makes Span<T> the preferred type for operating on a memory block, it also subjects Span<T> to some major restrictions. It is important to know when to use a Span<T> and when to use Memory<T>.

The following are our recommendations for successfully using Memory<T> and its related types. Note that guidance that applies to Memory<T> and Span<T> also applies to ReadOnlyMemory<T> and ReadOnlySpan<T> unless we explicitly note otherwise.

Rule #1: For a synchronous API, use Span<T> instead of Memory<T> as a parameter if possible.

Span<T> is more versatile than Memory<T> and can represent a wider variety of contiguous memory buffers. Span<T> also offers better performance than Memory<T>. Finally, you can use the Memory<T>.Span property to convert a Memory<T> instance to a Span<T>, although Span<T>-to-Memory<T> conversion isn't possible. So if your callers happen to have a Memory<T> instance, they'll be able to call your methods with Span<T> parameters anyway.

Using a parameter of type Span<T> instead of type Memory<T> also helps you write a correct consuming method implementation. You'll automatically get compile-time checks to ensure that you're not attempting to access the buffer beyond your method's lease (more on this later).

Sometimes, you'll have to use a Memory<T> parameter instead of a Span<T> parameter, even if you're fully synchronous. Perhaps an API that you depend accepts only Memory<T> arguments. This is fine, but be aware of the tradeoffs involved when using Memory<T> synchronously.

Rule #2: Use ReadOnlySpan<T> or ReadOnlyMemory<T> if the buffer should be read-only.

In the earlier examples, the `DisplayBufferToConsole` method only reads from the buffer; it doesn't modify the contents of the buffer. The method signature should be changed to the following.

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

In fact, if we combine this rule and Rule #1, we can do even better and rewrite the method signature as follows:

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

The `DisplayBufferToConsole` method now works with virtually every buffer type imaginable: `T[]`, storage allocated with stackalloc, and so on. You can even pass a String directly into it!

Rule #3: If your method accepts Memory<T> and returns `void`, you must not use the Memory<T> instance after your method returns.

This relates to the "lease" concept mentioned earlier. A void-returning method's lease on theMemory<T> instance begins when the method is entered, and it ends when the method exits. Consider the following example, which calls `Log` in a loop based on input from the console.

```
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
        using (var owner = MemoryPool<char>.Shared.Rent())
        {
            var memory = owner.Memory;
            var span = memory.Span;
            while (true)
            {
                int value = Int32.Parse(Console.ReadLine());
                if (value < 0)
                    return;

                int numCharsWritten = ToBuffer(value, span);
                Log(memory.Slice(0, numCharsWritten));
            }
        }
    }

    private static int ToBuffer(int value, Span<char> span)
    {
        string strValue = value.ToString();
        int length = strValue.Length;
        strValue.AsSpan().CopyTo(span.Slice(0, length));
        return length;
    }
}
```

If `Log` is a fully synchronous method, this code will behave as expected because there is only one active consumer of the memory instance at any given time. But imagine instead that `Log` has this implementation.

```
// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
    });
}
```

In this implementation, `Log` violates its lease because it still attempts to use the Memory<T> instance in the background after the original method has returned. The `Main` method could mutate the buffer while `Log` attempts to read from it, which could result in data corruption.

There are several ways to resolve this:

- The `Log` method can return a Task instead of `void`, as the following implementation of the `Log` method does.

  ```
  // An acceptable implementation.
  static Task Log(ReadOnlyMemory<char> message)
  {
      // Run in the background so that we don't block the main thread while performing IO.
      return Task.Run(() => {
                  StreamWriter sw = File.AppendText(@".\input-numbers.dat");
          sw.WriteLine(message);
          sw.Flush();
      });
  }
  ```

- `Log` can instead be implemented as follows:

  ```
  // An acceptable implementation.
  static void Log(ReadOnlyMemory<char> message)
  {
      string defensiveCopy = message.ToString();
      // Run in the background so that we don't block the main thread while performing IO.
      Task.Run(() => {
          StreamWriter sw = File.AppendText(@".\input-numbers.dat");
          sw.WriteLine(defensiveCopy);
          sw.Flush();
      });
  }
  ```

Rule #4: If your method accepts a Memory<T> and returns a Task, you must not use the Memory<T> instance after the Task transitions to a terminal state.

This is just the async variant of Rule #3. The `Log` method from the earlier example can be written as follows to comply with this rule:

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while performing IO.
    Task.Run(() => {
        string defensiveCopy = message.ToString();
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

Here, "terminal state" means that the task transitions to a completed, faulted, or canceled state. In other words, "terminal state" means "anything that would cause await to throw or to continue execution."

This guidance applies to methods that return Task, Task<TResult>, ValueTask<TResult>, or any similar type.

Rule #5: If your constructor accepts Memory<T> as a parameter, instance methods on the constructed object are assumed to be consumers of the Memory<T> instance.

Consider the following example:

```
class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
      Console.WriteLine(value);
    }
}
```

Here, the `OddValueExtractor` constructor accepts a `ReadOnlyMemory<int>` as a constructor parameter, so the constructor itself is a consumer of the `ReadOnlyMemory<int>` instance, and all instance methods on the returned value are also consumers of the original `ReadOnlyMemory<int>` instance. This means that `TryReadNextOddValue` consumes the `ReadOnlyMemory<int>` instance, even though the instance isn't passed directly to the `TryReadNextOddValue` method.

Rule #6: If you have a settable Memory<T>-typed property (or an equivalent instance method) on your type, instance methods on that object are assumed to be consumers of the Memory<T> instance.

This is really just a variant of Rule #5. This rule exists because property setters or equivalent methods are assumed to capture and persist their inputs, so instance methods on the same object may utilize the captured state.

The following example triggers this rule:

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

Rule #7: If you have an IMemoryOwner<T> reference, you must at some point dispose of it or transfer its ownership (but not both).

Since a Memory<T> instance may be backed by either managed or unmanaged memory, the owner must call MemoryPool<T>.Dispose when work performed on the Memory<T> instance is complete. Alternatively, the owner may transfer ownership of the IMemoryOwner<T> instance to a different component, at which point the acquiring component becomes responsible for calling MemoryPool<T>.Dispose at the appropriate time (more on this later).

Failure to call the Dispose method may lead to unmanaged memory leaks or other performance degradation.

This rule also applies to code that calls factory methods like MemoryPool<T>.Rent. The caller becomes the owner of the returned IMemoryOwner<T> and is responsible for disposing of the instance when finished.

Rule #8: If you have an IMemoryOwner<T> parameter in your API surface, you are accepting ownership of that instance.

Accepting an instance of this type signals that your component intends to take ownership of this instance. Your component becomes responsible for proper disposal according to Rule #7.

Any component that transfers ownership of the IMemoryOwner<T> instance to a different component should no longer use that instance after the method call completes.

> **IMPORTANT**
> If your constructor accepts IMemoryOwner<T> as a parameter, its type should implement IDisposable, and your Dispose method should call MemoryPool<T>.Dispose.

Rule #9: If you're wrapping a synchronous p/invoke method, your API should accept Span<T> as a parameter.

According to Rule #1, Span<T> is generally the correct type to use for synchronous APIs. You can pin Span<T> instances via the `fixed` keyword, as in the following example.

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

In the previous example, `pbData` can be null if, for example, the input span is empty. If the exported method absolutely requires that `pbData` be non-null, even if `cbData` is 0, the method can be implemented as follows:

```
public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

Rule #10: If you're wrapping an asynchronous p/invoke method, your API should accept Memory<T> as a parameter.

Since you cannot use the `fixed` keyword across asynchronous operations, you use the Memory<T>.Pin method to pin Memory<T> instances, regardless of the kind of contiguous memory the instance represents. The following example shows how to use this API to perform an asynchronous p/invoke call.

```
using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);

[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int cbData, IntPtr pState, IntPtr
lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr = GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;
```

```
    // make the call
    int result;
    try
    {
        result = ExportedAsyncMethod((byte*)memoryHandle.Pointer, data.Length, pState, _callbackPtr);
    }
    catch
    {
        ((GCHandle)pState).Free(); // cleanup since callback won't be invoked
        memoryHandle.Dispose();
        throw;
    }

    if (result != PENDING)
    {
        // Operation completed synchronously; invoke callback manually
        // for result processing and cleanup.
        MyCompletedCallbackImplementation(pState, result);
    }

    return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)state;
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}

private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}
```

See also

- System.Memory<T>
- System.Buffers.IMemoryOwner<T>
- System.Span<T>

Native interoperability

5/6/2019 • 2 minutes to read • Edit Online

The following articles show the various ways of doing "native interoperability" in .NET.

There are a few reasons why you'd want to call into native code:

- Operating systems come with a large volume of APIs that aren't present in the managed class libraries. A prime example for this scenario would be access to hardware or operating system management functions.
- Communicating with other components that have or can produce C-style ABIs (native ABIs), such as Java code that is exposed via Java Native Interface (JNI) or any other managed language that could produce a native component.
- On Windows, most of the software that gets installed, such as the Microsoft Office suite, registers COM components that represent their programs and allow developers to automate them or use them. This also requires native interoperability.

The previous list doesn't cover all of the potential situations and scenarios in which the developer would want/like/need to interface with native components. .NET class library, for instance, uses the native interoperability support to implement a fair number of its APIs, like console support and manipulation, file system access and others. However, it's important to note that there's an option if needed.

> **NOTE**
> Most of the examples in this section will be presented for all three supported platforms for .NET Core (Windows, Linux and macOS). However, for some short and illustrative examples, just one sample is shown that uses Windows filenames and extensions (that is, "dll" for libraries). This doesn't mean that those features aren't available on Linux or macOS, it was done merely for convenience sake.

See also

- Platform Invoke (P/Invoke)
- Type marshaling
- Native interoperability best practices

Platform Invoke (P/Invoke)

5/14/2019 • 7 minutes to read • Edit Online

P/Invoke is a technology that allows you to access structs, callbacks, and functions in unmanaged libraries from your managed code. Most of the P/Invoke API is contained in two namespaces: `System` and `System.Runtime.InteropServices`. Using these two namespaces give you the tools to describe how you want to communicate with the native component.

Let's start from the most common example, and that is calling unmanaged functions in your managed code. Let's show a message box from a command-line application:

```
using System;
using System.Runtime.InteropServices;

public class Program
{
    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);

    public static void Main(string[] args)
    {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

The previous example is simple, but it does show off what's needed to invoke unmanaged functions from managed code. Let's step through the example:

- Line #1 shows the using statement for the `System.Runtime.InteropServices` namespace that holds all the items needed.
- Line #7 introduces the `DllImport` attribute. This attribute is crucial, as it tells the runtime that it should load the unmanaged DLL. The string passed in is the DLL our target function is in. Additionally, it specifies which character set to use for marshalling the strings. Finally, it specifies that this function calls SetLastError and that the runtime should capture that error code so the user can retrieve it via Marshal.GetLastWin32Error().
- Line #8 is the crux of the P/Invoke work. It defines a managed method that has the **exact same signature** as the unmanaged one. The declaration has a new keyword that you can notice, `extern`, which tells the runtime this is an external method, and that when you invoke it, the runtime should find it in the DLL specified in `DllImport` attribute.

The rest of the example is just invoking the method as you would any other managed method.

The sample is similar for macOS. The name of the library in the `DllImport` attribute needs to change since macOS has a different scheme of naming dynamic libraries. The following sample uses the `getpid(2)` function to get the process ID of the application and print it out to the console:

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libSystem shared library and define the method
        // corresponding to the native function.
        [DllImport("libSystem.dylib")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

It is also similar on Linux. The function name is the same, since `getpid(2)` is a standard POSIX system call.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libc shared library and define the method
        // corresponding to the native function.
        [DllImport("libc.so.6")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

Invoking managed code from unmanaged code

The runtime allows communication to flow in both directions, enabling you to call back into managed code from native functions by using function pointers. The closest thing to a function pointer in managed code is a **delegate**, so this is what is used to allow callbacks from native code into managed code.

The way to use this feature is similar to the managed to native process previously described. For a given callback, you define a delegate that matches the signature and pass that into the external method. The runtime will take care of everything else.

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
    public static class Program
    {
        // Define a delegate that corresponds to the unmanaged function.
        private delegate bool EnumWC(IntPtr hwnd, IntPtr lParam);

        // Import user32.dll (containing the function we need) and define
        // the method corresponding to the native function.
        [DllImport("user32.dll")]
        private static extern int EnumWindows(EnumWC lpEnumFunc, IntPtr lParam);

        // Define the implementation of the delegate; here, we simply output the window handle.
        private static bool OutputWindow(IntPtr hwnd, IntPtr lParam)
        {
            Console.WriteLine(hwnd.ToInt64());
            return true;
        }

        public static void Main(string[] args)
        {
            // Invoke the method; note the delegate as a first parameter.
            EnumWindows(OutputWindow, IntPtr.Zero);
        }
    }
}
```

Before walking through the example, it's good to review the signatures of the unmanaged functions you need to work with. The function to be called to enumerate all of the windows has the following signature:
`BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);`

The first parameter is a callback. The said callback has the following signature:
`BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);`

Now, let's walk through the example:

- Line #9 in the example defines a delegate that matches the signature of the callback from unmanaged code. Notice how the LPARAM and HWND types are represented using `IntPtr` in the managed code.
- Lines #13 and #14 introduce the `EnumWindows` function from the user32.dll library.
- Lines #17 - 20 implement the delegate. For this simple example, we just want to output the handle to the console.
- Finally, in line #24, the external method is called and passed in the delegate.

The Linux and macOS examples are shown below. For them, we use the `ftw` function that can be found in `libc`, the C library. This function is used to traverse directory hierarchies and it takes a pointer to a function as one of its parameters. The said function has the following signature:
`int (*fn) (const char *fpath, const struct stat *sb, int typeflag)`.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libc.so.6")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code. You can find more information
    // about this in the section on marshalling below.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

macOS example uses the same function, and the only difference is the argument to the `DllImport` attribute, as macOS keeps `libc` in a different place.

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libSystem.dylib")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

Both of the previous examples depend on parameters, and in both cases, the parameters are given as managed types. Runtime does the "right thing" and processes these into its equivalents on the other side. Learn about how types are marshaled to native code in our page on Type marshaling.

More resources

- PInvoke.net wiki an excellent Wiki with information on common Windows APIs and how to call them.
- P/Invoke in C++/CLI
- Mono documentation on P/Invoke

Type marshaling

6/13/2019 • 4 minutes to read • Edit Online

Marshaling is the process of transforming types when they need to cross between managed and native code.

Marshaling is needed because the types in the managed and unmanaged code are different. In managed code, for instance, you have a `String`, while in the unmanaged world strings can be Unicode ("wide"), non-Unicode, null-terminated, ASCII, etc. By default, the P/Invoke subsystem tries to do the right thing based on the default behavior, described on this article. However, for those situations where you need extra control, you can employ the MarshalAs attribute to specify what is the expected type on the unmanaged side. For instance, if you want the string to be sent as a null-terminated ANSI string, you could do it like this:

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

Default rules for marshaling common types

Generally, the runtime tries to do the "right thing" when marshaling to require the least amount of work from you. The following tables describe how each type is marshaled by default when used in a parameter or field. The C99/C++11 fixed-width integer and character types are used to ensure that the following table is correct for all platforms. You can use any native type that has the same alignment and size requirements as these types.

This first table describes the mappings for various types for whom the marshaling is the same for both P/Invoke and field marshaling.

.NET TYPE	NATIVE TYPE
`byte`	`uint8_t`
`sbyte`	`int8_t`
`short`	`int16_t`
`ushort`	`uint16_t`
`int`	`int32_t`
`uint`	`uint32_t`
`long`	`int64_t`
`ulong`	`uint64_t`
`char`	Either `char` or `char16_t` depending on the `CharSet` of the P/Invoke or structure. See the charset documentation.
`string`	Either `char*` or `char16_t*` depending on the `CharSet` of the P/Invoke or structure. See the charset documentation.

.NET TYPE	NATIVE TYPE
`System.IntPtr`	`intptr_t`
`System.UIntPtr`	`uintptr_t`
.NET Pointer types (ex. `void*`)	`void*`
Type derived from `System.Runtime.InteropServices.SafeHandle`	`void*`
Type derived from `System.Runtime.InteropServices.CriticalHandle`	`void*`
`bool`	Win32 `BOOL` type
`decimal`	COM `DECIMAL` struct
.NET Delegate	Native function pointer
`System.DateTime`	Win32 `DATE` type
`System.Guid`	Win32 `GUID` type

A few categories of marshaling have different defaults if you're marshaling as a parameter or structure.

.NET TYPE	NATIVE TYPE (PARAMETER)	NATIVE TYPE (FIELD)
.NET array	A pointer to the start of an array of native representations of the array elements.	Not allowed without a `[MarshalAs]` attribute
A class with a `LayoutKind` of `Sequential` or `Explicit`	A pointer to the native representation of the class	The native representation of the class

The following table includes the default marshaling rules that are Windows-only. On non-Windows platforms, you cannot marshal these types.

.NET TYPE	NATIVE TYPE (PARAMETER)	NATIVE TYPE (FIELD)	
`object`	`VARIANT`	`IUnknown*`	
`System.Array`	COM interface	Not allowed without a `[MarshalAs]` attribute	
`System.ArgIterator`	`va_list`	Not allowed	
`System.Collections.IEnumerat`	`IEnumVARIANT*`	Not allowed	
`System.Collections.IEnumerab`	`IDispatch*`	Not allowed	

.NET TYPE	NATIVE TYPE (PARAMETER)	NATIVE TYPE (FIELD)	
`System.DateTimeOffset`	`int64_t` representing the number of ticks since midnight on January 1, 1601		`int64_t` representing the number of ticks since midnight on January 1, 1601

Some types can only be marshaled as parameters and not as fields. These types are listed in the following table:

.NET TYPE	NATIVE TYPE (PARAMETER ONLY)
`System.Text.StringBuilder`	Either `char*` or `char16_t*` depending on the `CharSet` of the P/Invoke. See the charset documentation.
`System.ArgIterator`	`va_list` (on Windows x86/x64/arm64 only)
`System.Runtime.InteropServices.ArrayWithOffset`	`void*`
`System.Runtime.InteropServices.HandleRef`	`void*`

If these defaults don't do exactly what you want, you can customize how parameters are marshaled. The parameter marshaling article walks you through how to customize how different parameter types are marshaled.

Default marshaling in COM scenarios

When you are calling methods on COM objects in .NET, the .NET runtime changes the default marshaling rules to match common COM semantics. The following table lists the rules that .NET runtimes uses in COM scenarios:

.NET TYPE	NATIVE TYPE (COM METHOD CALLS)
`bool`	`VARIANT_BOOL`
`StringBuilder`	`LPWSTR`
`string`	`BSTR`
Delegate types	`_Delegate*` in .NET Framework. Disallowed in .NET Core.
`System.Drawing.Color`	`OLECOLOR`
.NET array	`SAFEARRAY`
`string[]`	`SAFEARRAY` of `BSTR`s

Marshaling classes and structs

Another aspect of type marshaling is how to pass in a struct to an unmanaged method. For instance, some of the unmanaged methods require a struct as a parameter. In these cases, you need to create a corresponding struct or a class in managed part of the world to use it as a parameter. However, just defining the class isn't enough, you also need to instruct the marshaler how to map fields in the class to the unmanaged struct. Here the `StructLayout` attribute becomes useful.

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime systemTime);

[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milsecond;
}

public static void Main(string[] args) {
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

The previous code shows a simple example of calling into `GetSystemTime()` function. The interesting bit is on line 4. The attribute specifies that the fields of the class should be mapped sequentially to the struct on the other (unmanaged) side. This means that the naming of the fields isn't important, only their order is important, as it needs to correspond to the unmanaged struct, shown in the following example:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME*;
```

Sometimes the default marshaling for your structure doesn't do what you need. The Customizing structure marshaling article teaches you how to customize how your structure is marshaled.

Customizing structure marshaling

8/15/2019 • 6 minutes to read • Edit Online

Sometimes the default marshaling rules for structures aren't exactly what you need. The .NET runtimes provide a few extension points for you to customize your structure's layout and how fields are marshaled.

Customizing structure layout

.NET provides the System.Runtime.InteropServices.StructLayoutAttribute attribute and the System.Runtime.InteropServices.LayoutKind enumeration to allow you to customize how fields are placed in memory. The following guidance will help you avoid common issues.

✔ **CONSIDER** using `LayoutKind.Sequential` whenever possible.

✔ **DO** only use `LayoutKind.Explicit` in marshaling when your native struct is also has an explicit layout, such as a union.

AVOID using `LayoutKind.Explicit` when marshaling structures on non-Windows platforms. The .NET Core runtime doesn't support passing explicit structures by value to native functions on Intel or AMD 64-bit non-Windows systems. However, the runtime supports passing explicit structures by reference on all platforms.

Customizing boolean field marshaling

Native code has many different boolean representations. On Windows alone, there are three ways to represent boolean values. The runtime doesn't know the native definition of your structure, so the best it can do is make a guess on how to marshal your boolean values. The .NET runtime provides a way to indicate how to marshal your boolean field. The following examples show how to marshal .NET `bool` to different native boolean types.

Boolean values default to marshaling as a native 4-byte Win32 `BOOL` value as shown in the following example:

```
public struct WinBool
{
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

If you want to be explicit, you can use the UnmanagedType.Bool value to get the same behavior as above:

```
public struct WinBool
{
    [MarshalAs(UnmanagedType.Bool)]
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

Using the `UnmanagedType.U1` or `UnmanagedType.I1` values below, you can tell the runtime to marshal the `b` field as a 1-byte native `bool` type.

```
public struct CBool
{
    [MarshalAs(UnmanagedType.U1)]
    public bool b;
}
```

```
struct CBool
{
    public bool b;
};
```

On Windows, you can use the UnmanagedType.VariantBool value to tell the runtime to marshal your boolean value to a 2-byte `VARIANT_BOOL` value:

```
public struct VariantBool
{
    [MarshalAs(UnmanagedType.VariantBool)]
    public bool b;
}
```

```
struct VariantBool
{
    public VARIANT_BOOL b;
};
```

NOTE

`VARIANT_BOOL` is different than most bool types in that `VARIANT_TRUE = -1` and `VARIANT_FALSE = 0`. Additionally, all values that aren't equal to `VARIANT_TRUE` are considered false.

Customizing array field marshaling

.NET also includes a few ways to customize array marshaling.

By default, .NET marshals arrays as a pointer to a contiguous list of the elements:

```
public struct DefaultArray
{
    public int[] values;
}
```

```
struct DefaultArray
{
    int* values;
};
```

If you're interfacing with COM APIs, you may have to marshal arrays as `SAFEARRAY*` objects. You can use the System.Runtime.InteropServices.MarshalAsAttribute and the UnmanagedType.SafeArray value to tell the runtime to marshal an array as a `SAFEARRAY*`:

```
public struct SafeArrayExample
{
    [MarshalAs(UnmanagedType.SafeArray)]
    public int[] values;
}
```

```
struct SafeArrayExample
{
    SAFEARRAY* values;
};
```

If you need to customize what type of element is in the `SAFEARRAY`, then you can use the MarshalAsAttribute.SafeArraySubType and MarshalAsAttribute.SafeArrayUserDefinedSubType fields to customize the exact element type of the `SAFEARRAY`.

If you need to marshal the array in-place, you can use the UnmanagedType.ByValArray value to tell the marshaler to marshal the array in-place. When you're using this marshaling, you also must supply a value to the MarshalAsAttribute.SizeConst field for the number of elements in the array so the runtime can correctly allocate space for the structure.

```
public struct InPlaceArray
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
    public int[] values;
}
```

```
struct InPlaceArray
{
    int values[4];
};
```

NOTE

.NET doesn't support marshaling a variable length array field as a C99 Flexible Array Member.

Customizing string field marshaling

.NET also provides a wide variety of customizations for marshaling string fields.

By default, .NET marshals a string as a pointer to a null-terminated string. The encoding depends on the value of the StructLayoutAttribute.CharSet field in the System.Runtime.InteropServices.StructLayoutAttribute. If no attribute is specified, the encoding defaults to an ANSI encoding.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char* str;
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

If you need to use different encodings for different fields or just prefer to be more explicit in your struct definition, you can use the UnmanagedType.LPStr or UnmanagedType.LPWStr values on a System.Runtime.InteropServices.MarshalAsAttribute attribute.

```
public struct AnsiString
{
    [MarshalAs(UnmanagedType.LPStr)]
    public string str;
}
```

```
struct AnsiString
{
    char* str;
};
```

```
public struct UnicodeString
{
    [MarshalAs(UnmanagedType.LPWStr)]
    public string str;
}
```

```
struct UnicodeString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

If you want to marshal your strings using the UTF-8 encoding, you can use the UnmanagedType.LPUTF8Str value in your MarshalAsAttribute.

```
public struct UTF8String
{
    [MarshalAs(UnmanagedType.LPUTF8Str)]
    public string str;
}
```

```
struct UTF8String
{
    char* str;
};
```

> **NOTE**
> Using UnmanagedType.LPUTF8Str requires either .NET Framework 4.7 (or later versions) or .NET Core 1.1 (or later versions). It isn't available in .NET Standard 2.0.

If you're working with COM APIs, you may need to marshal a string as a `BSTR`. Using the UnmanagedType.BStr value, you can marshal a string as a `BSTR`.

```
public struct BString
{
    [MarshalAs(UnmanagedType.BStr)]
    public string str;
}
```

```
struct BString
{
    BSTR str;
};
```

When using a WinRT-based API, you may need to marshal a string as an `HSTRING`. Using the UnmanagedType.HString value, you can marshal a string as a `HSTRING`.

```
public struct HString
{
    [MarshalAs(UnmanagedType.HString)]
    public string str;
}
```

```
struct BString
{
    HSTRING str;
};
```

If your API requires you to pass the string in-place in the structure, you can use the UnmanagedType.ByValTStr value. Do note that the encoding for a string marshaled by `ByValTStr` is determined from the `CharSet` attribute. Additionally, it requires that a string length is passed by the MarshalAsAttribute.SizeConst field.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char str[4];
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char16_t str[4]; // Could also be wchar_t[4] on Windows.
};
```

Customizing decimal field marshaling

If you're working on Windows, you might encounter some APIs that use the native `CY` or `CURRENCY` structure. By default, the .NET `decimal` type marshals to the native `DECIMAL` structure. However, you can use a MarshalAsAttribute with the UnmanagedType.Currency value to instruct the marshaler to convert a `decimal` value to a native `CY` value.

```
public struct Currency
{
    [MarshalAs(UnmanagedType.Currency)]
    public decimal dec;
}
```

```
struct Currency
{
    CY dec;
};
```

Marshaling `System.Object`s

On Windows, you can marshal `object`-typed fields to native code. You can marshal these fields to one of three types:

- `VARIANT`
- `IUnknown*`
- `IDispatch*`

By default, an `object`-typed field will be marshaled to an `IUnknown*` that wraps the object.

```
public struct ObjectDefault
{
    public object obj;
}
```

```
struct ObjectDefault
{
    IUnknown* obj;
};
```

If you want to marshal an object field to an `IDispatch*`, add a MarshalAsAttribute with the UnmanagedType.IDispatch value.

```
public struct ObjectDispatch
{
    [MarshalAs(UnmanagedType.IDispatch)]
    public object obj;
}
```

```
struct ObjectDispatch
{
    IDispatch* obj;
};
```

If you want to marshal it as a `VARIANT`, add a MarshalAsAttribute with the UnmanagedType.Struct value.

```
public struct ObjectVariant
{
    [MarshalAs(UnmanagedType.Struct)]
    public object obj;
}
```

```
struct ObjectVariant
{
    VARIANT obj;
};
```

The following table describes how different runtime types of the `obj` field map to the various types stored in a `VARIANT`:

.NET TYPE	VARIANT TYPE		.NET TYPE	VARIANT TYPE
`byte`	`VT_UI1`		`System.Runtime.InteropS`	`VT_BSTR` `BStrWrapper`
`sbyte`	`VT_I1`		`object`	`VT_DISPATCH`
`short`	`VT_I2`		`System.Runtime.InteropS`	`VT_UNKNOWN` `nownWrapper`
`ushort`	`VT_UI2`		`System.Runtime.InteropS`	`VT_DISPATCH` `atchWrapper`
`int`	`VT_I4`		`System.Reflection.Missi`	`VT_ERROR`

.NET TYPE	VARIANT TYPE		.NET TYPE	VARIANT TYPE
`uint`	`VT_UI4`		`(object)null`	`VT_EMPTY`
`long`	`VT_I8`		`bool`	`VT_BOOL`
`ulong`	`VT_UI8`		`System.DateTime`	`VT_DATE`
`float`	`VT_R4`		`decimal`	`VT_DECIMAL`
`double`	`VT_R8`		`System.Runtime.InteropS`	`VT_CURRENCY` `encyWrapper`
`char`	`VT_UI2`		`System.DBNull`	`VT_NULL`
`string`	`VT_BSTR`			

Customizing parameter marshaling

5/6/2019 • 2 minutes to read • Edit Online

When the .NET runtime's default parameter marshaling behavior doesn't do what you want, use can use the System.Runtime.InteropServices.MarshalAsAttribute attribute to customize how your parameters are marshaled.

Customizing string parameters

.NET has a variety of formats for marshaling strings. These methods are split into distinct sections on C-style strings and Windows-centric string formats.

C-Style strings

Each of these formats passes a null-terminated string to native code. They differ by the encoding of the native string.

`SYSTEM.RUNTIME.INTEROPSERVICES.UNMANAGEDTYPE` VALUE	ENCODING
LPStr	ANSI
LPUTF8Str	UTF-8
LPWStr	UTF-16
LPTStr	UTF-16

The UnmanagedType.VBByRefStr format is slightly different. Like `LPWStr`, it marshals the string to a native C-style string encoded in UTF-16. However, the managed signature has you pass in the string by reference and the matching native signature takes the string by value. This distinction allows you to use a native API that takes a string by value and modifies it in-place without having to use a `StringBuilder`. We recommend against manually using this format since it's prone to cause confusion with the mismatching native and managed signatures.

Windows-centric string formats

When interacting with COM or OLE interfaces, you'll likely find that the native functions take strings as `BSTR` arguments. You can use the UnmanagedType.BStr unmanaged type to marshal a string as a `BSTR`.

If you're interacting with WinRT APIs, you can use the UnmanagedType.HString format to marshal a string as an `HSTRING`.

Customizing array parameters

.NET also provides you multiple ways to marshal array parameters. If you're calling an API that takes a C-style array, use the UnmanagedType.LPArray unmanaged type. If the values in the array need customized marshaling, you can use the ArraySubType field on the `[MarshalAs]` attribute for that.

If you're using COM APIs, you'll likely have to marshal your array parameters as `SAFEARRAY*`s. To do so, you can use the UnmanagedType.SafeArray unmanaged type. The default type of the elements of the `SAFEARRAY` can be seen in the table on customizing `object` fields. You can use the MarshalAsAttribute.SafeArraySubType and MarshalAsAttribute.SafeArrayUserDefinedSubType fields to customize the exact element type of the `SAFEARRAY`.

Customizing boolean or decimal parameters

For information on marshaling boolean or decimal parameters, see Customizing structure marshaling.

Customizing object parameters (Windows-only)

On Windows, the .NET runtime provides a number of different ways to marshal object parameters to native code.

Marshaling as specific COM interfaces

If your API takes a pointer to a COM object, you can use any of the following `UnmanagedType` formats on an `object`-typed parameter to tell .NET to marshal as these specific interfaces:

- `IUnknown`
- `IDispatch`
- `IInspectable`

Additionally, if your type is marked `[ComVisible(true)]` or you're marshaling the `object` type, you can use the UnmanagedType.Interface format to marshal your object as a COM callable wrapper for the COM view of your type.

Marshaling to a `VARIANT`

If your native API takes a Win32 `VARIANT`, you can use the UnmanagedType.Struct format on your `object` parameter to marshal your objects as `VARIANT`s. See the documentation on customizing `object` fields for a mapping between .NET types and `VARIANT` types.

Custom marshalers

If you want to project a native COM interface into a different managed type, you can use the `UnmanagedType.CustomMarshaler` format and an implementation of ICustomMarshaler to provide your own custom marshaling code.

Native interoperability best practices

5/6/2019 • 9 minutes to read • Edit Online

.NET gives you a variety of ways to customize your native interoperability code. This article includes the guidance that Microsoft's .NET teams follow for native interoperability.

General guidance

The guidance in this section applies to all interop scenarios.

- ✔ **DO** use the same naming and capitalization for your methods and parameters as the native method you want to call.
- ✔ **CONSIDER** using the same naming and capitalization for constant values.
- ✔ **DO** use .NET types that map closest to the native type. For example, in C#, use `uint` when the native type is `unsigned int`.
- ✔ **DO** only use `[In]` and `[Out]` attributes when the behavior you want differs from the default behavior.
- ✔ **CONSIDER** using System.Buffers.ArrayPool<T> to pool your native array buffers.
- ✔ **CONSIDER** wrapping your P/Invoke declarations in a class with the same name and capitalization as your native library.
  - This allows your `[DllImport]` attributes to use the C# `nameof` language feature to pass in the name of the native library and ensure that you didn't misspell the name of the native library.

DllImport attribute settings

SETTING	DEFAULT	RECOMMENDATION	DETAILS
PreserveSig	`true`	keep default	When this is explicitly set to false, failed HRESULT return values will be turned into exceptions (and the return value in the definition becomes null as a result).
SetLastError	`false`	depends on the API	Set this to true if the API uses GetLastError and use Marshal.GetLastWin32Error to get the value. If the API sets a condition that says it has an error, get the error before making other calls to avoid inadvertently having it overwritten.
CharSet	`CharSet.None`, which falls back to `CharSet.Ansi` behavior	Explicitly use `CharSet.Unicode` or `CharSet.Ansi` when strings or characters are present in the definition	This specifies marshaling behavior of strings and what `ExactSpelling` does when `false`. Note that `CharSet.Ansi` is actually UTF8 on Unix. *Most* of the time Windows uses Unicode while Unix uses UTF8. See more information on the documentation on charsets.

SETTING	DEFAULT	RECOMMENDATION	DETAILS
ExactSpelling	`false`	`true`	Set this to true and gain a slight perf benefit as the runtime will not look for alternate function names with either an "A" or "W" suffix depending on the value of the `CharSet` setting ("A" for `CharSet.Ansi` and "W" for `CharSet.Unicode`).

String parameters

When the CharSet is Unicode or the argument is explicitly marked as `[MarshalAs(UnmanagedType.LPWSTR)]` *and* the string is passed by value (not `ref` or `out`), the string will be pinned and used directly by native code (rather than copied).

Remember to mark the `[DllImport]` as `Charset.Unicode` unless you explicitly want ANSI treatment of your strings.

DO NOT use `[Out] string` parameters. String parameters passed by value with the `[Out]` attribute can destabilize the runtime if the string is an interned string. See more information about string interning in the documentation for String.Intern.

AVOID `StringBuilder` parameters. `StringBuilder` marshaling *always* creates a native buffer copy. As such, it can be extremely inefficient. Take the typical scenario of calling a Windows API that takes a string:

1. Create a SB of the desired capacity (allocates managed capacity) **{1}**
2. Invoke
   a. Allocates a native buffer **{2}**
   b. Copies the contents if `[In]` *(the default for a* `StringBuilder` *parameter)*
   c. Copies the native buffer into a newly allocated managed array if `[Out]` **{3}** *(also the default for* `StringBuilder`*)*
3. `ToString()` allocates yet another managed array **{4}**

That is *{4}* allocations to get a string out of native code. The best you can do to limit this is to reuse the `StringBuilder` in another call but this still only saves *1* allocation. It's much better to use and cache a character buffer from `ArrayPool` - you can then get down to just the allocation for the `ToString()` on subsequent calls.

The other issue with `StringBuilder` is that it always copies the return buffer back up to the first null. If the passed back string isn't terminated or is a double-null-terminated string, your P/Invoke is incorrect at best.

If you *do* use `StringBuilder`, one last gotcha is that the capacity does **not** include a hidden null, which is always accounted for in interop. It's common for people to get this wrong as most APIs want the size of the buffer *including* the null. This can result in wasted/unnecessary allocations. Additionally, this gotcha prevents the runtime from optimizing `StringBuilder` marshaling to minimize copies.

✔ **CONSIDER** using `char[]`s from an `ArrayPool`.

For more information on string marshaling, see Default Marshaling for Strings and Customizing string marshaling.

> **Windows Specific**
> For `[Out]` strings the CLR will use `CoTaskMemFree` by default to free strings or `SysStringFree` for strings that are marked as `UnmanagedType.BSTR`.
> **For most APIs with an output string buffer:**
> The passed in character count must include the null. If the returned value is less than the passed in character count the call has succeeded and the value is the number of characters *without* the trailing null. Otherwise the count is

> the required size of the buffer *including* the null character.
>
> - Pass in 5, get 4: The string is 4 characters long with a trailing null.
> - Pass in 5, get 6: The string is 5 characters long, need a 6 character buffer to hold the null.
>
> Windows Data Types for Strings

Boolean parameters and fields

Booleans are easy to mess up. By default, a .NET `bool` is marshaled to a Windows `BOOL`, where it's a 4-byte value. However, the `_Bool`, and `bool` types in C and C++ are a *single* byte. This can lead to hard to track down bugs as half the return value will be discarded, which will only *potentially* change the result. For more for information on marshaling .NET `bool` values to C or C++ `bool` types, see the documentation on customizing boolean field marshaling.

GUIDs

GUIDs are usable directly in signatures. Many Windows APIs take `GUID&` type aliases like `REFIID`. When passed by ref, they can either be passed by `ref` or with the `[MarshalAs(UnmanagedType.LPStruct)]` attribute.

GUID	BY-REF GUID
`KNOWNFOLDERID`	`REFKNOWNFOLDERID`

 DO NOT Use `[MarshalAs(UnmanagedType.LPStruct)]` for anything other than `ref` GUID parameters.

Blittable types

Blittable types are types that have the same bit-level representation in managed and native code. As such they do not need to be converted to another format to be marshaled to and from native code, and as this improves performance they should be preferred.

Blittable types:

- `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `single`, `double`
- non-nested one-dimensional arrays of blittable types (for example, `int[]`)
- structs and classes with fixed layout that only have blittable value types for instance fields
  - fixed layout requires `[StructLayout(LayoutKind.Sequential)]` or `[StructLayout(LayoutKind.Explicit)]`
  - structs are `LayoutKind.Sequential` by default, classes are `LayoutKind.Auto`

NOT blittable:

- `bool`

SOMETIMES blittable:

- `char`, `string`

When blittable types are passed by reference, they're simply pinned by the marshaller instead of being copied to an intermediate buffer. (Classes are inherently passed by reference, structs are passed by reference when used with `ref` or `out`.)

`char` is blittable in a one-dimensional array **or** if it's part of a type that contains it's explicitly marked with `[StructLayout]` with `CharSet = CharSet.Unicode`.

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct UnicodeCharStruct
{
    public char c;
}
```

`string` is blittable if it isn't contained in another type and it's being passed as an argument that is marked with `[MarshalAs(UnmanagedType.LPWStr)]` or the `[DllImport]` has `CharSet = CharSet.Unicode` set.

You can see if a type is blittable by attempting to create a pinned `GCHandle`. If the type isn't a string or considered blittable, `GCHandle.Alloc` will throw an `ArgumentException`.

✔ **DO** make your structures blittable when possible.

For more information, see:

- Blittable and Non-Blittable Types
- Type Marshaling

Keeping managed objects alive

`GC.KeepAlive()` will ensure an object stays in scope until the KeepAlive method is hit.

`HandleRef` allows the marshaller to keep an object alive for the duration of a P/Invoke. It can be used instead of `IntPtr` in method signatures. `SafeHandle` effectively replaces this class and should be used instead.

`GCHandle` allows pinning a managed object and getting the native pointer to it. The basic pattern is:

```
GCHandle handle = GCHandle.Alloc(obj, GCHandleType.Pinned);
IntPtr ptr = handle.AddrOfPinnedObject();
handle.Free();
```

Pinning isn't the default for `GCHandle`. The other major pattern is for passing a reference to a managed object through native code and back to managed code, usually with a callback. Here is the pattern:

```
GCHandle handle = GCHandle.Alloc(obj);
SomeNativeEnumerator(callbackDelegate, GCHandle.ToIntPtr(handle));

// In the callback
GCHandle handle = GCHandle.FromIntPtr(param);
object managedObject = handle.Target;

// After the last callback
handle.Free();
```

Don't forget that `GCHandle` needs to be explicitly freed to avoid memory leaks.

Common Windows data types

Here is a list of data types commonly used in Windows APIs and which C# types to use when calling into the Windows code.

The following types are the same size on 32-bit and 64-bit Windows, despite their names.

WIDTH	WINDOWS	C (WINDOWS)	C#	ALTERNATIVE
32	`BOOL`	`int`	`int`	`bool`

WIDTH	WINDOWS	C (WINDOWS)	C#	ALTERNATIVE
8	`BOOLEAN`	`unsigned char`	`byte`	`[MarshalAs(UnmanagedType.U1` `bool`
8	`BYTE`	`unsigned char`	`byte`	
8	`CHAR`	`char`	`sbyte`	
8	`UCHAR`	`unsigned char`	`byte`	
16	`SHORT`	`short`	`short`	
16	`CSHORT`	`short`	`short`	
16	`USHORT`	`unsigned short`	`ushort`	
16	`WORD`	`unsigned short`	`ushort`	
16	`ATOM`	`unsigned short`	`ushort`	
32	`INT`	`int`	`int`	
32	`LONG`	`long`	`int`	
32	`ULONG`	`unsigned long`	`uint`	
32	`DWORD`	`unsigned long`	`uint`	
64	`QWORD`	`long long`	`long`	
64	`LARGE_INTEGER`	`long long`	`long`	
64	`LONGLONG`	`long long`	`long`	
64	`ULONGLONG`	`unsigned long long`	`ulong`	
64	`ULARGE_INTEGER`	`unsigned long long`	`ulong`	
32	`HRESULT`	`long`	`int`	
32	`NTSTATUS`	`long`	`int`	

The following types, being pointers, do follow the width of the platform. Use `IntPtr` / `UIntPtr` for these.

SIGNED POINTER TYPES (USE `INTPTR`)	UNSIGNED POINTER TYPES (USE `UINTPTR`)
`HANDLE`	`WPARAM`
`HWND`	`UINT_PTR`
`HINSTANCE`	`ULONG_PTR`

SIGNED POINTER TYPES (USE `INTPTR`)	UNSIGNED POINTER TYPES (USE `UINTPTR`)
`LPARAM`	`SIZE_T`
`LRESULT`	
`LONG_PTR`	
`INT_PTR`	

A Windows `PVOID` which is a C `void*` can be marshaled as either `IntPtr` or `UIntPtr`, but prefer `void*` when possible.

Windows Data Types

Data Type Ranges

Structs

Managed structs are created on the stack and aren't removed until the method returns. By definition then, they are "pinned" (it won't get moved by the GC). You can also simply take the address in unsafe code blocks if native code won't use the pointer past the end of the current method.

Blittable structs are much more performant as they can simply be used directly by the marshaling layer. Try to make structs blittable (for example, avoid `bool`). For more information, see the Blittable Types section.

If the struct is blittable, use `sizeof()` instead of `Marshal.SizeOf<MyStruct>()` for better performance. As mentioned above, you can validate that the type is blittable by attempting to create a pinned `GCHandle`. If the type is not a string or considered blittable, `GCHandle.Alloc` will throw an `ArgumentException`.

Pointers to structs in definitions must either be passed by `ref` or use `unsafe` and `*`.

✔ **DO** match the managed struct as closely as possible to the shape and names that are used in the official platform documentation or header.

✔ **DO** use the C# `sizeof()` instead of `Marshal.SizeOf<MyStruct>()` for blittable structures to improve performance.

An array like `INT_PTR Reserved1[2]` has to be marshaled to two `IntPtr` fields, `Reserved1a` and `Reserved1b`. When the native array is a primitive type, we can use the `fixed` keyword to write it a little more cleanly. For example, `SYSTEM_PROCESS_INFORMATION` looks like this in the native header:

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
...
} SYSTEM_PROCESS_INFORMATION
```

In C#, we can write it like this:

```
internal unsafe struct SYSTEM_PROCESS_INFORMATION
{
    internal uint NextEntryOffset;
    internal uint NumberOfThreads;
    private fixed byte Reserved1[48];
    internal Interop.UNICODE_STRING ImageName;
    ...
}
```

However, there are some gotchas with fixed buffers. Fixed buffers of non-blittable types won't be correctly marshaled, so the in-place array needs to be expanded out to multiple individual fields. Additionally, in .NET Framework and .NET Core before 3.0, if a struct containing a fixed buffer field is nested within a non-blittable struct, the fixed buffer field won't be correctly marshaled to native code.

Charsets and marshaling

8/6/2019 • 2 minutes to read • Edit Online

The way `char` values, `string` objects, and `System.Text.StringBuilder` objects are marshaled depends on the value of the `CharSet` field on either the P/Invoke or structure. You can set the `CharSet` of a P/Invoke by setting the DllImportAttribute.CharSet field when declaring your P/Invoke. To set the `CharSet` for a structure, set the StructLayoutAttribute.CharSet field on your struct declaration. When these attribute fields are not set, it is up to the language compiler to determine which `CharSet` to use. C# and VB use the Ansi charset by default.

The following table shows a mapping between each charset and how a character or string is represented when marshaled with that charset:

`CHARSET` VALUE	WINDOWS	.NET CORE 2.2 AND EARLIER ON UNIX	.NET CORE 3.0 AND LATER AND MONO ON UNIX
Ansi	`char` (the system default Windows (ANSI) code page)	`char` (UTF-8)	`char` (UTF-8)
Unicode	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char16_t` (UTF-16)
Auto	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char` (UTF-8)

Make sure you know what representation your native representation expects when picking your charset.

COM Interop in .NET

8/13/2019 • 2 minutes to read • Edit Online

The Component Object Model (COM) lets an object expose its functionality to other components and to host applications on Windows platforms. To help enable users to interoperate with their existing code bases, the .NET Framework has always provided strong support for interoperating with COM libraries. In .NET Core 3.0, a large portion of this support has been added to .NET Core on Windows. The documentation here will explain how the common COM interop techonologies work and how you can utilize them to interoperate with your existing COM libraries.

- COM Wrappers
- COM Callable Wrappers
- Runtime Callable Wrappers
- Qualifying .NET Types for COM Interoperation

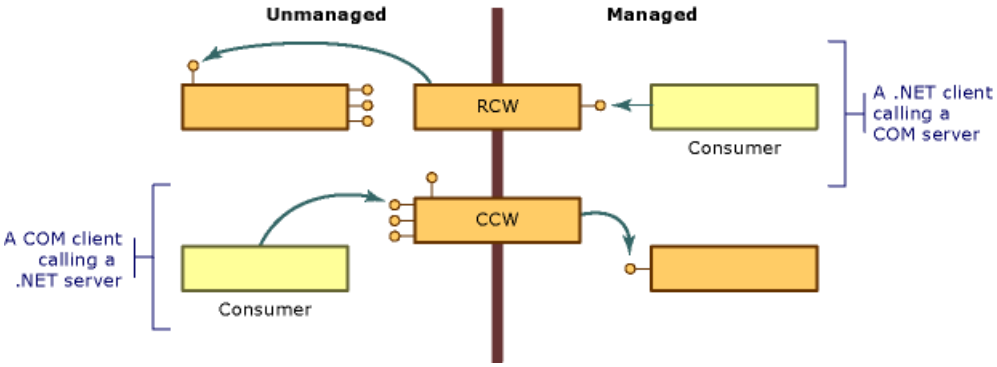
COM Wrappers

7/30/2019 • 2 minutes to read • Edit Online

COM differs from the .NET runtime object model in several important ways:

- Clients of COM objects must manage the lifetime of those objects; the common language runtime manages the lifetime of objects in its environment.
- Clients of COM objects discover whether a service is available by requesting an interface that provides that service and getting back an interface pointer, or not. Clients of .NET objects can obtain a description of an object's functionality using reflection.
- NET objects reside in memory managed by the .NET runtime execution environment. The execution environment can move objects around in memory for performance reasons and update all references to the objects it moves. Unmanaged clients, having obtained a pointer to an object, rely on the object to remain at the same location. These clients have no mechanism for dealing with an object whose location is not fixed.

To overcome these differences, the runtime provides wrapper classes to make both managed and unmanaged clients think they are calling objects within their respective environment. Whenever your managed client calls a method on a COM object, the runtime creates a runtime callable wrapper (RCW). RCWs abstract the differences between managed and unmanaged reference mechanisms, among other things. The runtime also creates a COM callable wrapper (CCW) to reverse the process, enabling a COM client to seamlessly call a method on a .NET object. As the following illustration shows, the perspective of the calling code determines which wrapper class the runtime creates.



In most cases, the standard RCW or CCW generated by the runtime provides adequate marshaling for calls that cross the boundary between COM and the .NET runtime. Using custom attributes, you can optionally adjust the way the runtime represents managed and unmanaged code.

See also

- Advanced COM Interoperability in .NET Framework
- Runtime Callable Wrapper
- COM Callable Wrapper
- Customizing Standard Wrappers in .NET Framework
- How to: Customize Runtime Callable Wrappers in .NET Framework

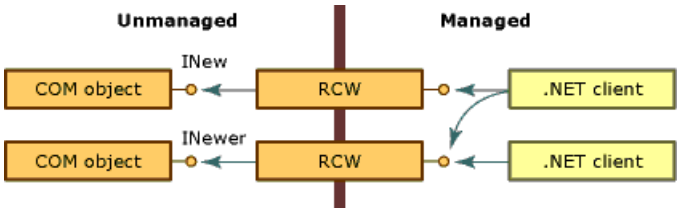
Runtime Callable Wrapper

7/30/2019 • 3 minutes to read • Edit Online

The common language runtime exposes COM objects through a proxy called the runtime callable wrapper (RCW). Although the RCW appears to be an ordinary object to .NET clients, its primary function is to marshal calls between a .NET client and a COM object.

The runtime creates exactly one RCW for each COM object, regardless of the number of references that exist on that object. The runtime maintains a single RCW per process for each object. If you create an RCW in one application domain or apartment, and then pass a reference to another application domain or apartment, a proxy to the first object will be used. As the following illustration shows, any number of managed clients can hold a reference to the COM objects that expose INew and INewer interfaces.

The following image shows the process for accessing COM objects through the runtime callable wrapper:



Using metadata derived from a type library, the runtime creates both the COM object being called and a wrapper for that object. Each RCW maintains a cache of interface pointers on the COM object it wraps and releases its reference on the COM object when the RCW is no longer needed. The runtime performs garbage collection on the RCW.

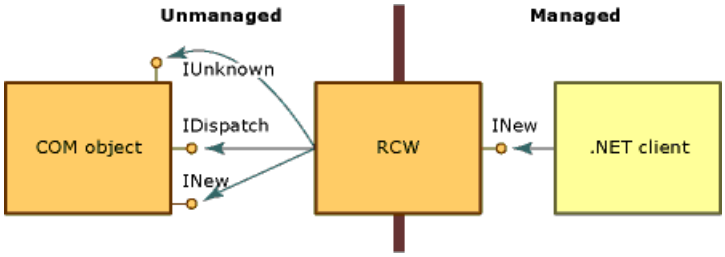
Among other activities, the RCW marshals data between managed and unmanaged code, on behalf of the wrapped object. Specifically, the RCW provides marshaling for method arguments and method return values whenever the client and server have different representations of the data passed between them.

The standard wrapper enforces built-in marshaling rules. For example, when a .NET client passes a String type as part of an argument to an unmanaged object, the wrapper converts the string to a BSTR type. Should the COM object return a BSTR to its managed caller, the caller receives a String. Both the client and the server send and receive data that is familiar to them. Other types require no conversion. For instance, a standard wrapper will always pass a 4-byte integer between managed and unmanaged code without converting the type.

Marshaling selected interfaces

The primary goal of the runtime callable wrapper (RCW) is to hide the differences between the managed and unmanaged programming models. To create a seamless transition, the RCW consumes selected COM interfaces without exposing them to the .NET client, as shown in the following illustration.

The following image shows COM interfaces and the runtime callable wrapper:



When created as an early-bound object, the RCW is a specific type. It implements the interfaces that the COM

object implements and exposes the methods, properties, and events from the object's interfaces. In the illustration, the RCW exposes the INew interface but consumes the **IUnknown** and **IDispatch** interfaces. Further, the RCW exposes all members of the INew interface to the .NET client.

The RCW consumes the interfaces listed in the following table, which are exposed by the object it wraps.

INTERFACE	DESCRIPTION
IDispatch	For late binding to COM objects through reflection.
IErrorInfo	Provides a textual description of the error, its source, a Help file, Help context, and the GUID of the interface that defined the error (always **GUID_NULL** for .NET classes).
IProvideClassInfo	If the COM object being wrapped implements **IProvideClassInfo**, the RCW extracts the type information from this interface to provide better type identity.
IUnknown	For object identity, type coercion, and lifetime management: - Object identity The runtime distinguishes between COM objects by comparing the value of the **IUnknown** interface for each object. - Type coercion The RCW recognizes the dynamic type discovery performed by the **QueryInterface** method. - Lifetime management Using the **QueryInterface** method, the RCW gets and holds a reference to an unmanaged object until the runtime performs garbage collection on the wrapper, which releases the unmanaged object.

The RCW optionally consumes the interfaces listed in the following table, which are exposed by the object it wraps.

INTERFACE	DESCRIPTION
IConnectionPoint and **IConnectionPointContainer**	The RCW converts objects that expose the connection-point event style to delegate-based events.
IDispatchEx (.NET Framework Only)	If the class implements **IDispatchEx**, the RCW implements **IExpando**. The **IDispatchEx** interface is an extension of the **IDispatch** interface that, unlike **IDispatch**, enables enumeration, addition, deletion, and case-sensitive calling of members.
IEnumVARIANT	Enables COM types that support enumerations to be treated as collections.

See also

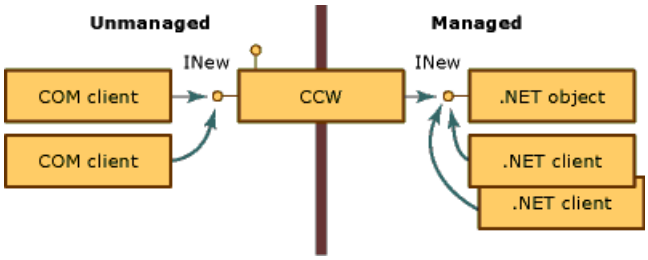
- COM Wrappers
- COM Callable Wrapper
- Type Library to Assembly Conversion Summary
- Importing a Type Library as an Assembly

COM Callable Wrapper

7/30/2019 • 8 minutes to read • Edit Online

When a COM client calls a .NET object, the common language runtime creates the managed object and a COM callable wrapper (CCW) for the object. Unable to reference a .NET object directly, COM clients use the CCW as a proxy for the managed object.

The runtime creates exactly one CCW for a managed object, regardless of the number of COM clients requesting its services. As the following illustration shows, multiple COM clients can hold a reference to the CCW that exposes the INew interface. The CCW, in turn, holds a single reference to the managed object that implements the interface and is garbage collected. Both COM and .NET clients can make requests on the same managed object simultaneously.



COM callable wrappers are invisible to other classes running within the .NET runtime. Their primary purpose is to marshal calls between managed and unmanaged code; however, CCWs also manage the object identity and object lifetime of the managed objects they wrap.

Object Identity

The runtime allocates memory for the .NET object from its garbage-collected heap, which enables the runtime to move the object around in memory as necessary. In contrast, the runtime allocates memory for the CCW from a noncollected heap, making it possible for COM clients to reference the wrapper directly.

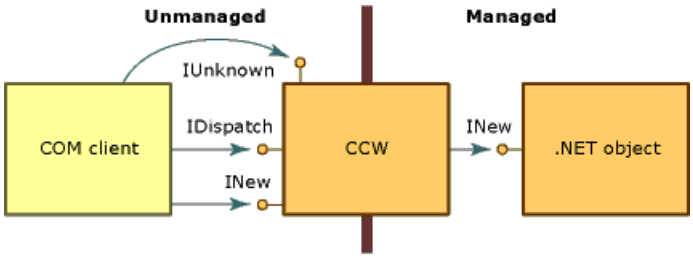
Object Lifetime

Unlike the .NET client it wraps, the CCW is reference-counted in traditional COM fashion. When the reference count on the CCW reaches zero, the wrapper releases its reference on the managed object. A managed object with no remaining references is collected during the next garbage-collection cycle.

Simulating COM interfaces

CCW exposes all public, COM-visible interfaces, data types, and return values to COM clients in a manner that is consistent with COM's enforcement of interface-based interaction. For a COM client, invoking methods on a .NET object is identical to invoking methods on a COM object.

To create this seamless approach, the CCW manufactures traditional COM interfaces, such as **IUnknown** and **IDispatch**. As the following illustration shows, the CCW maintains a single reference on the .NET object that it wraps. Both the COM client and the .NET object interact with each other through the proxy and stub construction of the CCW.



In addition to exposing the interfaces that are explicitly implemented by a class in the managed environment, the .NET runtime supplies implementations of the COM interfaces listed in the following table on behalf of the object. A .NET class can override the default behavior by providing its own implementation of these interfaces. However, the runtime always provides the implementation for the **IUnknown** and **IDispatch** interfaces.

INTERFACE	DESCRIPTION
IDispatch	Provides a mechanism for late binding to type.
IErrorInfo	Provides a textual description of the error, its source, a Help file, Help context, and the GUID of the interface that defined the error (always **GUID_NULL** for .NET classes).
IProvideClassInfo	Enables COM clients to gain access to the **ITypeInfo** interface implemented by a managed class. Returns `COR_E_NOTSUPPORTED` on .NET Core for types not imported from COM.
ISupportErrorInfo	Enables a COM client to determine whether the managed object supports the **IErrorInfo** interface. If so, enables the client to obtain a pointer to the latest exception object. All managed types support the **IErrorInfo** interface.
ITypeInfo (.NET Framework Only)	Provides type information for a class that is exactly the same as the type information produced by Tlbexp.exe.
IUnknown	Provides the standard implementation of the **IUnknown** interface with which the COM client manages the lifetime of the CCW and provides type coercion.

A managed class can also provide the COM interfaces described in the following table.

INTERFACE	DESCRIPTION
The (*_classname*) class interface	Interface, exposed by the runtime and not explicitly defined, that exposes all public interfaces, methods, properties, and fields that are explicitly exposed on a managed object.
IConnectionPoint and **IConnectionPointContainer**	Interface for objects that source delegate-based events (an interface for registering event subscribers).
IDispatchEx (.NET Framework Only)	Interface supplied by the runtime if the class implements **IExpando**. The **IDispatchEx** interface is an extension of the **IDispatch** interface that, unlike **IDispatch**, enables enumeration, addition, deletion, and case-sensitive calling of members.
IEnumVARIANT	Interface for collection-type classes, which enumerates the objects in the collection if the class implements **IEnumerable**.

Introducing the class interface

The class interface, which is not explicitly defined in managed code, is an interface that exposes all public methods, properties, fields, and events that are explicitly exposed on the .NET object. This interface can be a dual or dispatch-only interface. The class interface receives the name of the .NET class itself, preceded by an underscore. For example, for class Mammal, the class interface is _Mammal.

For derived classes, the class interface also exposes all public methods, properties, and fields of the base class. The derived class also exposes a class interface for each base class. For example, if class Mammal extends class MammalSuperclass, which itself extends System.Object, the .NET object exposes to COM clients three class interfaces named _Mammal, _MammalSuperclass, and _Object.

For example, consider the following .NET class:

```
' Applies the ClassInterfaceAttribute to set the interface to dual.
<ClassInterface(ClassInterfaceType.AutoDual)> _
' Implicitly extends System.Object.
Public Class Mammal
    Sub Eat()
    Sub Breathe()
    Sub Sleep()
End Class
```

```
// Applies the ClassInterfaceAttribute to set the interface to dual.
[ClassInterface(ClassInterfaceType.AutoDual)]
// Implicitly extends System.Object.
public class Mammal
{
    public void Eat() {}
    public void Breathe() {}
    public void Sleep() {}
}
```

The COM client can obtain a pointer to a class interface named `_Mammal`. On .NET Framework, you can use the Type Library Exporter (Tlbexp.exe) tool to generate a type library containing the `_Mammal` interface definition. The Type Library Exporter is not supported on .NET Core. If the `Mammal` class implemented one or more interfaces, the interfaces would appear under the coclass.

```
[odl, uuid(…), hidden, dual, nonextensible, oleautomation]
interface _Mammal : IDispatch
{
    [id(0x00000000), propget] HRESULT ToString([out, retval] BSTR*
        pRetVal);
    [id(0x60020001)] HRESULT Equals([in] VARIANT obj, [out, retval]
        VARIANT_BOOL* pRetVal);
    [id(0x60020002)] HRESULT GetHashCode([out, retval] short* pRetVal);
    [id(0x60020003)] HRESULT GetType([out, retval] _Type** pRetVal);
    [id(0x6002000d)] HRESULT Eat();
    [id(0x6002000e)] HRESULT Breathe();
    [id(0x6002000f)] HRESULT Sleep();
}
[uuid(…)]
coclass Mammal
{
    [default] interface _Mammal;
}
```

Generating the class interface is optional. By default, COM interop generates a dispatch-only interface for each class you export to a type library. You can prevent or modify the automatic creation of this interface by applying

the ClassInterfaceAttribute to your class. Although the class interface can ease the task of exposing managed classes to COM, its uses are limited.

Caution

Using the class interface, instead of explicitly defining your own, can complicate the future versioning of your managed class. Please read the following guidelines before using the class interface.

Define an explicit interface for COM clients to use rather than generating the class interface.

Because COM interop generates a class interface automatically, post-version changes to your class can alter the layout of the class interface exposed by the common language runtime. Since COM clients are typically unprepared to handle changes in the layout of an interface, they break if you change the member layout of the class.

This guideline reinforces the notion that interfaces exposed to COM clients must remain unchangeable. To reduce the risk of breaking COM clients by inadvertently reordering the interface layout, isolate all changes to the class from the interface layout by explicitly defining interfaces.

Use the **ClassInterfaceAttribute** to disengage the automatic generation of the class interface and implement an explicit interface for the class, as the following code fragment shows:

```
<ClassInterface(ClassInterfaceType.None)>Public Class LoanApp
    Implements IExplicit
    Sub M() Implements IExplicit.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.None)]
public class LoanApp : IExplicit
{
    int IExplicit.M() { return 0; }
}
```

The **ClassInterfaceType.None** value prevents the class interface from being generated when the class metadata is exported to a type library. In the preceding example, COM clients can access the `LoanApp` class only through the `IExplicit` interface.

Avoid caching dispatch identifiers (DispIds)

Using the class interface is an acceptable option for scripted clients, Microsoft Visual Basic 6.0 clients, or any late-bound client that does not cache the DispIds of interface members. DispIds identify interface members to enable late binding.

For the class interface, generation of DispIds is based on the position of the member in the interface. If you change the order of the member and export the class to a type library, you will alter the DispIds generated in the class interface.

To avoid breaking late-bound COM clients when using the class interface, apply the **ClassInterfaceAttribute** with the **ClassInterfaceType.AutoDispatch** value. This value implements a dispatch-only class interface, but omits the interface description from the type library. Without an interface description, clients are unable to cache DispIds at compile time. Although this is the default interface type for the class interface, you can apply the attribute value explicitly.

```
<ClassInterface(ClassInterfaceType.AutoDispatch)> Public Class LoanApp
    Implements IAnother
    Sub M() Implements IAnother.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.AutoDispatch)]
public class LoanApp
{
    public int M() { return 0; }
}
```

To get the DispId of an interface member at run time, COM clients can call **IDispatch.GetIdsOfNames**. To invoke a method on the interface, pass the returned DispId as an argument to **IDispatch.Invoke**.

Restrict using the dual interface option for the class interface.

Dual interfaces enable early and late binding to interface members by COM clients. At design time and during testing, you might find it useful to set the class interface to dual. For a managed class (and its base classes) that will never be modified, this option is also acceptable. In all other cases, avoid setting the class interface to dual.

An automatically generated dual interface might be appropriate in rare cases; however, more often it creates version-related complexity. For example, COM clients using the class interface of a derived class can easily break with changes to the base class. When a third party provides the base class, the layout of the class interface is out of your control. Further, unlike a dispatch-only interface, a dual interface (**ClassInterfaceType.AutoDual**) provides a description of the class interface in the exported type library. Such a description encourages late-bound clients to cache DispIds at compile time.

Ensure that all COM event notifications are late-bound.

By default, COM type information is embedded directly into managed assemblies, which eliminates the need for primary interop assemblies (PIAs). However, one of the limitations of embedded type information is that it does not support delivery of COM event notifications by early-bound vtable calls, but only supports late-bound `IDispatch::Invoke` calls.

If your application requires early-bound calls to COM event interface methods, you can set the **Embed Interop Types** property in Visual Studio to `true`, or include the following element in your project file:

```
<EmbedInteropTypes>True</EmbedInteropTypes>
```

See also

- ClassInterfaceAttribute
- COM Wrappers
- Exposing .NET Framework Components to COM
- Exposing .NET Core Components to COM
- Qualifying .NET Types for Interoperation
- Runtime Callable Wrapper

Qualifying .NET Types for COM Interoperation

7/30/2019 • 2 minutes to read • Edit Online

If you intend to expose types in an assembly to COM applications, consider the requirements of COM interop at design time. Managed types (class, interface, structure, and enumeration) seamlessly integrate with COM types when you adhere to the following guidelines:

- Classes should implement interfaces explicitly.

  Although COM interop provides a mechanism to automatically generate an interface containing all members of the class and the members of its base class, it is far better to provide explicit interfaces. The automatically generated interface is called the class interface. For guidelines, see Introducing the class interface.

  You can use Visual Basic, C#, and C++ to incorporate interface definitions in your code, instead of having to use Interface Definition Language (IDL) or its equivalent. For syntax details, see your language documentation.

- Managed types must be public.

  Only public types in an assembly are registered and exported to the type library. As a result, only public types are visible to COM.

  Managed types expose features to other managed code that might not be exposed to COM. For instance, parameterized constructors, static methods, and constant fields are not exposed to COM clients. Further, as the runtime marshals data in and out of a type, the data might be copied or transformed.

- Methods, properties, fields, and events must be public.

  Members of public types must also be public if they are to be visible to COM. You can restrict the visibility of an assembly, a public type, or public members of a public type by applying the ComVisibleAttribute. By default, all public types and members are visible.

- Types must have a public parameterless constructor to be activated from COM.

  Managed, public types are visible to COM. However, without a public parameterless constructor (a constructor with no arguments), COM clients cannot create the type. COM clients can still use the type if it is activated by some other means.

- Types cannot be abstract.

  Neither COM clients nor .NET clients can create abstract types.

When exported to COM, the inheritance hierarchy of a managed type is flattened. Versioning also differs between managed and unmanaged environments. Types exposed to COM do not have the same versioning characteristics as other managed types.

See also

- ComVisibleAttribute
- Exposing .NET Framework Components to COM
- Introducing the class interface
- Applying Interop Attributes
- Packaging a .NET Framework Assembly for COM

Applying Interop Attributes

7/30/2019 • 4 minutes to read • Edit Online

The System.Runtime.InteropServices namespace provides three categories of interop-specific attributes: those applied by you at design time, those applied by COM interop tools and APIs during the conversion process, and those applied either by you or COM interop.

If you are unfamiliar with the task of applying attributes to managed code, see Extending Metadata Using Attributes. Like other custom attributes, you can apply interop-specific attributes to types, methods, properties, parameters, fields, and other members.

Design-Time Attributes

You can adjust the outcome of the conversion process performed by COM interop tools and APIs by using design-time attributes. The following table describes the attributes that you can apply to your managed source code. COM interop tools, on occasion, might also apply the attributes described in this table.

ATTRIBUTE	DESCRIPTION
AutomationProxyAttribute	Specifies whether the type should be marshaled using the Automation marshaler or a custom proxy and stub.
ClassInterfaceAttribute	Controls the type of interface generated for a class.
CoClassAttribute	Identifies the CLSID of the original coclass imported from a type library. COM interop tools typically apply this attribute.
ComImportAttribute	Indicates that a coclass or interface definition was imported from a COM type library. The runtime uses this flag to know how to activate and marshal the type. This attribute prohibits the type from being exported back to a type library. COM interop tools typically apply this attribute.
ComRegisterFunctionAttribute	Indicates that a method should be called when the assembly is registered for use from COM, so that user-written code can be executed during the registration process.
ComSourceInterfacesAttribute	Identifies interfaces that are sources of events for the class. COM interop tools can apply this attribute.
ComUnregisterFunctionAttribute	Indicates that a method should be called when the assembly is unregistered from COM, so that user-written code can execute during the process.
ComVisibleAttribute	Renders types invisible to COM when the attribute value equals **false**. This attribute can be applied to an individual type or to an entire assembly to control COM visibility. By default, all managed, public types are visible; the attribute is not needed to make them visible.

ATTRIBUTE	DESCRIPTION
DispIdAttribute	Specifies the COM dispatch identifier (DISPID) of a method or field. This attribute contains the DISPID for the method, field, or property it describes. COM interop tools can apply this attribute.
FieldOffsetAttribute	Indicates the physical position of each field within a class when used with the **StructLayoutAttribute**, and the **LayoutKind** is set to Explicit.
GuidAttribute	Specifies the globally unique identifier (GUID) of a class, interface, or an entire type library. The string passed to the attribute must be a format that is an acceptable constructor argument for the type **System.Guid**. COM interop tools can apply this attribute.
IDispatchImplAttribute	Indicates which **IDispatch** interface implementation the common language runtime uses when exposing dual interfaces and dispinterfaces to COM.
InAttribute	Indicates that data should be marshaled in to the caller. Can be used to attribute parameters.
InterfaceTypeAttribute	Controls how a managed interface is exposed to COM clients (Dual, IUnknown-derived, or IDispatch only). COM interop tools can apply this attribute.
LCIDConversionAttribute	Indicates that an unmanaged method signature expects an LCID parameter. COM interop tools can apply this attribute.
MarshalAsAttribute	Indicates how the data in fields or parameters should be marshaled between managed and unmanaged code. The attribute is always optional because each data type has default marshaling behavior. COM interop tools can apply this attribute.
OptionalAttribute	Indicates that a parameter is optional. COM interop tools can apply this attribute.
OutAttribute	Indicates that the data in a field or parameter must be marshaled from a called object back to its caller.
PreserveSigAttribute	Suppresses the HRESULT or retval signature transformation that normally takes place during interoperation calls. The attribute affects marshaling as well as type library exporting. COM interop tools can apply this attribute.
ProgIdAttribute	Specifies the ProgID of a .NET Framework class. Can be used to attribute classes.

ATTRIBUTE	DESCRIPTION
StructLayoutAttribute	Controls the physical layout of the fields of a class. COM interop tools can apply this attribute.

Conversion-Tool Attributes

The following table describes attributes that COM interop tools apply during the conversion process. You do not apply these attributes at design time.

ATTRIBUTE	DESCRIPTION
ComAliasNameAttribute	Indicates the COM alias for a parameter or field type. Can be used to attribute parameters, fields, or return values.
ComConversionLossAttribute	Indicates that information about a class or interface was lost when it was imported from a type library to an assembly.
ComEventInterfaceAttribute	Identifies the source interface and the class that implements the methods of the event interface.
ImportedFromTypeLibAttribute	Indicates that the assembly was originally imported from a COM type library. This attribute contains the type library definition of the original type library.
TypeLibFuncAttribute	Contains the **FUNCFLAGS** that were originally imported for this function from the COM type library.
TypeLibTypeAttribute	Contains the **TYPEFLAGS** that were originally imported for this type from the COM type library.
TypeLibVarAttribute	Contains the **VARFLAGS** that were originally imported for this variable from the COM type library.

See also

- System.Runtime.InteropServices
- Exposing .NET Framework Components to COM
- Attributes
- Qualifying .NET Types for Interoperation
- Packaging a .NET Framework Assembly for COM

Collections and Data Structures

6/4/2019 • 4 minutes to read • Edit Online

Similar data can often be handled more efficiently when stored and manipulated as a collection. You can use the System.Array class or the classes in the System.Collections, System.Collections.Generic, System.Collections.Concurrent, System.Collections.Immutable namespaces to add, remove, and modify either individual elements or a range of elements in a collection.

There are two main types of collections; generic collections and non-generic collections. Generic collections were added in the .NET Framework 2.0 and provide collections that are type-safe at compile time. Because of this, generic collections typically offer better performance. Generic collections accept a type parameter when they are constructed and do not require that you cast to and from the Object type when you add or remove items from the collection. In addition, most generic collections are supported in Windows Store apps. Non-generic collections store items as Object, require casting, and most are not supported for Windows Store app development. However, you may see non-generic collections in older code.

Starting with the .NET Framework 4, the collections in the System.Collections.Concurrent namespace provide efficient thread-safe operations for accessing collection items from multiple threads. The immutable collection classes in the System.Collections.Immutable namespace (NuGet package) are inherently thread-safe because operations are performed on a copy of the original collection and the original collection cannot be modified.

Common collection features

All collections provide methods for adding, removing or finding items in the collection. In addition, all collections that directly or indirectly implement the ICollection interface or the ICollection<T> interface share these features:

- **The ability to enumerate the collection**

  .NET Framework collections either implement System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The foreach, in statement and the For Each...Next Statement use the enumerator exposed by the GetEnumerator method and hide the complexity of manipulating the enumerator. In addition, any collection that implements System.Collections.Generic.IEnumerable<T> is considered a *queryable type* and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They are typically more concise and readable than standard `foreach` loops, and provide filtering, ordering and grouping capabilities. LINQ queries can also improve performance. For more information, see LINQ to Objects (C#), LINQ to Objects (Visual Basic), Parallel LINQ (PLINQ), Introduction to LINQ Queries (C#), and Basic Query Operations (Visual Basic).

- **The ability to copy the collection contents to an array**

  All collections can be copied to an array using the **CopyTo** method; however, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- **Capacity and Count properties**

  The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This reduces the code required to use the collection; however, the performance of the collection might be negatively affected. For example, for List<T>, If Count is less than Capacity, adding an item is an O(1) operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an O(n) operation, where n is Count. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

A BitArray is a special case; its capacity is the same as its length, which is the same as its count.

- **A consistent lower bound**

  The lower bound of a collection is the index of its first element. All indexed collections in the System.Collections namespaces have a lower bound of zero, meaning they are 0-indexed. Array has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the **Array** class using Array.CreateInstance.

- **Synchronization for access from multiple threads** (System.Collections classes only).

  Non-generic collection types in the System.Collections namespace provide some thread safety with synchronization; typically exposed through the SyncRoot and IsSynchronized members. These collections are not thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the System.Collections.Concurrent namespace or consider using an immutable collection. For more information, see Thread-Safe Collections.

Choosing a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you are new to generic collections, this table will help you choose the generic collection that works the best for your task.

I WANT TO...	GENERIC COLLECTION OPTIONS	NON-GENERIC COLLECTION OPTIONS	THREAD-SAFE OR IMMUTABLE COLLECTION OPTIONS
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.)	ConcurrentDictionary<TKey, TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey, TValue>
Access items by index	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation

I WANT TO...	GENERIC COLLECTION OPTIONS	NON-GENERIC COLLECTION OPTIONS	THREAD-SAFE OR IMMUTABLE COLLECTION OPTIONS
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

Related Topics

TITLE	DESCRIPTION
Selecting a Collection Class	Describes the different collections and helps you select one for your scenario.
Commonly Used Collection Types	Describes commonly used generic and nongeneric collection types such as System.Array, System.Collections.Generic.List<T>, and System.Collections.Generic.Dictionary<TKey,TValue>.
When to Use Generic Collections	Discusses the use of generic collection types.
Comparisons and Sorts Within Collections	Discusses the use of equality comparisons and sorting comparisons in collections.
Sorted Collection Types	Describes sorted collections performance and characteristics
Hashtable and Dictionary Collection Types	Describes the features of generic and non-generic hash-based dictionary types.
Thread-Safe Collections	Describes collection types such as System.Collections.Concurrent.BlockingCollection<T> and System.Collections.Concurrent.ConcurrentBag<T> that support safe and efficient concurrent access from multiple threads.
System.Collections.Immutable	Introduces the immutable collections and provides links to the collection types.

Reference

System.Array
System.Collections
System.Collections.Concurrent

System.Collections.Generic
System.Collections.Specialized
System.Linq
System.Collections.Immutable

Numerics in .NET

10/24/2018 • 3 minutes to read • Edit Online

.NET provides a range of numeric integer and floating-point primitives, as well as System.Numerics.BigInteger, which is an integral type with no theoretical upper or lower bound, System.Numerics.Complex, which represents complex numbers, and a set of SIMD-enabled types in the System.Numerics namespace.

Integer types

.NET supports both signed and unsigned 8-, 16-, 32-, and 64-bit integer types, which are listed in the following table:

TYPE	SIGNED/UNSIGNED	SIZE (IN BYTES)	MINIMUM VALUE	MAXIMUM VALUE
System.Byte	Unsigned	1	0	255
System.Int16	Signed	2	-32,768	32,767
System.Int32	Signed	4	-2,147,483,648	2,147,483,647
System.Int64	Signed	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
System.SByte	Signed	1	-128	127
System.UInt16	Unsigned	2	0	65,535
System.UInt32	Unsigned	4	0	4,294,967,295
System.UInt64	Unsigned	8	0	18,446,744,073,709,551,615

Each integer type supports a set of standard arithmetic operators. The System.Math class provides methods for a broader set of mathematical functions.

You can also work with the individual bits in an integer value by using the System.BitConverter class.

> **NOTE**
> The unsigned integer types are not CLS-compliant. For more information, see Language Independence and Language-Independent Components.

BigInteger

The System.Numerics.BigInteger structure is an immutable type that represents an arbitrarily large integer whose value in theory has no upper or lower bounds. The methods of the BigInteger type closely parallel those of the other integral types.

Floating-point types

.NET includes three primitive floating-point types, which are listed in the following table:

TYPE	SIZE (IN BYTES)	APPROXIMATE RANGE	PRECISION
System.Single	4	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits
System.Double	8	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits
System.Decimal	16	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits

Both Single and Double types support special values that represent not-a-number and infinity. For example, the Double type provides the following values: Double.NaN, Double.NegativeInfinity, and Double.PositiveInfinity. You use the Double.IsNaN, Double.IsInfinity, Double.IsPositiveInfinity, and Double.IsNegativeInfinity methods to test for these special values.

Each floating-point type supports a set of standard arithmetic operators. The System.Math class provides methods for a broader set of mathematical functions. .NET Core 2.0 and later includes the System.MathF class that provides methods which accept arguments of the Single type.

You can also work with the individual bits in Double and Single values by using the System.BitConverter class. The System.Decimal structure has its own methods, Decimal.GetBits and Decimal.Decimal(Int32[]), for working with a decimal value's individual bits, as well as its own set of methods for performing some additional mathematical operations.

The Double and Single types are intended to be used for values that by their nature are imprecise (for example, the distance between two stars) and for applications in which a high degree of precision and small rounding error is not required. You should use the System.Decimal type for cases in which greater precision is required and rounding errors should be minimized.

> **NOTE**
> The Decimal type doesn't eliminate the need for rounding. Rather, it minimizes errors due to rounding.

Complex

The System.Numerics.Complex structure represents a complex number, that is, a number with a real number part and an imaginary number part. It supports a standard set of arithmetic, comparison, equality, explicit and implicit conversion operators, as well as mathematical, algebraic, and trigonometric methods.

SIMD-enabled types

The System.Numerics namespace includes a set of .NET SIMD-enabled types. SIMD (Single Instruction Multiple Data) operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

The .NET SIMD-enabled types include the following:

- The Vector2, Vector3, and Vector4 types, which represent vectors with 2, 3, and 4 Single values.
- Two matrix types, Matrix3x2, which represents a 3x2 matrix, and Matrix4x4, which represents a 4x4 matrix.
- The Plane type, which represents a plane in three-dimensional space.

- The Quaternion type, which represents a vector that is used to encode three-dimensional physical rotations.
- The Vector<T> type, which represents a vector of a specified numeric type and provides a broad set of operators that benefit from SIMD support. The count of a Vector<T> instance is fixed, but its value Vector<T>.Count depends on the CPU of the machine, on which code is executed.

  > **NOTE**
  >
  > The Vector<T> type is not included into the .NET Framework. You must install the System.Numerics.Vectors NuGet package to get access to this type.

The SIMD-enabled types are implemented in such a way that they can be used with non-SIMD-enabled hardware or JIT compilers. To take advantage of SIMD instructions, your 64-bit apps must be run by the runtime that uses the RyuJIT compiler, which is included in .NET Core and in the .NET Framework 4.6 and later versions. It adds SIMD support when targeting 64-bit processors.

See also

- Application Essentials
- Standard Numeric Format Strings

Dates, times, and time zones

5/2/2018 • 2 minutes to read • Edit Online

In addition to the basic DateTime structure, .NET provides the following classes that support working with time zones:

- TimeZone

  Use this class to work with the system's local time zone and the Coordinated Universal Time (UTC) zone.The functionality of the TimeZone class is largely superseded by the TimeZoneInfo class.

- TimeZoneInfo

  Use this class to work with any time zone that is predefined on a system, to create new time zones, and to easily convert dates and times from one time zone to another. For new development, use the TimeZoneInfo class instead of the TimeZone class.

- DateTimeOffset

  Use this structure to work with dates and times whose offset (or difference) from UTC is known. The DateTimeOffset structure combines a date and time value with that time's offset from UTC. Because of its relationship to UTC, an individual date and time value unambiguously identifies a single point in time. This makes a DateTimeOffset value more portable from one computer to another than a DateTime value.

This section of the documentation provides the information that you need to work with time zones and to create time zone-aware applications that can convert dates and times from one time zone to another.

In this section

Time zone overview Discusses the terminology, concepts, and issues involved in creating time zone-aware applications.

Choosing between DateTime, DateTimeOffset, TimeSpan, and TimeZoneInfo Discusses when to use the DateTime, DateTimeOffset, and TimeZoneInfo types when working with date and time data.

Finding the time zones defined on a local system Describes how to enumerate the time zones found on a local system.

How to: Enumerate time zones present on a computer Provides examples that enumerate the time zones defined in a computer's registry and that let users select a predefined time zone from a list.

How to: Access the predefined UTC and local time zone objects Describes how to access Coordinated Universal Time and the local time zone.

How to: Instantiate a TimeZoneInfo object Describes how to instantiate a TimeZoneInfo object from the local system registry.

Instantiating a DateTimeOffset object Discusses the ways in which a DateTimeOffset object can be instantiated, and the ways in which a DateTime value can be converted to a DateTimeOffset value.

How to: Create time zones without adjustment rules Describes how to create a custom time zone that does not support the transition to and from daylight saving time.

How to: Create time zones with adjustment rules Describes how to create a custom time zone that supports one or more transitions to and from daylight saving time.

Saving and restoring time zones Describes TimeZoneInfo support for serialization and deserialization of time zone data and illustrates some of the scenarios in which these features can be used.

How to: Save time zones to an embedded resource Describes how to create a custom time zone and save its information in a resource file.

How to: Restore time zones from an embedded resource Describes how to instantiate custom time zones that have been saved to an embedded resource file.

Performing arithmetic operations with dates and times Discusses the issues involved in adding, subtracting, and comparing DateTime and DateTimeOffset values.

How to: Use time zones in date and time arithmetic Discusses how to perform date and time arithmetic that reflects a time zone's adjustment rules.

Converting between DateTime and DateTimeOffset Describes how to convert between DateTime and DateTimeOffset values.

Converting times between time zones Describes how to convert times from one time zone to another.

How to: Resolve ambiguous times Describes how to resolve an ambiguous time by mapping it to the time zone's standard time.

How to: Let users resolve ambiguous times Describes how to let a user determine the mapping between an ambiguous local time and Coordinated Universal Time.

Reference

System.TimeZoneInfo

Handling and raising events

3/22/2019 • 7 minutes to read • Edit Online

Events in .NET are based on the delegate model. The delegate model follows the observer design pattern, which enables a subscriber to register with and receive notifications from a provider. An event sender pushes a notification that an event has happened, and an event receiver receives that notification and defines a response to it. This article describes the major components of the delegate model, how to consume events in applications, and how to implement events in your code.

For information about handling events in Windows 8.x Store apps, see Events and routed events overview.

Events

An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from some other program logic, such as changing a property's value. The object that raises the event is called the *event sender*. The event sender doesn't know which object or method will receive (handle) the events it raises. The event is typically a member of the event sender; for example, the Click event is a member of the Button class, and the PropertyChanged event is a member of the class that implements the INotifyPropertyChanged interface.

To define an event, you use the C# `event` or the Visual Basic `Event` keyword in the signature of your event class, and specify the type of delegate for the event. Delegates are described in the next section.

Typically, to raise an event, you add a method that is marked as `protected` and `virtual` (in C#) or `Protected` and `Overridable` (in Visual Basic). Name this method `On`*EventName*; for example, `OnDataReceived`. The method should take one parameter that specifies an event data object, which is an object of type EventArgs or a derived type. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the `On`*EventName* method of the base class to ensure that registered delegates receive the event.

The following example shows how to declare an event named `ThresholdReached`. The event is associated with the EventHandler delegate and raised in a method named `OnThresholdReached`.

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the class
}
```

```
Public Class Counter
    Public Event ThresholdReached As EventHandler

    Protected Overridable Sub OnThresholdReached(e As EventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    ' provide remaining implementation for the class
End Class
```

Delegates

A delegate is a type that holds a reference to a method. A delegate is declared with a signature that shows the return type and parameters for the methods it references, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. A delegate declaration is sufficient to define a delegate class.

Delegates have many uses in .NET. In the context of events, a delegate is an intermediary (or pointer-like mechanism) between the event source and the code that handles the event. You associate a delegate with an event by including the delegate type in the event declaration, as shown in the example in the previous section. For more information about delegates, see the Delegate class.

.NET provides the EventHandler and EventHandler<TEventArgs> delegates to support most event scenarios. Use the EventHandler delegate for all events that do not include event data. Use the EventHandler<TEventArgs> delegate for events that include data about the event. These delegates have no return type value and take two parameters (an object for the source of the event, and an object for event data).

Delegates are multicast, which means that they can hold references to more than one event-handling method. For details, see the Delegate reference page. Delegates provide flexibility and fine-grained control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

For scenarios where the EventHandler and EventHandler<TEventArgs> delegates do not work, you can define a delegate. Scenarios that require you to define a delegate are very rare, such as when you must work with code that does not recognize generics. You mark a delegate with the C# `delegate` and Visual Basic `Delegate` keyword in the declaration. The following example shows how to declare a delegate named `ThresholdReachedEventHandler`.

```
public delegate void ThresholdReachedEventHandler(object sender, ThresholdReachedEventArgs e);
```

```
Public Delegate Sub ThresholdReachedEventHandler(sender As Object, e As ThresholdReachedEventArgs)
```

Event data

Data that is associated with an event can be provided through an event data class. .NET provides many event data classes that you can use in your applications. For example, the SerialDataReceivedEventArgs class is the event data class for the SerialPort.DataReceived event. .NET follows a naming pattern of ending all event data classes with `EventArgs`. You determine which event data class is associated with an event by looking at the delegate for the event. For example, the SerialDataReceivedEventHandler delegate includes the SerialDataReceivedEventArgs class as one of its parameters.

The EventArgs class is the base type for all event data classes. EventArgs is also the class you use when an event does not have any data associated with it. When you create an event that is only meant to notify other classes that something happened and does not need to pass any data, include the EventArgs class as the second parameter in

the delegate. You can pass the EventArgs.Empty value when no data is provided. The EventHandler delegate includes the EventArgs class as a parameter.

When you want to create a customized event data class, create a class that derives from EventArgs, and then provide any members needed to pass data that is related to the event. Typically, you should use the same naming pattern as .NET and end your event data class name with `EventArgs`.

The following example shows an event data class named `ThresholdReachedEventArgs`. It contains properties that are specific to the event being raised.

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

```
Public Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class
```

Event handlers

To respond to an event, you define an event handler method in the event receiver. This method must match the signature of the delegate for the event you are handling. In the event handler, you perform the actions that are required when the event is raised, such as collecting user input after the user clicks a button. To receive notifications when the event occurs, your event handler method must subscribe to the event.

The following example shows an event handler method named `c_ThresholdReached` that matches the signature for the EventHandler delegate. The method subscribes to the `ThresholdReached` event.

```
class Program
{
    static void Main()
    {
        var c = new Counter();
        c.ThresholdReached += c_ThresholdReached;

        // provide remaining implementation for the class
    }

    static void c_ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
    }
}
```

```
Module Module1

    Sub Main()
        Dim c As New Counter()
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        ' provide remaining implementation for the class
    End Sub

    Sub c_ThresholdReached(sender As Object, e As EventArgs)
        Console.WriteLine("The threshold was reached.")
    End Sub
End Module
```

Static and dynamic event handlers

.NET allows subscribers to register for event notifications either statically or dynamically. Static event handlers are in effect for the entire life of the class whose events they handle. Dynamic event handlers are explicitly activated and deactivated during program execution, usually in response to some conditional program logic. For example, they can be used if event notifications are needed only under certain conditions or if an application provides multiple event handlers and run-time conditions define the appropriate one to use. The example in the previous section shows how to dynamically add an event handler. For more information, see Events (in Visual Basic) and Events (in C#).

Raising multiple events

If your class raises multiple events, the compiler generates one field per event delegate instance. If the number of events is large, the storage cost of one field per delegate may not be acceptable. For those situations, .NET provides event properties that you can use with another data structure of your choice to store event delegates.

Event properties consist of event declarations accompanied by event accessors. Event accessors are methods that you define to add or remove event delegate instances from the storage data structure. Note that event properties are slower than event fields, because each event delegate must be retrieved before it can be invoked. The trade-off is between memory and speed. If your class defines many events that are infrequently raised, you will want to implement event properties. For more information, see How to: Handle Multiple Events Using Event Properties.

Related topics

TITLE	DESCRIPTION
How to: Raise and Consume Events	Contains examples of raising and consuming events.
How to: Handle Multiple Events Using Event Properties	Shows how to use event properties to handle multiple events.
Observer Design Pattern	Describes the design pattern that enables a subscriber to register with, and receive notifications from, a provider.
How to: Consume Events in a Web Forms Application	Shows how to handle an event that is raised by a Web Forms control.

See also

- EventHandler
- EventHandler<TEventArgs>

- EventArgs
- Delegate
- Events (Visual Basic)
- Events (C# Programming Guide)
- Events and routed events overview (UWP apps)

Managed Execution Process

5/14/2019 • 7 minutes to read • Edit Online

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler.

   To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

2. Compiling your code to MSIL.

   Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

3. Compiling MSIL to native code.

   At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

4. Running code.

   The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Choosing a Compiler

To obtain the benefits provided by the common language runtime (CLR), you must use one or more language compilers that target the runtime, such as Visual Basic, C#, Visual C++, F#, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.

Because it is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. The language compiler you use determines which runtime features are available, and you design your code using those features. Your compiler, not the runtime, establishes the syntax your code must use. If your component must be completely usable by components written in other languages, your component's exported types must expose only language features that are included in the Language Independence and Language-Independent Components (CLS). You can use the CLSCompliantAttribute attribute to ensure that your code is CLS-compliant. For more information, see Language Independence and Language-Independent Components.

Back to top

Compiling to MSIL

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be run, MSIL must be converted to CPU-specific code, usually by a just-in-time (JIT) compiler. Because the common language runtime supplies one or more JIT compilers for each computer architecture it supports, the same set of MSIL can be JIT-compiled and run on any supported architecture.

When a compiler produces MSIL, it also produces metadata. Metadata describes the types in your code, including

the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time. The MSIL and metadata are contained in a portable executable (PE) file that is based on and that extends the published Microsoft PE and common object file format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file together with MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

Back to top

Compiling MSIL to Native Code

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. The .NET Framework provides two ways to perform this conversion:

- A .NET Framework just-in-time (JIT) compiler.
- The .NET Framework Ngen.exe (Native Image Generator).

Compilation by the JIT Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, if your managed code calls platform-specific native APIs or a platform-specific class library, it will run only on that operating system.

JIT compilation takes into account the possibility that some code might never be called during execution. Instead of using time and memory to convert all the MSIL in a PE file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Therefore, subsequent calls to the JIT-compiled method go directly to the native code.

Install-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it affects performance adversely at run time. In most cases, that diminished performance is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the Ngen.exe (Native Image Generator) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

- It performs the conversion from MSIL to native code before running the application instead of while the application is running.
- It compiles an entire assembly at a time, instead of one method at a time.
- It persists the generated code in the Native Image Cache as a file on disk.

Code Verification

As part of its compilation to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and

metadata to find out whether the code is type safe, which means that it accesses only the memory locations it is authorized to access. Type safety helps isolate objects from each other and helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

- A reference to a type is strictly compatible with the type being referenced.
- Only appropriately defined operations are invoked on an object.
- Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

Back to top

Running Code

The common language runtime provides the infrastructure that enables managed execution to take place and services that can be used during execution. Before a method can be run, it must be compiled to processor-specific code. Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, and then run. The next time the method is run, the existing JIT-compiled native code is run. The process of JIT-compiling and then running the code is repeated until execution is complete.

During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and enhanced deployment and versioning support.

In Microsoft Windows XP and Windows Vista, the operating system loader checks for managed modules by examining a bit in the COFF header. The bit being set denotes a managed module. If the loader detects managed modules, it loads mscoree.dll, and `_CorValidateImage` and `_CorImageUnloading` notify the loader when the managed module images are loaded and unloaded. `_CorValidateImage` performs the following actions:

1. Ensures that the code is valid managed code.
2. Changes the entry point in the image to an entry point in the runtime.

On 64-bit Windows, `_CorValidateImage` modifies the image that is in memory by transforming it from PE32 to PE32+ format.

Back to top

See also

- Overview
- Language Independence and Language-Independent Components
- Metadata and Self-Describing Components
- Ilasm.exe (IL Assembler)
- Security
- Interoperating with Unmanaged Code

- Deployment
- Assemblies in the Common Language Runtime
- Application Domains

Metadata and Self-Describing Components

4/26/2019 • 8 minutes to read • Edit Online

In the past, a software component (.exe or .dll) that was written in one language could not easily use a software component that was written in another language. COM provided a step towards solving this problem. The .NET Framework makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies. This information, called metadata, helps components to interact seamlessly.

Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, and your code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member that is defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
  - Identity (name, version, culture, public key).
  - The types that are exported.
  - Other assemblies that this assembly depends on.
  - Security permissions needed to run.
- Description of types.
  - Name, visibility, base class, and interfaces implemented.
  - Members (methods, fields, properties, events, nested types).
- Attributes.
  - Additional descriptive elements that modify types and members.

Benefits of Metadata

Metadata is the key to a simpler programming model, and eliminates the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata enables .NET Framework languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

- Self-describing files.

  Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, so you can use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

- Language interoperability and easier component-based design.

  Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshaling or using custom interoperability code.

- Attributes.

  The .NET Framework lets you declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout the .NET Framework and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET Framework files through user-defined custom attributes. For more information, see Attributes.

Metadata and the PE File Structure

Metadata is stored in one section of a .NET Framework portable executable (PE) file, while Microsoft intermediate language (MSIL) is stored in another section of the PE file. The metadata portion of the file contains a series of table and heap data structures. The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file. You might encounter metadata tokens when you use tools such as the MSIL Disassembler (Ildasm.exe) to view your code's MSIL, for example.

Metadata Tables and Heaps

Each metadata table holds information about the elements of your program. For example, one metadata table describes the classes in your code, another table describes the fields, and so on. If you have ten classes in your code, the class table will have tens rows, one for each class. Metadata tables reference other tables and heaps. For example, the metadata table for classes references the table for methods.

Metadata also stores information in four heap structures: string, blob, user string, and GUID. All the strings used to name types and members are stored in the string heap. For example, a method table does not directly store the name of a particular method, but points to the method's name stored in the string heap.

Metadata Tokens

Each row of each metadata table is uniquely identified in the MSIL portion of the PE file by a metadata token. Metadata tokens are conceptually similar to pointers, persisted in MSIL, that reference a particular metadata table.

A metadata token is a four-byte number. The top byte denotes the metadata table to which a particular token refers (method, type, and so on). The remaining three bytes specify the row in the metadata table that corresponds to the programming element being described. If you define a method in C# and compile it into a PE file, the following metadata token might exist in the MSIL portion of the PE file:

```
0x06000004
```

The top byte (`0x06`) indicates that this is a **MethodDef** token. The lower three bytes (`000004`) tells the common language runtime to look in the fourth row of the **MethodDef** table for the information that describes this method definition.

Metadata within a PE File

When a program is compiled for the common language runtime, it is converted to a PE file that consists of three parts. The following table describes the contents of each part.

PE SECTION	CONTENTS OF PE SECTION

PE SECTION	CONTENTS OF PE SECTION
PE header	The index of the PE file's main sections and the address of the entry point. The runtime uses this information to identify the file as a PE file and to determine where execution starts when loading the program into memory.
MSIL instructions	The Microsoft intermediate language instructions (MSIL) that make up your code. Many MSIL instructions are accompanied by metadata tokens.
Metadata	Metadata tables and heaps. The runtime uses this section to record information about every type and member in your code. This section also includes custom attributes and security information.

Run-Time Use of Metadata

To better understand metadata and its role in the common language runtime, it might be helpful to construct a simple program and illustrate how metadata affects its run-time life. The following code example shows two methods inside a class called `MyApp`. The `Main` method is the program entry point, while the `Add` method simply returns the sum of two integer arguments.

```
Public Class MyApp
   Public Shared Sub Main()
      Dim ValueOne As Integer = 10
      Dim ValueTwo As Integer = 20
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo))
   End Sub

   Public Shared Function Add(One As Integer, Two As Integer) As Integer
      Return (One + Two)
   End Function
End Class
```

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
   }
}
```

When the code runs, the runtime loads the module into memory and consults the metadata for this class. Once loaded, the runtime performs extensive analysis of the method's Microsoft intermediate language (MSIL) stream to convert it to fast native machine instructions. The runtime uses a just-in-time (JIT) compiler to convert the MSIL instructions to native machine code one method at a time as needed.

The following example shows part of the MSIL produced from the previous code's `Main` function. You can view the MSIL and metadata from any .NET Framework application using the MSIL Disassembler (Ildasm.exe).

```
.entrypoint
.maxstack  3
.locals ([0] int32 ValueOne,
         [1] int32 ValueTwo,
         [2] int32 V_2,
         [3] int32 V_3)
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldc.i4.s   20
IL_0005:  stloc.1
IL_0006:  ldstr      "The Value is: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003 */
```

The JIT compiler reads the MSIL for the whole method, analyzes it thoroughly, and generates efficient native instructions for the method. At `IL_000d`, a metadata token for the `Add` method (`/*` `06000003 */`) is encountered and the runtime uses the token to consult the third row of the **MethodDef** table.

The following table shows part of the **MethodDef** table referenced by the metadata token that describes the `Add` method. While other metadata tables exist in this assembly and have their own unique values, only this table is discussed.

ROW	RELATIVE VIRTUAL ADDRESS (RVA)	IMPLFLAGS	FLAGS	NAME (POINTS TO STRING HEAP.)	SIGNATURE (POINTS TO BLOB HEAP.)
1	0x00002050	IL Managed	Public ReuseSlot SpecialName RTSpecialName .ctor	.ctor (constructor)	
2	0x00002058	IL Managed	Public Static ReuseSlot	Main	String
3	0x0000208c	IL Managed	Public Static ReuseSlot	Add	int, int, int

Each column of the table contains important information about your code. The **RVA** column allows the runtime to calculate the starting memory address of the MSIL that defines this method. The **ImplFlags** and **Flags** columns contain bitmasks that describe the method (for example, whether the method is public or private). The **Name** column indexes the name of the method from the string heap. The **Signature** column indexes the definition of the method's signature in the blob heap.

The runtime calculates the desired offset address from the **RVA** column in the third row and returns this address

to the JIT compiler, which then proceeds to the new address. The JIT compiler continues to process MSIL at the new address until it encounters another metadata token and the process is repeated.

Using metadata, the runtime has access to all the information it needs to load your code and process it into native machine instructions. In this manner, metadata enables self-describing files and, together with the common type system, cross-language inheritance.

Related Topics

TITLE	DESCRIPTION
Attributes	Describes how to apply attributes, write custom attributes, and retrieve information that is stored in attributes.

Building Console Applications in the .NET Framework

8/22/2019 • 2 minutes to read • Edit Online

Applications in the .NET Framework can use the System.Console class to read characters from and write characters to the console. Data from the console is read from the standard input stream, data to the console is written to the standard output stream, and error data to the console is written to the standard error output stream. These streams are automatically associated with the console when the application starts and are presented as the In, Out, and Error properties, respectively.

The value of the Console.In property is a System.IO.TextReader object, whereas the values of the Console.Out and Console.Error properties are System.IO.TextWriter objects. You can associate these properties with streams that do not represent the console, making it possible for you to point the stream to a different location for input or output. For example, you can redirect the output to a file by setting the Console.Out property to a System.IO.StreamWriter, which encapsulates a System.IO.FileStream by means of the Console.SetOut method. The Console.In and Console.Out properties do not need to refer to the same stream.

> **NOTE**
> For more information about building console applications, including examples in C#, Visual Basic, and C++, see the documentation for the Console class.

If the console does not exist, as in a Windows-based application, output written to the standard output stream will not be visible, because there is no console to write the information to. Writing information to an inaccessible console does not cause an exception to be raised.

Alternately, to enable the console for reading and writing within a Windows-based application that is developed using Visual Studio, open the project's **Properties** dialog box, click the **Application** tab, and set the **Application type** to **Console Application**.

Console applications lack a message pump that starts by default. Therefore, console calls to Microsoft Win32 timers might fail.

The **System.Console** class has methods that can read individual characters or entire lines from the console. Other methods convert data and format strings, and then write the formatted strings to the console. For more information on formatting strings, see Formatting Types.

See also

- System.Console
- Formatting Types

.NET Framework Application Essentials

5/2/2018 • 2 minutes to read • Edit Online

This section of the .NET Framework documentation provides information about basic application development tasks in the .NET Framework.

In This Section

Base Types
Discusses formatting and parsing base data types and using regular expressions to process text.

Collections and Data Structures
Discusses the various collection types available in the .NET Framework, including stacks, queues, lists, arrays, and structs.

Generics
Describes the Generics feature, including the generic collections, delegates, and interfaces provided by the .NET Framework. Provides links to feature documentation for C#, Visual Basic and Visual C++, and to supporting technologies such as Reflection.

Numerics
Describes the numeric types in the .NET Framework.

Events
Provides an overview of the event model in the .NET Framework.

Exceptions
Describes error handling provided by the .NET Framework and the fundamentals of handling exceptions.

File and Stream I/O
Explains how you can perform synchronous and asynchronous file and data stream access and how to use to isolated storage.

Dates, Times, and Time Zones
Describes how to work with time zones and time zone conversions in time zone-aware applications.

Application Domains and Assemblies
Describes how to create and work with assemblies and application domains.

Serialization
Discusses the process of converting the state of an object into a form that can be persisted or transported.

Resources in Desktop Apps
Describes the .NET Framework support for creating and storing resources. This section also describes support for localized resources and the satellite assembly resource model for packaging and deploying those localized resources.

Globalization and Localization
Provides information to help you design and develop world-ready applications.

Accessibility
Provides information about Microsoft UI Automation, which is an accessibility framework that addresses the needs of assistive technology products and automated test frameworks by providing programmatic access to information about the user interface (UI).

Attributes

Describes how you can use attributes to customize metadata.

64-bit Applications

Discusses issues relevant to developing applications that will run on a Windows 64-bit operating system.

Related Sections

Development Guide

Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.

Security

Provides information about the classes and services in the common language runtime and the .NET Framework that facilitate secure application development.

File and Stream I/O

6/12/2019 • 7 minutes to read • Edit Online

File and stream I/O (input/output) refers to the transfer of data either to or from a storage medium. In the .NET Framework, the `System.IO` namespaces contain types that enable reading and writing, both synchronously and asynchronously, on data streams and files. These namespaces also contain types that perform compression and decompression on files, and types that enable communication through pipes and serial ports.

A file is an ordered and named collection of bytes that has persistent storage. When you work with files, you work with directory paths, disk storage, and file and directory names. In contrast, a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums (for example, disks or memory). Just as there are several backing stores other than disks, there are several kinds of streams other than file streams, such as network, memory, and pipe streams.

Files and directories

You can use the types in the System.IO namespace to interact with files and directories. For example, you can get and set properties for files and directories, and retrieve collections of files and directories based on search criteria.

For path naming conventions and the ways to express a file path for Windows systems, including with the DOS device syntax supported in .NET Core 1.1 and later and the .NET Framework 4.6.2 and later, see File path formats on Windows systems.

Here are some commonly used file and directory classes:

- File - provides static methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- FileInfo - provides instance methods for creating, copying, deleting, moving, and opening files, and helps create a FileStream object.
- Directory - provides static methods for creating, moving, and enumerating through directories and subdirectories.
- DirectoryInfo - provides instance methods for creating, moving, and enumerating through directories and subdirectories.
- Path - provides methods and properties for processing directory strings in a cross-platform manner.

You should always provide robust exception handling when calling filesystem methods. For more information, see Handling I/O errors.

In addition to using these classes, Visual Basic users can use the methods and properties provided by the Microsoft.VisualBasic.FileIO.FileSystem class for file I/O.

See How to: Copy Directories, How to: Create a Directory Listing, and How to: Enumerate Directories and Files.

Streams

The abstract base class Stream supports reading and writing bytes. All classes that represent streams inherit from the Stream class. The Stream class and its derived classes provide a common view of data sources and repositories, and isolate the programmer from the specific details of the operating system and underlying devices.

Streams involve three fundamental operations:

- Reading - transferring data from a stream into a data structure, such as an array of bytes.
- Writing - transferring data to a stream from a data source.
- Seeking - querying and modifying the current position within a stream.

Depending on the underlying data source or repository, a stream might support only some of these capabilities. For example, the PipeStream class does not support seeking. The CanRead, CanWrite, and CanSeek properties of a stream specify the operations that the stream supports.

Here are some commonly used stream classes:

- FileStream – for reading and writing to a file.
- IsolatedStorageFileStream – for reading and writing to a file in isolated storage.
- MemoryStream – for reading and writing to memory as the backing store.
- BufferedStream – for improving performance of read and write operations.
- NetworkStream – for reading and writing over network sockets.
- PipeStream – for reading and writing over anonymous and named pipes.
- CryptoStream – for linking data streams to cryptographic transformations.

For an example of working with streams asynchronously, see Asynchronous File I/O.

Readers and writers

The System.IO namespace also provides types for reading encoded characters from streams and writing them to streams. Typically, streams are designed for byte input and output. The reader and writer types handle the conversion of the encoded characters to and from bytes so the stream can complete the operation. Each reader and writer class is associated with a stream, which can be retrieved through the class's `BaseStream` property.

Here are some commonly used reader and writer classes:

- BinaryReader and BinaryWriter – for reading and writing primitive data types as binary values.
- StreamReader and StreamWriter – for reading and writing characters by using an encoding value to convert the characters to and from bytes.
- StringReader and StringWriter – for reading and writing characters to and from strings.
- TextReader and TextWriter – serve as the abstract base classes for other readers and writers that read and write characters and strings, but not binary data.

See How to: Read Text from a File, How to: Write Text to a File, How to: Read Characters from a String, and How to: Write Characters to a String.

Asynchronous I/O operations

Reading or writing a large amount of data can be resource-intensive. You should perform these tasks asynchronously if your application needs to remain responsive to the user. With synchronous I/O operations, the UI thread is blocked until the resource-intensive operation has completed. Use asynchronous I/O operations when developing Windows 8.x Store apps to prevent creating the impression that your app has stopped working.

The asynchronous members contain `Async` in their names, such as the CopyToAsync, FlushAsync, ReadAsync, and WriteAsync methods. You use these methods with the `async` and `await` keywords.

For more information, see Asynchronous File I/O.

Compression

Compression refers to the process of reducing the size of a file for storage. Decompression is the process of extracting the contents of a compressed file so they are in a usable format. The System.IO.Compression namespace contains types for compressing and decompressing files and streams.

The following classes are frequently used when compressing and decompressing files and streams:

- ZipArchive – for creating and retrieving entries in the zip archive.
- ZipArchiveEntry – for representing a compressed file.
- ZipFile – for creating, extracting, and opening a compressed package.
- ZipFileExtensions – for creating and extracting entries in a compressed package.
- DeflateStream – for compressing and decompressing streams using the Deflate algorithm.
- GZipStream – for compressing and decompressing streams in gzip data format.

See How to: Compress and Extract Files.

Isolated storage

Isolated storage is a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data. The storage provides a virtual file system that is isolated by user, assembly, and (optionally) domain. Isolated storage is particularly useful when your application does not have permission to access user files. You can save settings or files for your application in a manner that is controlled by the computer's security policy.

Isolated storage is not available for Windows 8.x Store apps; instead, use application data classes in the Windows.Storage namespace. For more information, see Application data.

The following classes are frequently used when implementing isolated storage:

- IsolatedStorage – provides the base class for isolated storage implementations.
- IsolatedStorageFile – provides an isolated storage area that contains files and directories.
- IsolatedStorageFileStream - exposes a file within isolated storage.

See Isolated Storage.

I/O operations in Windows Store apps

The .NET for Windows 8.x Store apps contains many of the types for reading from and writing to streams; however, this set does not include all the .NET Framework I/O types.

Some important differences to note when using I/O operations in Windows 8.x Store apps:

- Types specifically related to file operations, such as File, FileInfo, Directory and DirectoryInfo, are not included in the .NET for Windows 8.x Store apps. Instead, use the types in the Windows.Storage namespace of the Windows Runtime, such as StorageFile and StorageFolder.
- Isolated storage is not available; instead, use application data.
- Use asynchronous methods, such as ReadAsync and WriteAsync, to prevent blocking the UI thread.
- The path-based compression types ZipFile and ZipFileExtensions are not available. Instead, use the types in the Windows.Storage.Compression namespace.

You can convert between .NET Framework streams and Windows Runtime streams, if necessary. For more information, see How to: Convert Between .NET Framework Streams and Windows Runtime Streams or WindowsRuntimeStreamExtensions.

For more information about I/O operations in a Windows 8.x Store app, see Quickstart: Reading and writing files.

I/O and security

When you use the classes in the System.IO namespace, you must follow operating system security requirements such as access control lists (ACLs) to control access to files and directories. This requirement is in addition to any FileIOPermission requirements. You can manage ACLs programmatically. For more information, see How to: Add or Remove Access Control List Entries.

Default security policies prevent Internet or intranet applications from accessing files on the user's computer. Therefore, do not use the I/O classes that require a path to a physical file when writing code that will be downloaded over the Internet or intranet. Instead, use isolated storage for traditional .NET Framework applications, or use application data for Windows 8.x Store apps.

A security check is performed only when the stream is constructed. Therefore, do not open a stream and then pass it to less-trusted code or application domains.

Related topics

- Common I/O Tasks
  Provides a list of I/O tasks associated with files, directories, and streams, and links to relevant content and examples for each task.
- Asynchronous File I/O
  Describes the performance advantages and basic operation of asynchronous I/O.
- Isolated Storage
  Describes a data storage mechanism that provides isolation and safety by defining standardized ways of associating code with saved data.
- Pipes
  Describes anonymous and named pipe operations in the .NET Framework.
- Memory-Mapped Files
  Describes memory-mapped files, which contain the contents of files on disk in virtual memory. You can use memory-mapped files to edit very large files and to create shared memory for interprocess communication.

Globalizing and localizing .NET applications

4/28/2019 • 2 minutes to read • Edit Online

Developing a world-ready application, including an application that can be localized into one or more languages, involves three steps: globalization, localizability review, and localization.

Globalization

This step involves designing and coding an application that is culture-neutral and language-neutral, and that supports localized user interfaces and regional data for all users. It involves making design and programming decisions that are not based on culture-specific assumptions. While a globalized application is not localized, it nevertheless is designed and written so that it can be subsequently localized into one or more languages with relative ease.

Localizability review

This step involves reviewing an application's code and design to ensure that it can be localized easily and to identify potential roadblocks for localization, and verifying that the application's executable code is separated from its resources. If the globalization stage was effective, the localizability review will confirm the design and coding choices made during globalization. The localizability stage may also identify any remaining issues so that an application's source code doesn't have to be modified during the localization stage.

Localization

This step involves customizing an application for specific cultures or regions. If the globalization and localizability steps have been performed correctly, localization consists primarily of translating the user interface.

Following these three steps provides two advantages:

- It frees you from having to retrofit an application that is designed to support a single culture, such as U.S. English, to support additional cultures.
- It results in localized applications that are more stable and have fewer bugs.

.NET provides extensive support for the development of world-ready and localized applications. In particular, many type members in the .NET class library aid globalization by returning values that reflect the conventions of either the current user's culture or a specified culture. Also, .NET supports satellite assemblies, which facilitate the process of localizing an application.

For additional information, see the Globalization documentation.

In this section

Globalization

Discusses the first stage of creating a world-ready application, which involves designing and coding an application that is culture-neutral and language-neutral.

Localizability review

Discusses the second stage of creating a localized application, which involves identifying potential roadblocks to localization.

Localization

Discusses the final stage of creating a localized application, which involves customizing an application's user

interface for specific regions or cultures.

Culture-insensitive string operations

Describes how to use .NET methods and classes that are culture-sensitive by default to obtain culture-insensitive results.

Best practices for developing world-ready applications

Describes the best practices to follow for globalization, localization, and developing world-ready ASP.NET applications.

Reference

- System.Globalization namespace

  Contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings.

- System.Resources namespace

  Provides classes for creating, manipulating, and using resources.

- System.Text namespace

  Contains classes representing ASCII, ANSI, Unicode, and other character encodings.

- Resgen.exe (Resource File Generator)

  Describes how to use Resgen.exe to convert .txt files and XML-based resource format (.resx) files to common language runtime binary .resources files.

- Winres.exe (Windows Forms Resource Editor)

  Describes how to use Winres.exe to localize Windows Forms forms.

Extending Metadata Using Attributes

5/2/2018 • 2 minutes to read • Edit Online

The common language runtime allows you to add keyword-like descriptive declarations, called attributes, to annotate programming elements such as types, fields, methods, and properties. When you compile your code for the runtime, it is converted into Microsoft intermediate language (MSIL) and placed inside a portable executable (PE) file along with metadata generated by the compiler. Attributes allow you to place extra descriptive information into metadata that can be extracted using runtime reflection services. The compiler creates attributes when you declare instances of special classes that derive from System.Attribute.

The .NET Framework uses attributes for a variety of reasons and to address a number of issues. Attributes describe how to serialize data, specify characteristics that are used to enforce security, and limit optimizations by the just-in-time (JIT) compiler so the code remains easy to debug. Attributes can also record the name of a file or the author of code, or control the visibility of controls and members during forms development.

Related Topics

TITLE	DESCRIPTION
Applying Attributes	Describes how to apply an attribute to an element of your code.
Writing Custom Attributes	Describes how to design custom attribute classes.
Retrieving Information Stored in Attributes	Describes how to retrieve custom attributes for code that is loaded into the execution context.
Metadata and Self-Describing Components	Provides an overview of metadata and describes how it is implemented in a .NET Framework portable executable (PE) file.
How to: Load Assemblies into the Reflection-Only Context	Explains how to retrieve custom attribute information in the reflection-only context.

Reference

System.Attribute

Framework Design Guidelines

2/26/2019 • 2 minutes to read

This section provides guidelines for designing libraries that extend and interact with the .NET Framework. The goal is to help library designers ensure API consistency and ease of use by providing a unified programming model that is independent of the programming language used for development. We recommend that you follow these design guidelines when developing classes and components that extend the .NET Framework. Inconsistent library design adversely affects developer productivity and discourages adoption.

The guidelines are organized as simple recommendations prefixed with the terms `Do`, `Consider`, `Avoid`, and `Do not`. These guidelines are intended to help class library designers understand the trade-offs between different solutions. There might be situations where good library design requires that you violate these design guidelines. Such cases should be rare, and it is important that you have a clear and compelling reason for your decision.

These guidelines are excerpted from the book *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*, by Krzysztof Cwalina and Brad Abrams.

In This Section

Naming Guidelines
Provides guidelines for naming assemblies, namespaces, types, and members in class libraries.

Type Design Guidelines
Provides guidelines for using static and abstract classes, interfaces, enumerations, structures, and other types.

Member Design Guidelines
Provides guidelines for designing and using properties, methods, constructors, fields, events, operators, and parameters.

Designing for Extensibility
Discusses extensibility mechanisms such as subclassing, using events, virtual members, and callbacks, and explains how to choose the mechanisms that best meet your framework's requirements.

Design Guidelines for Exceptions
Describes design guidelines for designing, throwing, and catching exceptions.

Usage Guidelines
Describes guidelines for using common types such as arrays, attributes, and collections, supporting serialization, and overloading equality operators.

Common Design Patterns
Provides guidelines for choosing and implementing dependency properties.

Portions © 2005, 2009 Microsoft Corporation. All rights reserved.

Reprinted by permission of Pearson Education, Inc. from Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See also

- Overview
- Development Guide

XML Documents and Data

7/9/2019 • 3 minutes to read • Edit Online

The .NET Framework provides a comprehensive and integrated set of classes that enable you to build XML-aware apps easily. The classes in the following namespaces support parsing and writing XML, editing XML data in memory, data validation, and XSLT transformation.

- System.Xml
- System.Xml.XPath
- System.Xml.Xsl
- System.Xml.Schema
- System.Xml.Linq

For a full list, search for "System.Xml" on the .NET API browser.

The classes in these namespaces support World Wide Web Consortium (W3C) recommendations. For example:

- The System.Xml.XmlDocument class implements the W3C Document Object Model (DOM) Level 1 Core and DOM Level 2 Core recommendations.
- The System.Xml.XmlReader and System.Xml.XmlWriter classes support the W3C XML 1.0 and the Namespaces in XML recommendations.
- Schemas in the System.Xml.Schema.XmlSchemaSet class support the W3C XML Schema Part 1: Structures and XML Schema Part 2: Datatypes recommendations.
- Classes in the System.Xml.Xsl namespace support XSLT transformations that conform to the W3C XSLT 1.0 recommendation.

The XML classes in the .NET Framework provide these benefits:

- **Productivity.** LINQ to XML (C#) and LINQ to XML (Visual Basic) makes it easier to program with XML and provides a query experience that is similar to SQL.
- **Extensibility.** The XML classes in the .NET Framework are extensible through the use of abstract base classes and virtual methods. For example, you can create a derived class of the XmlUrlResolver class that stores the cache stream to the local disk.
- **Pluggable architecture.** The .NET Framework provides an architecture in which components can utilize one another, and data can be streamed between components. For example, a data store, such as an XPathDocument or XmlDocument object, can be transformed with the XslCompiledTransform class, and the output can then be streamed either into another store or returned as a stream from a web service.
- **Performance.** For better app performance, some of the XML classes in the .NET Framework support a streaming-based model with the following characteristics:
  - Minimal caching for forward-only, pull-model parsing (XmlReader).
  - Forward-only validation (XmlReader).
  - Cursor style navigation that minimizes node creation to a single virtual node while providing random access to the document (XPathNavigator).

For better performance whenever XSLT processing is required, you can use the XPathDocument class, which is an optimized, read-only store for XPath queries designed to work efficiently with the XslCompiledTransform class.

- **Integration with ADO.NET.** The XML classes and ADO.NET are tightly integrated to bring together relational data and XML. The DataSet class is an in-memory cache of data retrieved from a database. The DataSet class has the ability to read and write XML by using the XmlReader and XmlWriter classes, to persist its internal relational schema structure as XML schemas (XSD), and to infer the schema structure of an XML document.

In This Section

XML Processing Options Discusses options for processing XML data.

Processing XML Data In-Memory Discusses the three models for processing XML data in-memory: LINQ to XML (C#) and LINQ to XML (Visual Basic), the XmlDocument class (based on the W3C Document Object Model), and the XPathDocument class (based on the XPath data model).

XSLT Transformations
Describes how to use the XSLT processor.

XML Schema Object Model (SOM)
Describes the classes used for building and manipulating XML Schemas (XSD) by providing an XmlSchema class to load and edit a schema.

XML Integration with Relational Data and ADO.NET
Describes how the .NET Framework enables real-time, synchronous access to both the relational and hierarchical representations of data through the DataSet object and the XmlDataDocument object.

Managing Namespaces in an XML Document
Describes how the XmlNamespaceManager class is used to store and maintain namespace information.

Type Support in the System.Xml Classes
Describes how XML data types map to CLR types, how to convert XML data types, and other type support features in the System.Xml classes.

Related Sections

ADO.NET
Provides information on how to access data using ADO.NET.

Security
Provides an overview of the .NET Framework security system.

Security in .NET

6/2/2018 • 2 minutes to read • Edit Online

The common language runtime and the .NET provide many useful classes and services that enable developers to easily write secure code and enable system administrators to customize the permissions granted to code so that it can access protected resources. In addition, the runtime and the .NET provide useful classes and services that facilitate the use of cryptography and role-based security.

In This Section

Key Security Concepts
Provides an overview of common language runtime security features. This section is of interest to developers and system administrators.

Role-Based Security
Describes how to interact with role-based security in your code. This section is of interest to developers.

Cryptography Model
Provides an overview of cryptographic services provided by .NET. This section is of interest to developers.

Secure Coding Guidelines
Describes some of the best practices for creating reliable .NET applications. This section is of interest to developers.

Secure Coding Guidelines for Unmanaged Code
Describes some of the best practices and security concerns when calling unmanaged code.

Windows Identity Foundation
Describes how you can implement claims-based identity in your applications.

Security Changes Describes important changes to the .NET Framework security system.

Related Sections

Development Guide
Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.

Serialization in .NET

5/15/2019 • 2 minutes to read • Edit Online

Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be easily stored and transferred.

.NET features two serialization technologies:

- Binary serialization preserves type fidelity, which is useful for preserving the state of an object between different invocations of an application. For example, you can share an object between different applications by serializing it to the Clipboard. You can serialize an object to a stream, to a disk, to memory, over the network, and so forth. Remoting uses serialization to pass objects "by value" from one computer or application domain to another.
- XML serialization serializes only public properties and fields and does not preserve type fidelity. This is useful when you want to provide or consume data without restricting the application that uses the data. Because XML is an open standard, it is an attractive choice for sharing data across the Web. SOAP is likewise an open standard, which makes it an attractive choice.

In This Section

Serialization How-to Topics
Lists links to How-to topics contained in this section.

Binary Serialization
Describes the binary serialization mechanism that is included with the common language runtime.

XML and SOAP Serialization
Describes the XML and SOAP serialization mechanism that is included with the common language runtime.

Serialization Tools
These tools help develop serialization code.

Serialization Samples
The samples demonstrate how to do serialization.

Reference

System.Runtime.Serialization Contains classes that can be used for serializing and deserializing objects.

System.Xml.Serialization
Contains classes that can be used to serialize objects into XML format documents or streams.

Developing for Multiple Platforms with the .NET Framework

5/15/2019 • 3 minutes to read • Edit Online

You can develop apps for both Microsoft and non-Microsoft platforms by using the .NET Framework and Visual Studio.

Options for cross-platform development

> **IMPORTANT**
>
> Because Portable Class Library projects target only a very specific subset of .NET implementations, we strongly discourage their use in new application development. The recommended replacement is a .NET Standard library, which targets all .NET implementations that support a specific version of the .NET Standard. For more information, see .NET Standard.

To develop for multiple platforms, you can share source code or binaries, and you can make calls between .NET Framework code and Windows Runtime APIs.

IF YOU WANT TO...	USE...
Share source code between Windows Phone 8.1 and Windows 8.1 apps	**Shared projects** (Universal Apps template in Visual Studio 2013, Update 2). - Currently no Visual Basic support. - You can separate platform-specific code by using #`if` statements. For details, see: - Start coding - Using Visual Studio to build Universal XAML Apps (blog post) - Using Visual Studio to Build XAML Converged Apps (video)

IF YOU WANT TO...	USE...
Share binaries between apps that target different platforms	**Portable Class Library projects** for code that is platform-agnostic. - This approach is typically used for code that implements business logic. - You can use Visual Basic or C#. - API support varies by platform. - Portable Class Library projects that target Windows 8.1 and Windows Phone 8.1 support Windows Runtime APIs and XAML. These features aren't available in older versions of the Portable Class Library. - If needed, you can abstract out platform-specific code by using interfaces or abstract classes. For details, see: - Portable Class Library - How to Make Portable Class Libraries Work for You (blog post) - Using Portable Class Library with MVVM - App Resources for Libraries That Target Multiple Platforms - .NET Portability Analyzer (Visual Studio extension)
Share source code between apps for platforms other than Windows 8.1 and Windows Phone 8.1	**Add as link** feature. - This approach is suitable for app logic that's common to both apps but not portable, for some reason. You can use this feature for C# or Visual Basic code. For example, Windows Phone 8 and Windows 8 share Windows Runtime APIs, but Portable Class Libraries do not support Windows Runtime for those platforms. You can use `Add as link` to share common Windows Runtime code between a Windows Phone 8 app and a Windows Store app that targets Windows 8. For details, see: - Share code with Add as Link - How to: Add Existing Items to a Project
Write Windows Store apps using the .NET Framework or call Windows Runtime APIs from .NET Framework code	**Windows Runtime APIs** from your .NET Framework C# or Visual Basic code, and use the .NET Framework to create Windows Store apps. You should be aware of API differences between the two platforms. However, there are classes to help you work with those differences. For details, see: - .NET Framework Support for Windows Store Apps and Windows Runtime - Passing a URI to the Windows Runtime - WindowsRuntimeStreamExtensions - WindowsRuntimeSystemExtensions

IF YOU WANT TO...	USE...
Build .NET Framework apps for non-Microsoft platforms	**Portable Class Library reference assemblies** in the .NET Framework, and a Visual Studio extension or third-party tool such as Xamarin. For details, see: - Portable Class Library now available on all platforms. (blog post) - Xamarin documentation
Use JavaScript and HTML for cross-platform development	**Universal App templates** in Visual Studio 2013, Update 2 to develop against Windows Runtime APIs for Windows 8.1 and Windows Phone 8.1. Currently, you can't use JavaScript and HTML with .NET Framework APIs to develop cross-platform apps. For details, see: - JavaScript Project Templates - Porting a Windows Runtime app using JavaScript to Windows Phone

.NET Core Guide

4/4/2019 • 2 minutes to read • Edit Online

.NET Core is an open-source, general-purpose development platform maintained by Microsoft and the .NET community on GitHub. It's cross-platform (supporting Windows, macOS, and Linux) and can be used to build device, cloud, and IoT applications.

See About .NET Core to learn more about .NET Core, including its characteristics, supported languages and frameworks, and key APIs.

Check out .NET Core Tutorials to learn how to create a simple .NET Core application. It only takes a few minutes to get your first app up and running. If you want to try .NET Core in your browser, look at the Numbers in C# online tutorial.

Download .NET Core 2.2

Download the .NET Core 2.2 SDK to try .NET Core on your Windows, macOS, or Linux machine. Visit dotnet/core if you prefer to use Docker containers.

All .NET Core versions are available at .NET Core Downloads if you're looking for another .NET Core version.

.NET Core 2.2

The latest version is .NET Core 2.2. New features include: framework-dependent deployments, startup hooks, AAD authentication with Azure SQL, and support for Windows ARM32.

Create your first application

After installing the .NET Core SDK, open a command prompt. Type the following `dotnet` commands to create and run a C# application.

```
dotnet new console
dotnet run
```

You should see the following output:

```
Hello World!
```

Support

.NET Core is supported by Microsoft, on Windows, macOS and Linux. It's updated for security and quality several times a year, typically monthly.

.NET Core binary distributions are built and tested on Microsoft-maintained servers in Azure and supported just like any Microsoft product.

Red Hat supports .NET Core on Red Hat Enterprise Linux (RHEL). Red Hat builds .NET Core from source and makes it available in the Red Hat Software Collections. Red Hat and Microsoft collaborate to ensure that .NET Core works well on RHEL.

About .NET Core

6/27/2019 • 6 minutes to read • Edit Online

.NET Core has the following characteristics:

- **Cross-platform:** Runs on Windows, macOS and Linux operating systems.
- **Consistent across architectures:** Runs your code with the same behavior on multiple architectures, including x64, x86, and ARM.
- **Command-line tools:** Includes easy-to-use command-line tools that can be used for local development and in continuous-integration scenarios.
- **Flexible deployment:** Can be included in your app or installed side-by-side (user-wide or system-wide installations). Can be used with Docker containers.
- **Compatible:** .NET Core is compatible with .NET Framework, Xamarin and Mono, via .NET Standard.
- **Open source:** The .NET Core platform is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.
- **Supported by Microsoft:** .NET Core is supported by Microsoft, per .NET Core Support.

Languages

C#, Visual Basic, and F# languages can be used to write applications and libraries for .NET Core. These languages are or can be integrated into your favorite text editors and IDEs, including Visual Studio, Visual Studio Code, Sublime Text and Vim. This integration is provided, in part, by the good folks from the OmniSharp and Ionide projects.

APIs

.NET Core exposes APIs for many scenarios, a few of which follow:

- Primitive types, such as bool and int.
- Collections, such as System.Collections.Generic.List<T> and System.Collections.Generic.Dictionary<TKey,TValue>.
- Utility types, such as System.Net.Http.HttpClient, and System.IO.FileStream.
- Data types, such as System.Data.DataSet, and DbSet.
- High performance types, such as System.Numerics.Vector and Pipelines.

.NET Core provides compatibility with .NET Framework and Mono APIs by implementing the .NET Standard specification.

Frameworks

Multiple frameworks have been built on top of .NET Core:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Tizen

Composition

.NET Core is composed of the following parts:

- The .NET Core runtime, which provides a type system, assembly loading, a garbage collector, native interop and other basic services. .NET Core framework libraries provide primitive data types, app composition types and fundamental utilities.
- The ASP.NET runtime, which provides a framework for building modern cloud based internet connected applications, such as web apps, IoT apps and mobile backends.
- The .NET Core CLI tools and language compilers (Roslyn and F#) that enable the .NET Core developer experience.
- The dotnet tool, which is used to launch .NET Core apps and CLI tools. It selects the runtime and hosts the runtime, provides an assembly loading policy and launches apps and tools.

These components are distributed in the following ways:

- .NET Core Runtime -- includes the .NET Core runtime and framework libraries.
- ASP.NET Core Runtime -- includes ASP.NET Core and .NET Core runtime and framework libraries.
- .NET Core SDK -- includes the .NET CLI Tools, ASP.NET Core runtime, and .NET Core runtime and framework.

Open source

.NET Core is open source (MIT license) and was contributed to the .NET Foundation by Microsoft in 2014. It is now one of the most active .NET Foundation projects. It can be freely adopted by individuals and companies, including for personal, academic or commercial purposes. Multiple companies use .NET Core as part of apps, tools, new platforms and hosting services. Some of these companies make significant contributions to .NET Core on GitHub and provide guidance on the product direction as part of the .NET Foundation Technical Steering Group.

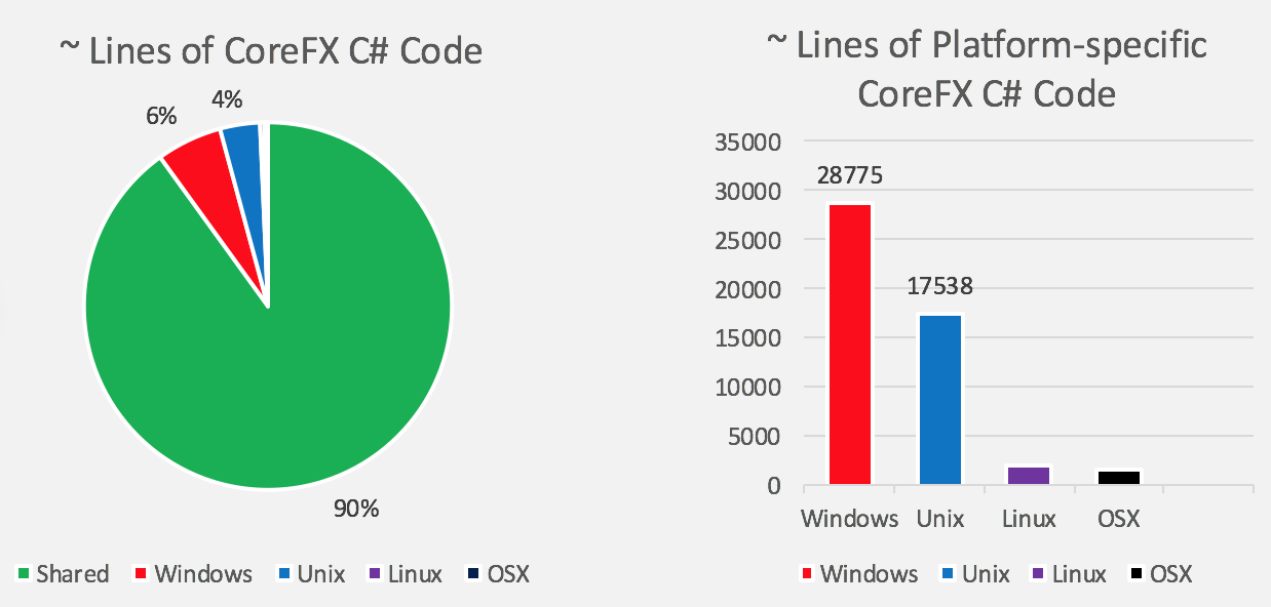
Designed for adaptability

.NET Core has been built as a very similar but unique product relative to other .NET products. It has been designed to enable broad adaptability to new platforms and workloads. It has several OS and CPU ports available and may be ported to many more.

The product is broken into several pieces, enabling the various parts to be adapted to new platforms at different times. The runtime and platform-specific foundational libraries must be ported as a unit. Platform-agnostic libraries should work as-is on all platforms, by construction. There is a project bias to reducing platform-specific implementations to increase developer efficiency, preferring platform-neutral C# code whenever an algorithm or API can be implemented in-full or in-part that way.

People commonly ask how .NET Core is implemented in order to support multiple operating systems. They typically ask if there are separate implementations or if conditional compilation is used. It's both, with a strong bias towards conditional compilation.

You can see in the following chart that the vast majority of CoreFX is platform-neutral code that is shared across all platforms. Platform-neutral code can be implemented as a single portable assembly that is used on all platforms.



Windows and Unix implementations are similar in size. Windows has a larger implementation since CoreFX implements some Windows-only features, such as Microsoft.Win32.Registry but does not yet implement many Unix-only concepts. You will also see that the majority of the Linux and macOS implementations are shared across a Unix implementation, while the Linux- and macOS-specific implementations are roughly similar in size.

There are a mix of platform-specific and platform-neutral libraries in .NET Core. You can see the pattern in a few examples:

- CoreCLR is platform-specific. It builds on top of OS subsystems, like the memory manager and thread scheduler.
- System.IO and System.Security.Cryptography.Algorithms are platform-specific, given that storage and cryptography APIs are different on each OS.
- System.Collections and System.Linq are platform-neutral, given that they create and operate over data structures.

Comparisons to other .NET implementations

It is perhaps easiest to understand the size and shape of .NET Core by comparing it to existing .NET implementations.

Comparison with .NET Framework

.NET was first announced by Microsoft in 2000 and then evolved from there. The .NET Framework has been the primary .NET implementation produced by Microsoft during that nearly two decade period.

The major differences between .NET Core and the .NET Framework:

- **App-models** -- .NET Core does not support all the .NET Framework app-models. In particular, it doesn't support ASP.NET Web Forms and ASP.NET MVC, but it supports ASP.NET Core MVC. It was announced that .NET Core 3 will support WPF and Windows Forms.
- **APIs** -- .NET Core contains a large subset of .NET Framework Base Class Library, with a different factoring (assembly names are different; members exposed on types differ in key cases). These differences require changes to port source to .NET Core in some cases (see microsoft/dotnet-apiport). .NET Core implements the .NET Standard API specification.
- **Subsystems** -- .NET Core implements a subset of the subsystems in the .NET Framework, with the goal of a simpler implementation and programming model. For example, Code Access Security (CAS) is not supported, while reflection is supported.
- **Platforms** -- The .NET Framework supports Windows and Windows Server while .NET Core also supports macOS and Linux.

- **Open Source** -- .NET Core is open source, while a read-only subset of the .NET Framework is open source.

While .NET Core is unique and has significant differences to the .NET Framework and other .NET implementations, it is straightforward to share code between these implementations, using either source or binary sharing techniques.

Because .NET Core supports side-by-side installation and its runtime is completely independent of the .NET Framework, it can be installed on machines with .NET Framework installed without any issues.

Comparison with Mono

Mono is the original cross-platform and open source .NET implementation, first shipping in 2004. It can be thought of as a community clone of the .NET Framework. The Mono project team relied on the open .NET standards (notably ECMA 335) published by Microsoft in order to provide a compatible implementation.

The major differences between .NET Core and Mono:

- **App-models** -- Mono supports a subset of the .NET Framework app-models (for example, Windows Forms) and some additional ones (for example, Xamarin.iOS) through the Xamarin product. .NET Core doesn't support these.
- **APIs** -- Mono supports a large subset of the .NET Framework APIs, using the same assembly names and factoring.
- **Platforms** -- Mono supports many platforms and CPUs.
- **Open Source** -- Mono and .NET Core both use the MIT license and are .NET Foundation projects.
- **Focus** -- The primary focus of Mono in recent years is mobile platforms, while .NET Core is focused on cloud and desktop workloads.

Get started with .NET Core

7/11/2019 • 2 minutes to read • Edit Online

This article provides information on getting started with .NET Core. .NET Core can be installed on Windows, Linux, and macOS. You can code in your favorite text editor and produce cross-platform libraries and applications.

If you're unsure what .NET Core is, or how it relates to other .NET technologies, start with the What is .NET overview. Put simply, .NET Core is an open-source, cross-platform implementation of .NET.

Create an application

First, download and install the .NET Core SDK on your computer.

Next, open a terminal such as **PowerShell**, **Command Prompt**, or **bash**. Type the following `dotnet` commands to create and run a C# application.

```
dotnet new console --output sample1
dotnet run --project sample1
```

You should see the following output:

```
Hello World!
```

Congratulations! You've created a simple .NET Core application. You can also use Visual Studio Code, Visual Studio (Windows only), or Visual Studio for Mac (macOS only), to create a .NET Core application.

Tutorials

You can get started developing .NET Core applications by following these step-by-step tutorials.

- Windows
- Linux
- macOS

- Build a C# "Hello World" Application with .NET Core in Visual Studio 2017.
- Build a C# class library with .NET Core in Visual Studio 2017.
- Build a Visual Basic "Hello World" application with .NET Core in Visual Studio 2017.
- Build a class library with Visual Basic and .NET Core in Visual Studio 2017.
- Watch a video on how to install and use Visual Studio Code and .NET Core.
- Watch a video on how to install and use Visual Studio 2017 and .NET Core.
- Getting started with .NET Core using the command-line.

See the Prerequisites for Windows development article for a list of the supported Windows versions.

Get started with C# and Visual Studio Code

6/26/2019 • 4 minutes to read • Edit Online

.NET Core gives you a fast and modular platform for creating applications that run on Windows, Linux, and macOS. Use Visual Studio Code with the C# extension to get a powerful editing experience with full support for C# IntelliSense (smart code completion) and debugging.

Prerequisites

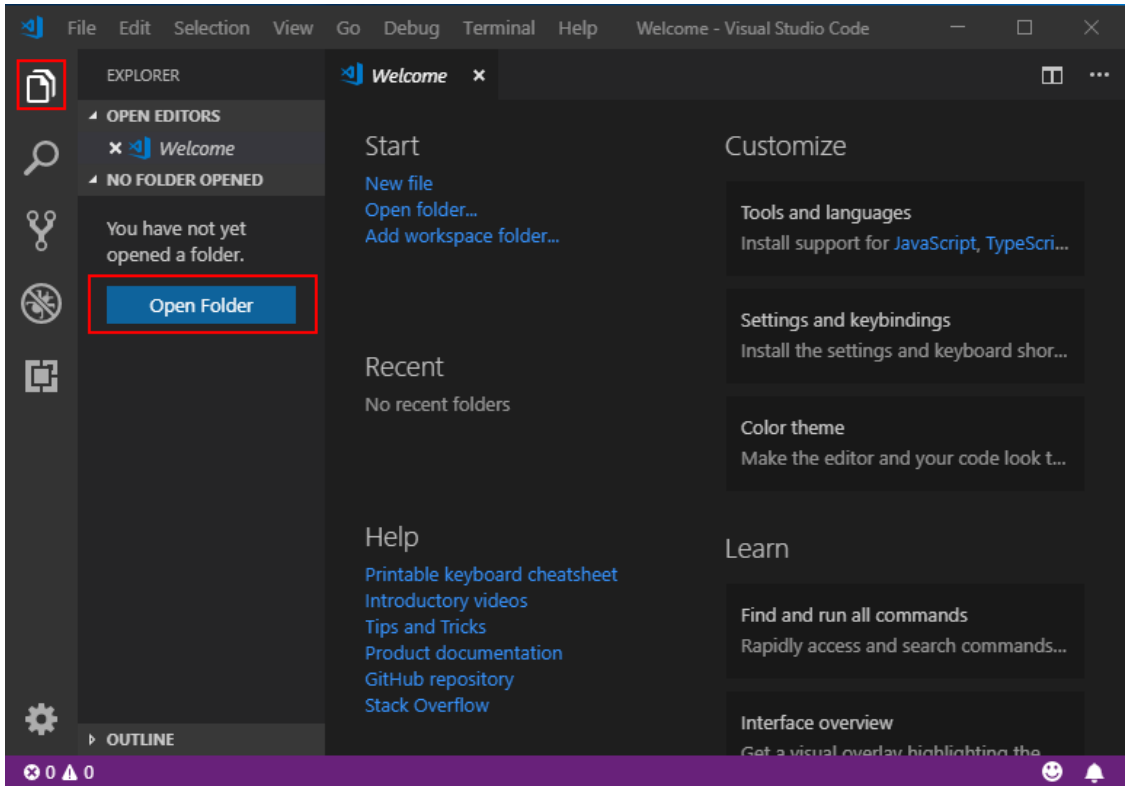
1. Install Visual Studio Code.
2. Install the .NET Core SDK.
3. Install the C# extension for Visual Studio Code. For more information about how to install extensions on Visual Studio Code, see VS Code Extension Marketplace.

Hello World

Let's get started with a simple "Hello World" program on .NET Core:

1. Open a project:

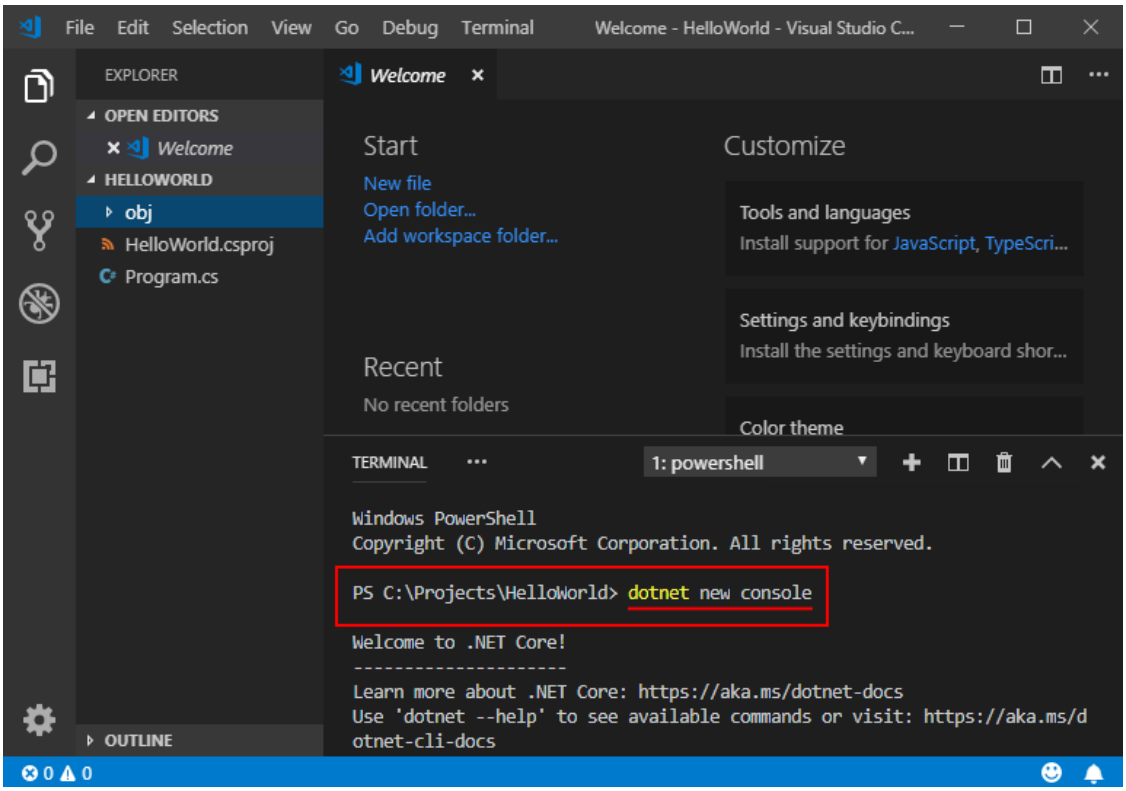
   - Open Visual Studio Code.
   - Click on the Explorer icon on the left menu and then click **Open Folder**.
   - Select **File** > **Open Folder** from the main menu to open the folder you want your C# project to be in and click **Select Folder**. For our example, we're creating a folder for our project named *HelloWorld*.



2. Initialize a C# project:

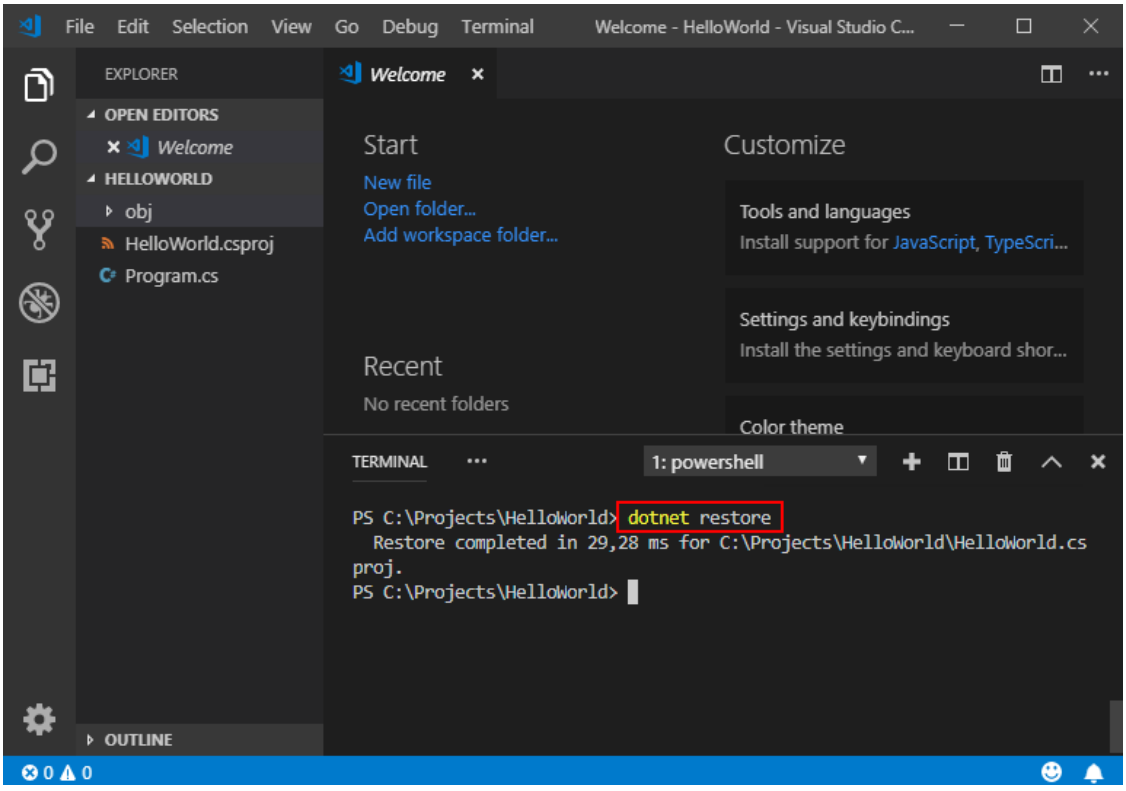
   - Open the Integrated Terminal from Visual Studio Code by selecting **View** > **Integrated Terminal** from the main menu.

- In the terminal window, type `dotnet new console`.
- This command creates a `Program.cs` file in your folder with a simple "Hello World" program already written, along with a C# project file named `HelloWorld.csproj`.



3. Resolve the build assets:

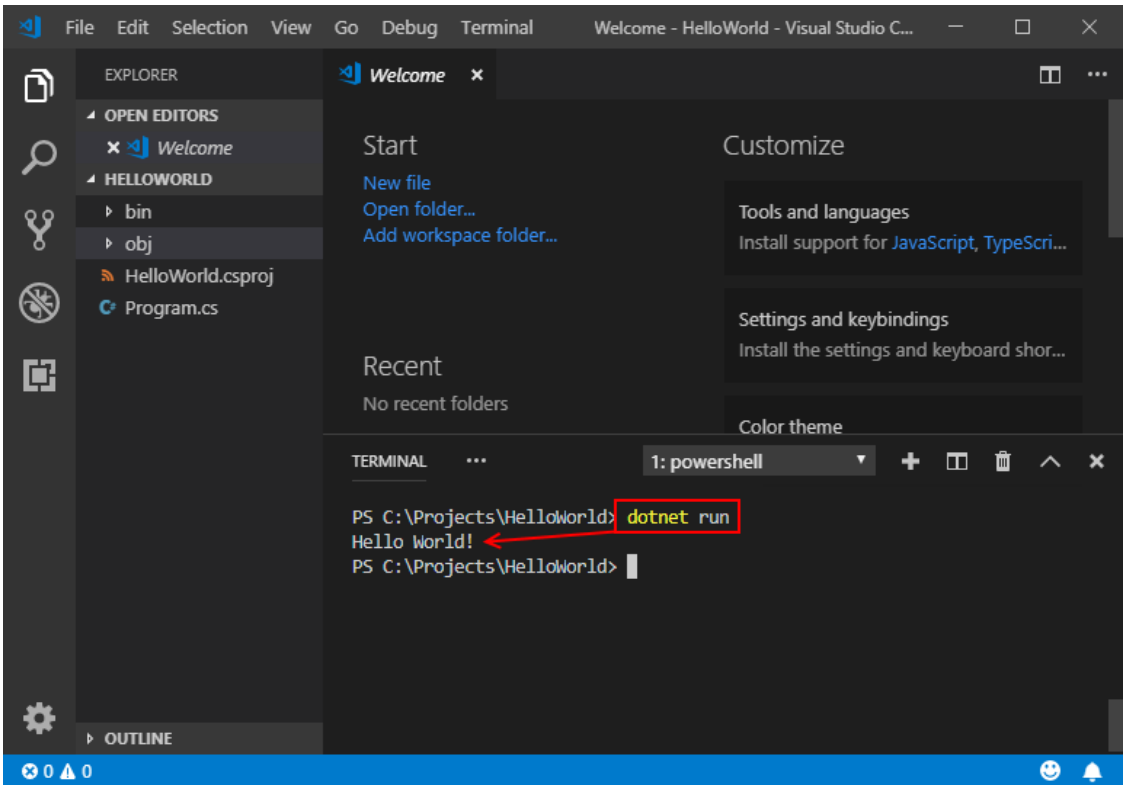
- For **.NET Core 1.x**, type `dotnet restore`. Running `dotnet restore` gives you access to the required .NET Core packages that are needed to build your project.



> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

4. Run the "Hello World" program:

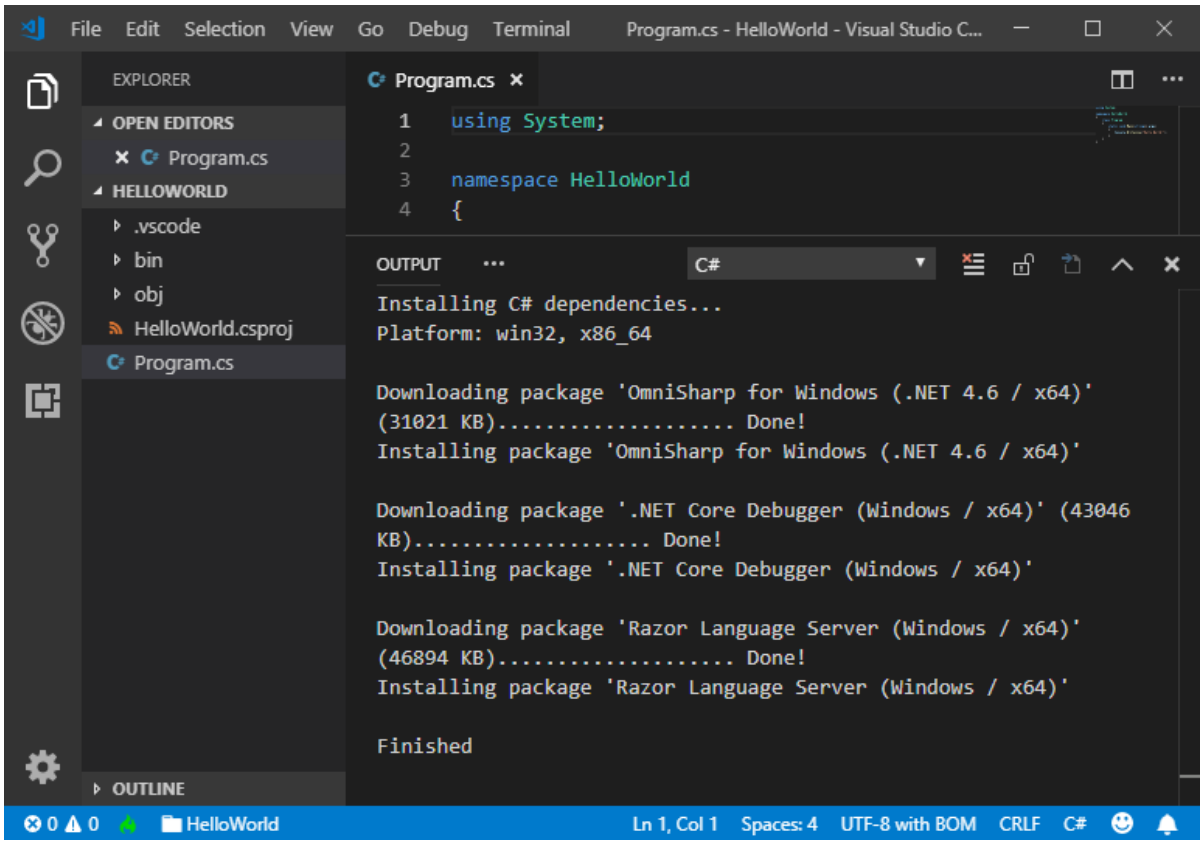
   - Type `dotnet run`.



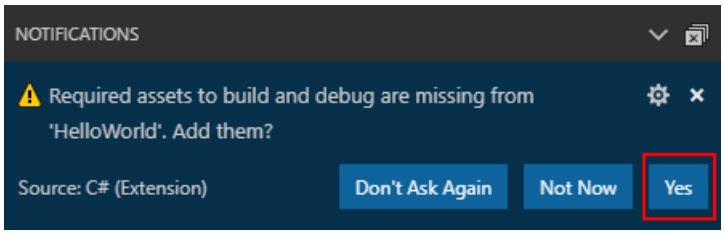
You can also watch a short video tutorial for further setup help on Windows, macOS, or Linux.

Debug

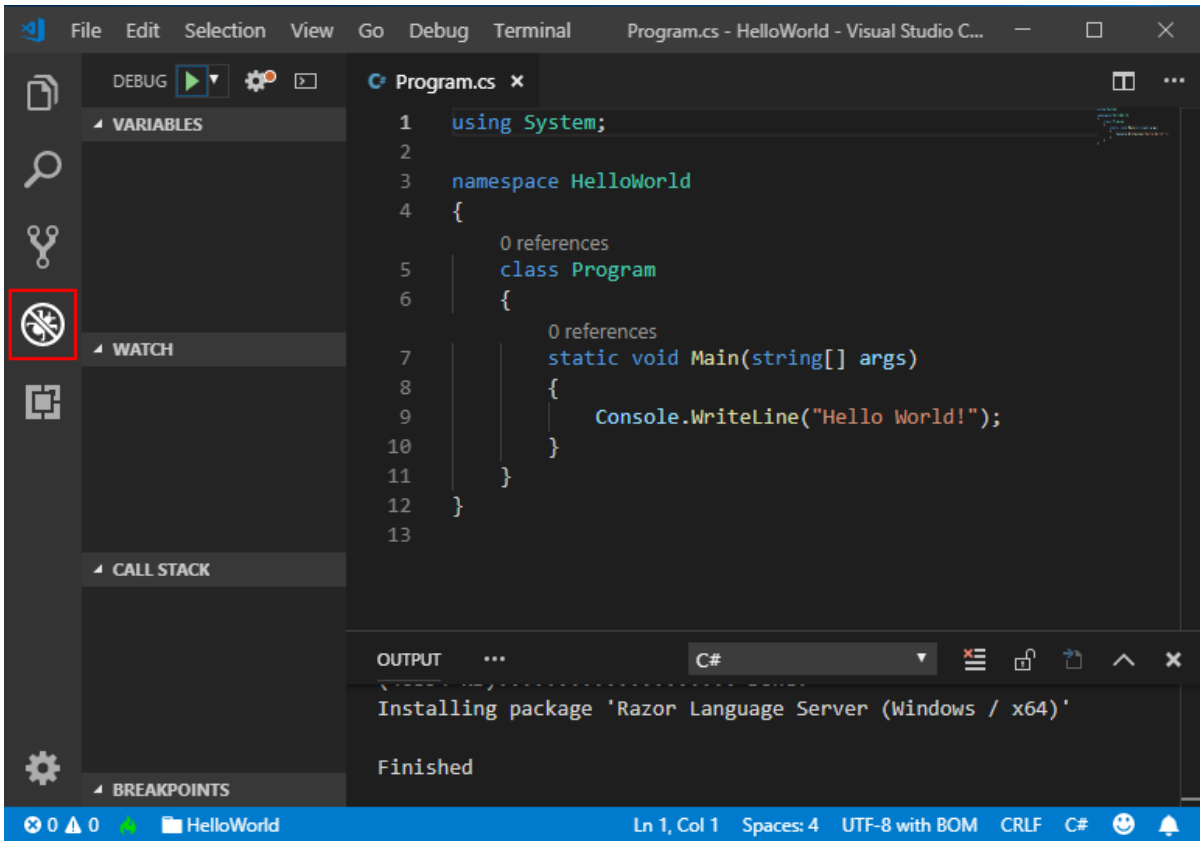
1. Open *Program.cs* by clicking on it. The first time you open a C# file in Visual Studio Code, OmniSharp loads in the editor.



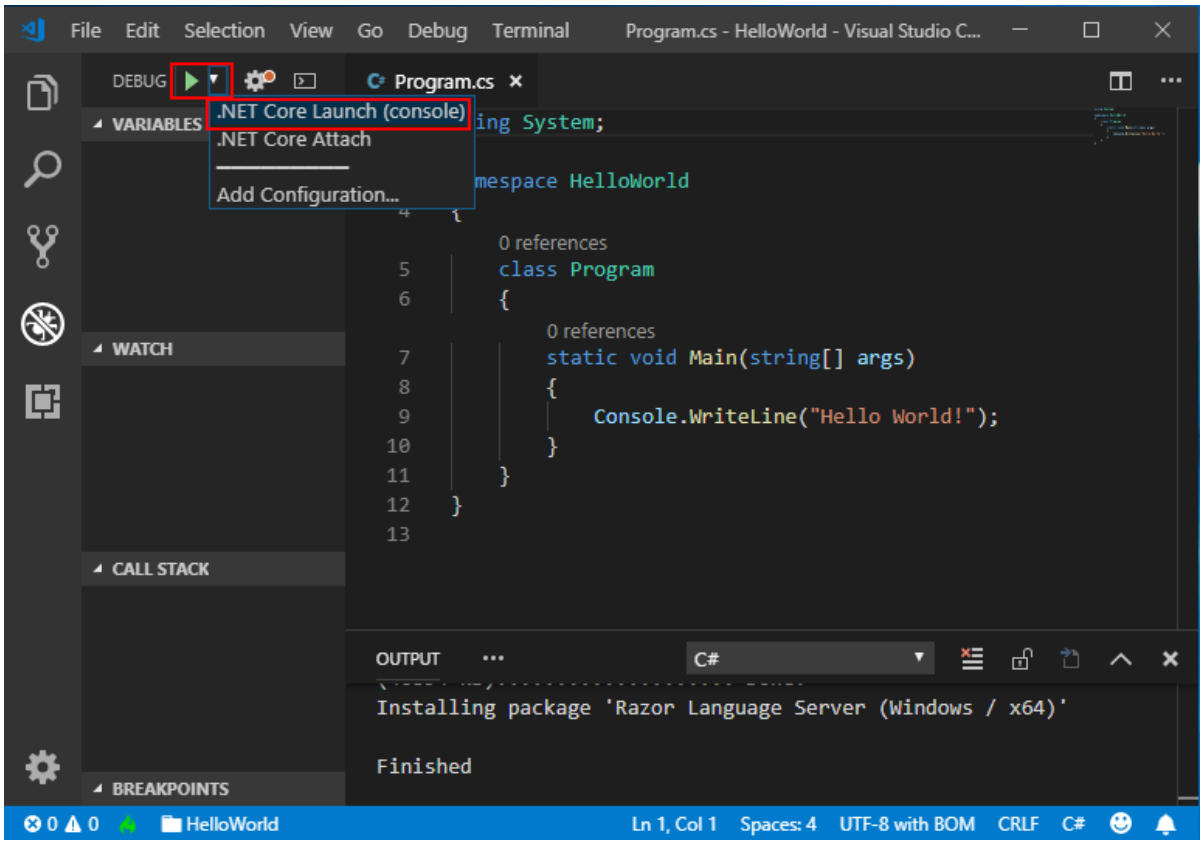
2. Visual Studio Code should prompt you to add the missing assets to build and debug your app. Select **Yes**.



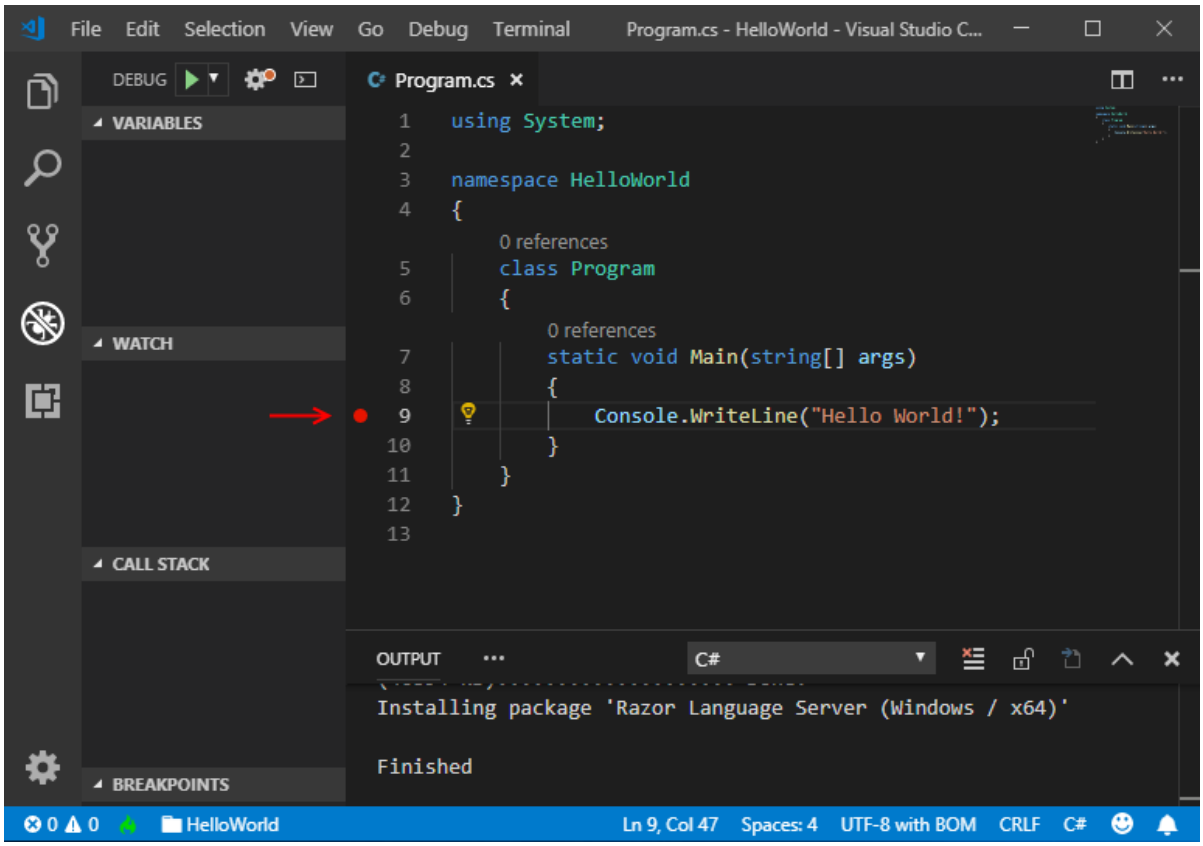
3. To open the Debug view, click on the Debugging icon on the left side menu.



4. Locate the green arrow at the top of the pane. Make sure the drop-down next to it has `.NET Core Launch (console)` selected.



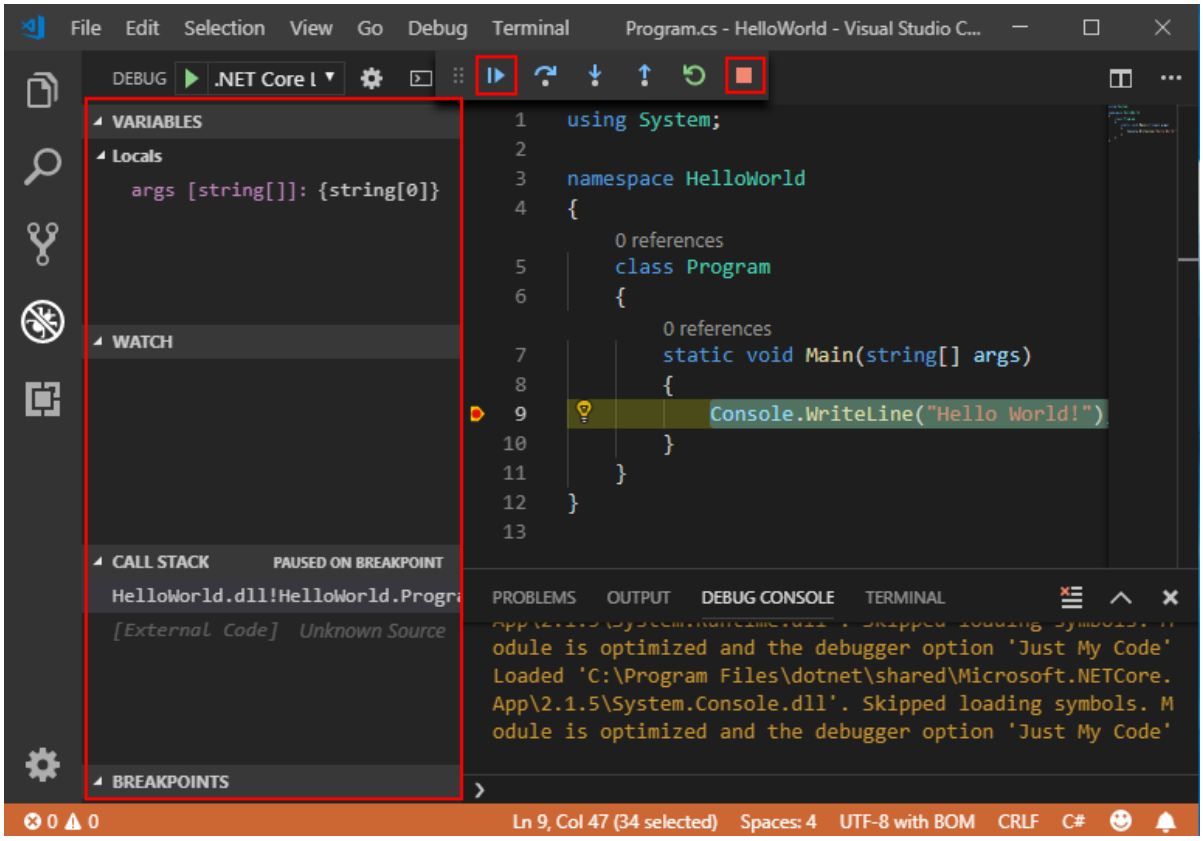
5. Add a breakpoint to your project by clicking on the **editor margin**, which is the space on the left of the line numbers in the editor, next to line 9, or move the text cursor onto line 9 in the editor and press F9.



6. To start debugging, select F5 or the green arrow. The debugger stops execution of your program when it reaches the breakpoint you set in the previous step.

   - While debugging, you can view your local variables in the top left pane or use the debug console.

7. Select the blue arrow at the top to continue debugging, or select the red square at the top to stop.



> **TIP**
>
> For more information and troubleshooting tips on .NET Core debugging with OmniSharp in Visual Studio Code, see Instructions for setting up the .NET Core debugger.

Add a class

1. To add a new class right-click in the VSCode Explorer and select **New File**. This adds a new file to the folder you have open in VSCode.
2. Name your file `MyClass.cs`. You must save it with a `.cs` extension at the end for it to be recognized as a csharp file.
3. Add the code below to create your first class. Make sure to include the correct namespace so you can reference it from your `Program.cs` file.

```
using System;

namespace HelloWorld
{
    public class MyClass
    {
        public string ReturnMessage()
        {
            return "Happy coding!";
        }
    }
}
```

4. Call your new class from your main method in `Program.cs` by adding the code below.

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            MyClass c1 = new MyClass();
            Console.WriteLine($"Hello World! {c1.ReturnMessage()}");
        }
    }
}
```

5. Save your changes and run your program again. The new message should appear with the appended string.

```
> dotnet run
Hello World! Happy coding!
```

FAQ

I'm missing required assets to build and debug C# in Visual Studio Code. My debugger says "No Configuration."

The Visual Studio Code C# extension can generate assets to build and debug for you. Visual Studio Code prompts you to generate these assets when you first open a C# project. If you didn't generate assets then, you can still run this command by opening the Command Palette (**View > Command Palette**) and typing ">.NET: Generate Assets for Build and Debug". Selecting this generates the .vscode, launch.json, and tasks.json configuration files that you need.

See also

- Setting up Visual Studio Code
- Debugging in Visual Studio Code

Build a C# Hello World application with the .NET Core SDK in Visual Studio 2017

8/20/2019 • 3 minutes to read • Edit Online

This topic provides a step-by-step introduction to building, debugging, and publishing a simple .NET Core console application using C# in Visual Studio 2017. Visual Studio 2017 provides a full-featured development environment for building .NET Core applications. As long as the application doesn't have platform-specific dependencies, the application can run on any platform that .NET Core targets and on any system that has .NET Core installed.

Prerequisites

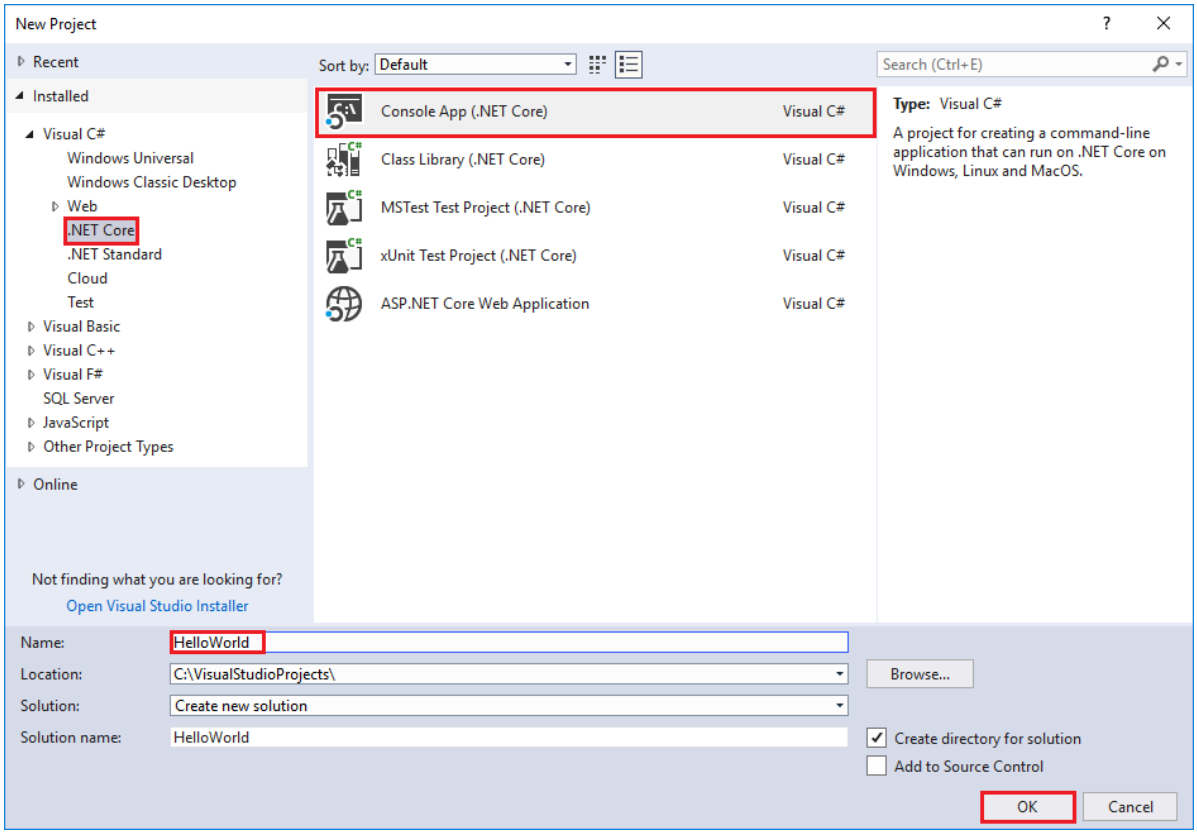
Visual Studio 2017 with the ".NET Core cross-platform development" workload installed. You can develop your app with either .NET Core 1.1 or .NET Core 2.0.

For more information, see the Prerequisites for .NET Core on Windows topic.

A simple Hello World application

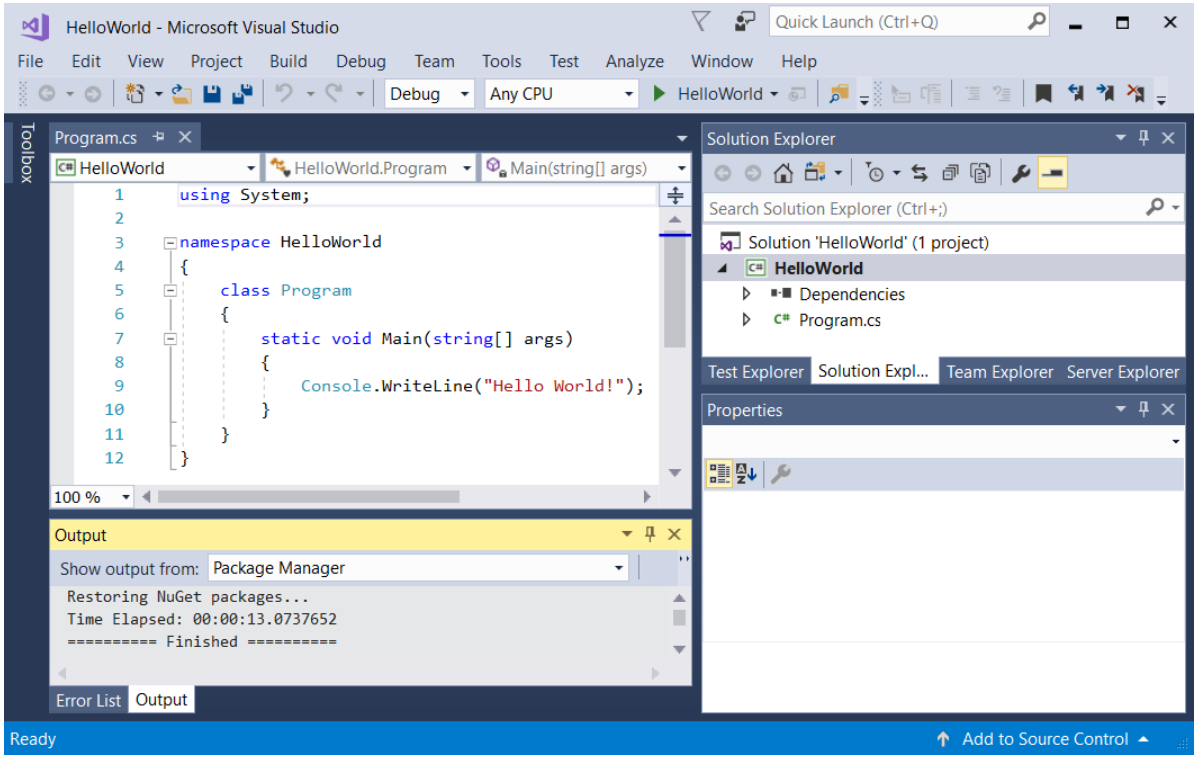
Begin by creating a simple "Hello World" console application. Follow these steps:

1. Launch Visual Studio 2017. Select **File** > **New** > **Project** from the menu bar. In the *New Project** dialog, select the **Visual C#** node followed by the **.NET Core** node. Then select the **Console App (.NET Core)** project template. In the **Name** text box, type "HelloWorld". Select the **OK** button.



2. Visual Studio uses the template to create your project. The C# Console Application template for .NET Core automatically defines a class, `Program`, with a single method, `Main`, that takes a String array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime when it launches the application. Any command-line arguments supplied when the application is launched are

available in the *args* array.



The template creates a simple "Hello World" application. It calls the Console.WriteLine(String) method to display the literal string "Hello World!" in the console window. By selecting the **HelloWorld** button with the green arrow on the toolbar, you can run the program in Debug mode. If you do, the console window is visible for only a brief time interval before it closes. This occurs because the `Main` method terminates and the application ends as soon as the single statement in the `Main` method executes.

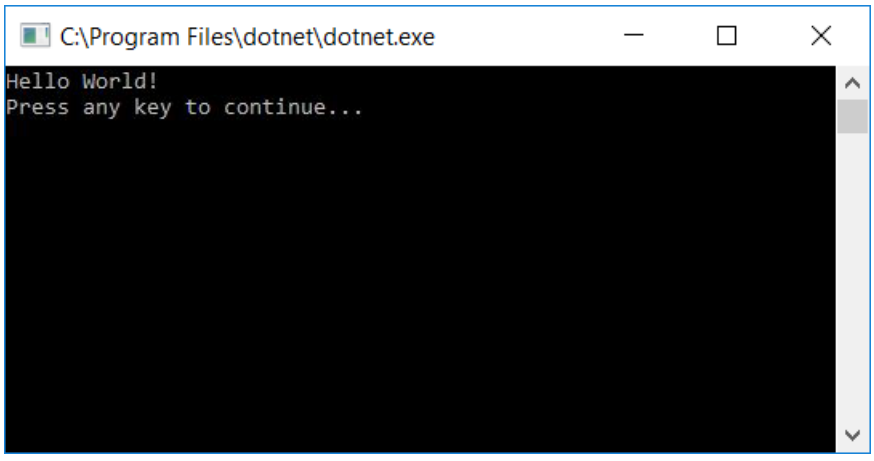
3. To cause the application to pause before it closes the console window, add the following code immediately after the call to the Console.WriteLine(String) method:

```
Console.Write("Press any key to continue...");
Console.ReadKey(true);
```

This code prompts the user to press any key and then pauses the program until a key is pressed.

4. On the menu bar, select **Build** > **Build Solution**. This compiles your program into an intermediate language (IL) that's converted into binary code by a just-in-time (JIT) compiler.

5. Run the program by selecting the **HelloWorld** button with the green arrow on the toolbar.



6. Press any key to close the console window.

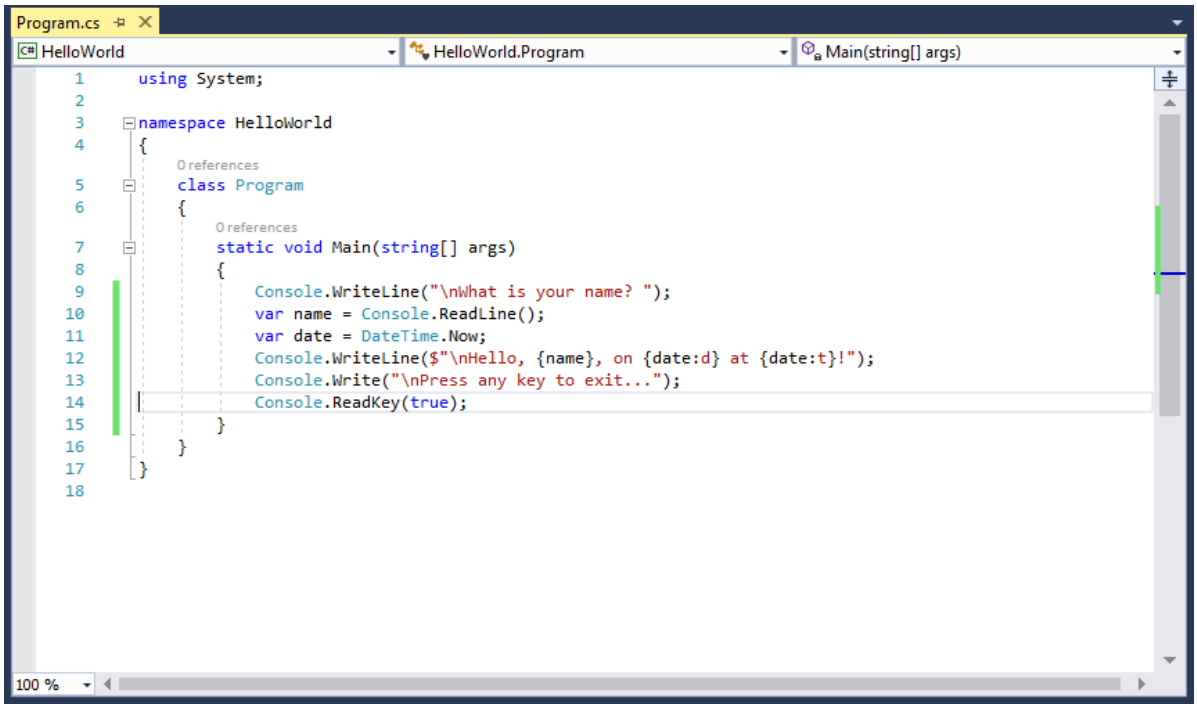
Enhancing the Hello World application

Enhance your application to prompt the user for their name and display it along with the date and time. To modify and test the program, do the following:

1. Enter the following C# code in the code window immediately after the opening bracket that follows the `static void Main(string[] args)` line and before the first closing bracket:

   ```
   Console.WriteLine("\nWhat is your name? ");
   var name = Console.ReadLine();
   var date = DateTime.Now;
   Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
   Console.Write("\nPress any key to exit...");
   Console.ReadKey(true);
   ```

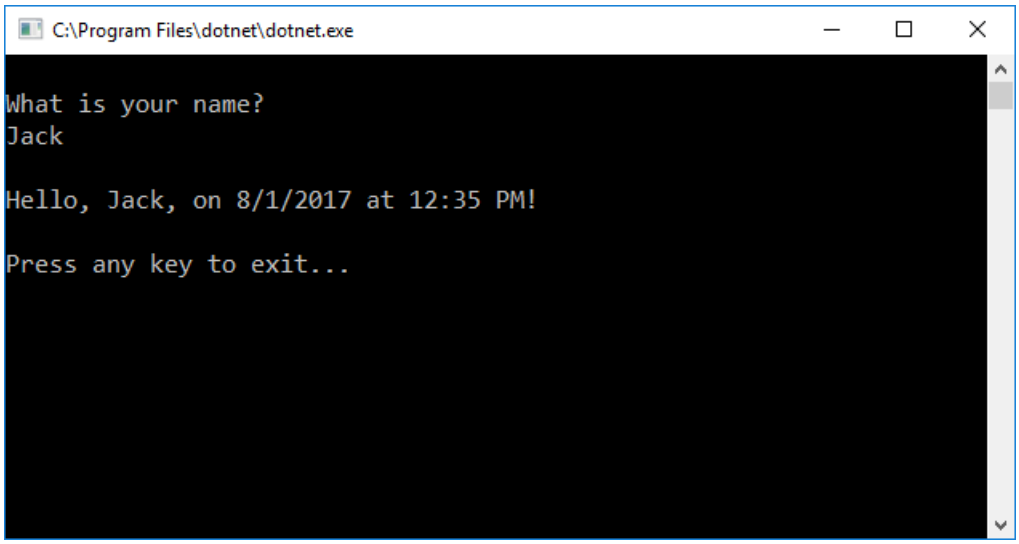
   This code replaces the contents of the `Main` method.

   This code displays "What is your name?" in the console window and waits until the user enters a string followed by the Enter key. It stores this string into a variable named `name`. It also retrieves the value of the DateTime.Now property, which contains the current local time, and assigns it to a variable named `date`. Finally, it uses an interpolated string to display these values in the console window.

2. Compile the program by choosing **Build** > **Build Solution**.

3. Run the program in Debug mode in Visual Studio by selecting the green arrow on the toolbar, pressing F5, or choosing the **Debug** > **Start Debugging** menu item. Respond to the prompt by entering a name and pressing the Enter key.



4. Press any key to close the console window.

You've created and run your application. To develop a professional application, take some additional steps to make your application ready for release:

- For information on debugging your application, see Debug your .NET Core Hello World application using Visual Studio 2017.
- For information on developing and publishing a distributable version of your application, see Publish your .NET Core Hello World application with Visual Studio 2017.

Related topics

Instead of a console application, you can also build a class library with .NET Core and Visual Studio 2017. For a step-by-step introduction, see Building a class library with C# and .NET Core in Visual Studio 2017.

You can also develop a .NET Core console app on Mac, Linux, and Windows by using Visual Studio Code, a downloadable code editor. For a step-by-step tutorial, see Getting Started with Visual Studio Code.

Build a Visual Basic Hello World application with the .NET Core SDK in Visual Studio 2017

8/20/2019 • 3 minutes to read • Edit Online

This topic provides a step-by-step introduction to building, debugging, and publishing a simple .NET Core console application using Visual Basic in Visual Studio 2017. Visual Studio 2017 provides a full-featured development environment for building .NET Core applications. As long as the application doesn't have platform-specific dependencies, the application can run on any platform that .NET Core targets and on any system that has .NET Core installed.

Prerequisites

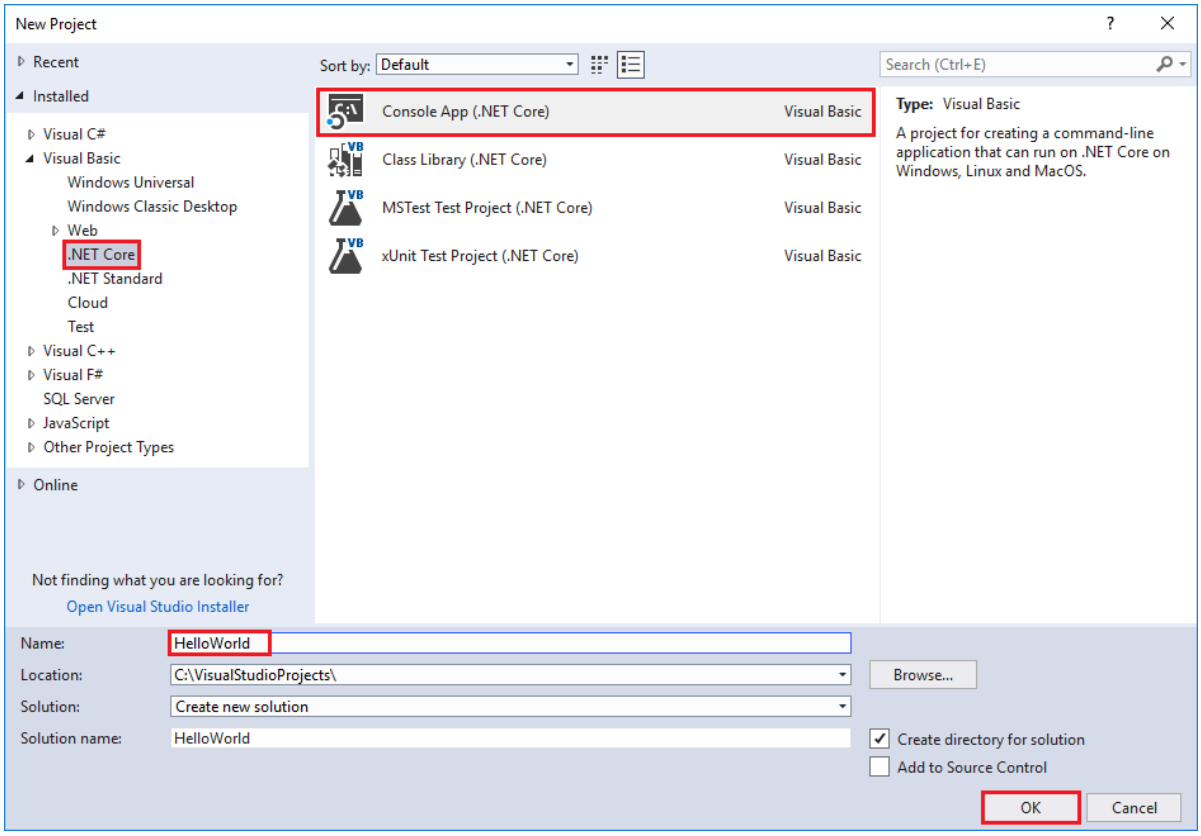
Visual Studio 2017 with the ".NET Core cross-platform development" workload installed. You can develop your app with .NET Core 2.0.

For more information, see Prerequisites for .NET Core on Windows.

A simple Hello World application

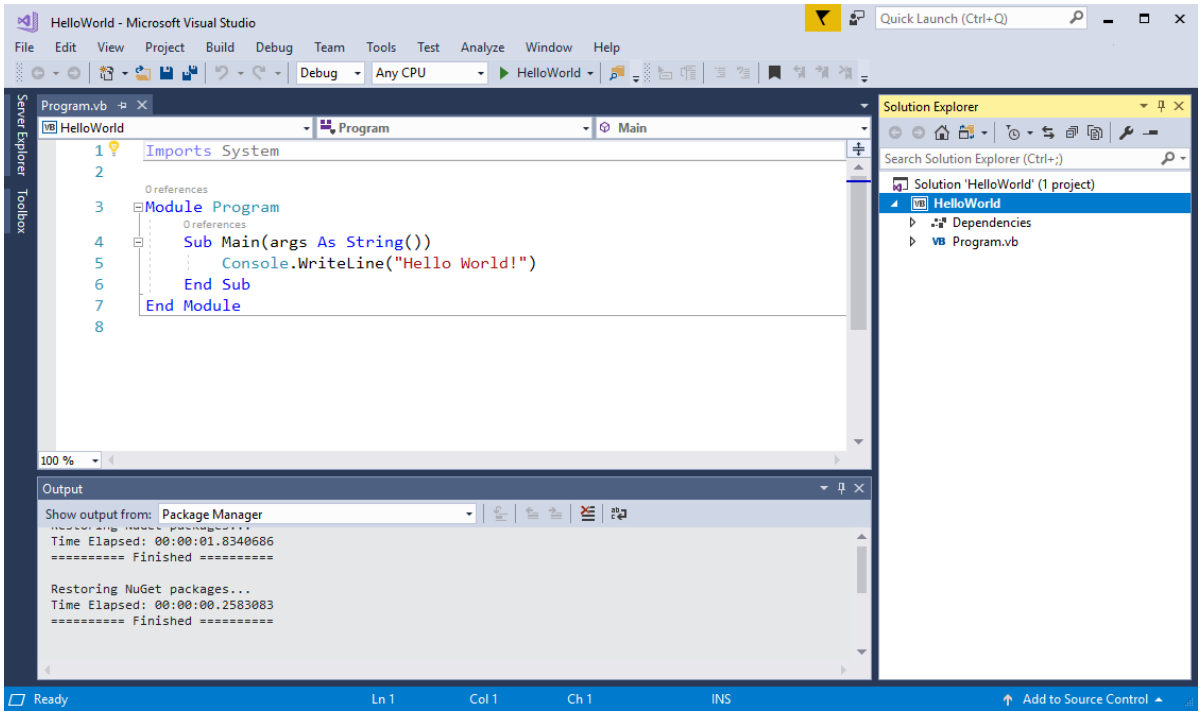
Begin by creating a simple "Hello World" console application. Follow these steps:

1. Launch Visual Studio 2017. Select **File** > **New** > **Project** from the menu bar. In the *New Project** dialog, select the **Visual Basic** node followed by the **.NET Core** node. Then select the **Console App (.NET Core)** project template. In the **Name** text box, type "HelloWorld". Select the **OK** button.



2. Visual Studio uses the template to create your project. The Visual Basic Console Application template for .NET Core automatically defines a class, `Program`, with a single method, `Main`, that takes a String array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime

when it launches the application. Any command-line arguments supplied when the application is launched are available in the *args* array.



The template creates a simple "Hello World" application. It calls the Console.WriteLine(String) method to display the literal string "Hello World!" in the console window. By selecting the **HelloWorld** button with the green arrow on the toolbar, you can run the program in Debug mode. If you do, the console window is visible for only a brief time interval before it closes. This occurs because the `Main` method terminates and the application ends as soon as the single statement in the `Main` method executes.

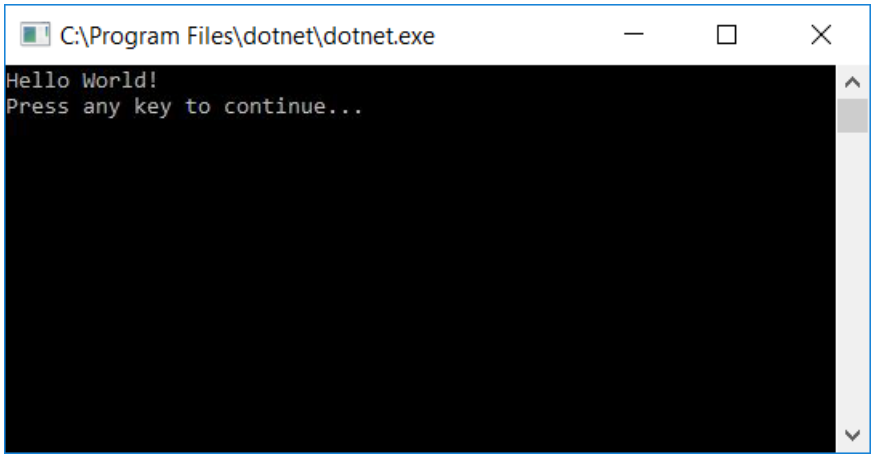
3. To cause the application to pause before it closes the console window, add the following code immediately after the call to the Console.WriteLine(String) method:

```
Console.Write("Press any key to continue...")
Console.ReadKey(true)
```

This code prompts the user to press any key and then pauses the program until a key is pressed.

4. On the menu bar, select **Build** > **Build Solution**. This compiles your program into an intermediate language (IL) that's converted into binary code by a just-in-time (JIT) compiler.

5. Run the program by selecting the **HelloWorld** button with the green arrow on the toolbar.



6. Press any key to close the console window.

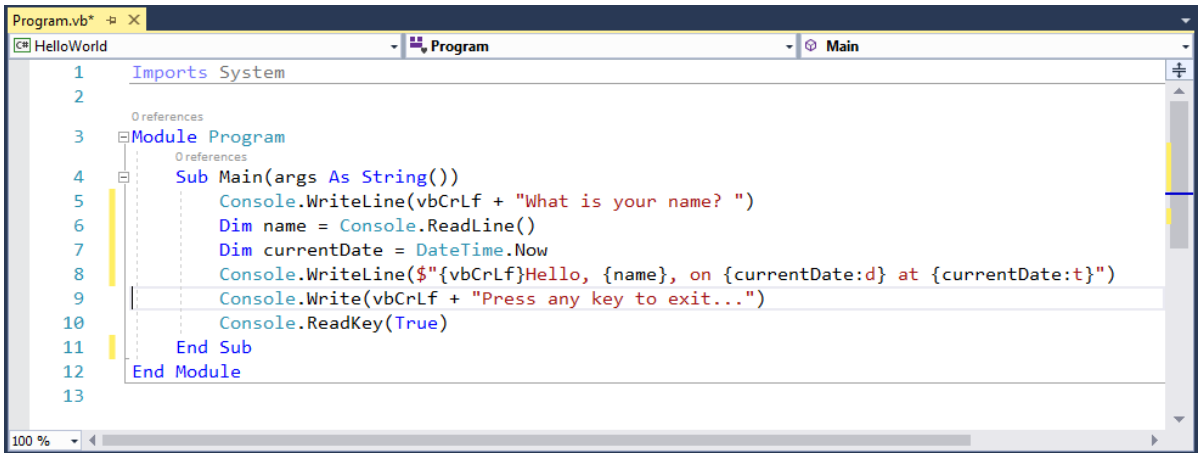
Enhancing the Hello World application

Enhance your application to prompt the user for his or her name and to display it along with the date and time. To modify and test the program, do the following:

1. Enter the following Visual Basic code in the code window immediately after the opening bracket that follows the `Sub Main(args As String())` line and before the first closing bracket:

   ```
   Console.WriteLine(vbCrLf + "What is your name? ")
   Dim name = Console.ReadLine()
   Dim currentDate = DateTime.Now
   Console.WriteLine($"{vbCrLf}Hello, {name}, on {currentDate:d} at {currentDate:t}")
   Console.Write(vbCrLf + "Press any key to exit... ")
   Console.ReadKey(True)
   ```

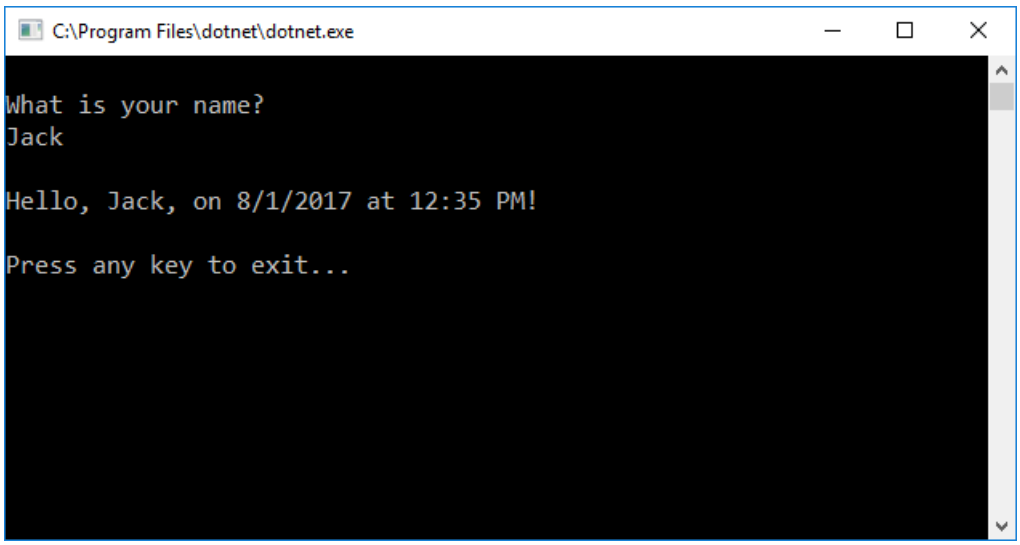
   This code replaces the contents of the `Main` method.

   This code displays "What is your name?" in the console window and waits until the user enters a string followed by the Enter key. It stores this string into a variable named `name`. It also retrieves the value of the DateTime.Now property, which contains the current local time, and assigns it to a variable named `currentDate`. Finally, it uses an interpolated string to display these values in the console window.

2. Compile the program by choosing **Build** > **Build Solution**.

3. Run the program in Debug mode in Visual Studio by selecting the green arrow on the toolbar, pressing F5, or choosing the **Debug** > **Start Debugging** menu item. Respond to the prompt by entering a name and pressing the Enter key.

4. Press any key to close the console window.

You've created and run your application. To develop a professional application, take some additional steps to make your application ready for release:

- To debug your application, see Debug your .NET Core Hello World application using Visual Studio 2017.
- To publish a distributable version of your application, see Publish your .NET Core Hello World application with Visual Studio 2017.

Related topics

Instead of a console application, you can also build a .NET Standard class library with Visual Basic, .NET Core, and Visual Studio 2017. For a step-by-step introduction, see Build a .NET Standard library with Visual Basic and .NET Core SDK in Visual Studio 2017.

Create a .NET Core application with plugins

7/19/2019 • 8 minutes to read • Edit Online

This tutorial shows you how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the `System.Runtime.Loader.AssemblyDependencyResolver` type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET Core 3.0 Preview 2 SDK or a newer version.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run `dotnet new console -o AppWithPlugin`.
2. To make building the project easier, create a Visual Studio solution file. Run `dotnet new sln` in the same folder.
3. Run `dotnet sln add AppWithPlugin/AppWithPlugin.csproj` to add the app project to the solution.

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it is still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions do not conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Create a simple plugin with no dependencies

Back in the root folder, do the following:

1. Run `dotnet new classlib -o HelloPlugin` to create a new class library project named `HelloPlugin`.
2. Run `dotnet sln add HelloPlugin/HelloPlugin.csproj` to add the project to the `AppWithPlugin` solution.
3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

In between the `<Project>` tags, add the following elements:

```
<ItemGroup>
<ProjectReference Include="..\PluginBase\PluginBase.csproj">
    <Private>false</Private>
</ProjectReference>
</ItemGroup>
```

The `<Private>false</Private>` element is very important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method will not find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Now that the `HelloPlugin` project is complete, we should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"` as an element of the `pluginPaths` array.

Create a plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJson` plugin projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. The project files themselves do not have any special information for the project references, and (after adding the plugin paths to the `pluginPaths` array) the plugins run perfectly, even if run in the same execution of the AppWithPlugin app. However, these projects do not copy the referenced assemblies to their output directory, so the assemblies need to be present on the user's machine for the plugins to work. There are two ways to work around this problem. The first option is to use the `dotnet publish` command to publish the class library. Alternatively, if you want to be able to use the output of `dotnet build` for your plugin, you can add the `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` property between the `<PropertyGroup>` tags in the plugin's project file. See the `XcopyablePlugin` plugin project for an example.

Other plugin examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository. The completed sample includes a few other examples of `AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

How to reference a plugin interface assembly defined in a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET Core 3.0, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Debug your C# or Visual Basic .NET Core Hello World application using Visual Studio 2017

5/6/2019 • 11 minutes to read • Edit Online

So far, you've followed the steps in Build a C# Hello World Application with .NET Core in Visual Studio 2017 or Build a Visual Basic Hello World Application with .NET Core in Visual Studio 2017 to create and run a simple console application. Once you've written and compiled your application, you can begin testing it. Visual Studio includes a comprehensive set of debugging tools that you can use when testing and troubleshooting your application.

Debugging in Debug mode

Debug and *Release* are two of Visual Studio's default build configurations. The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile your application in **Debug** mode.

You should always begin by testing your program in Debug mode. Debug mode turns off most compiler optimizations and provides richer information during the build process.

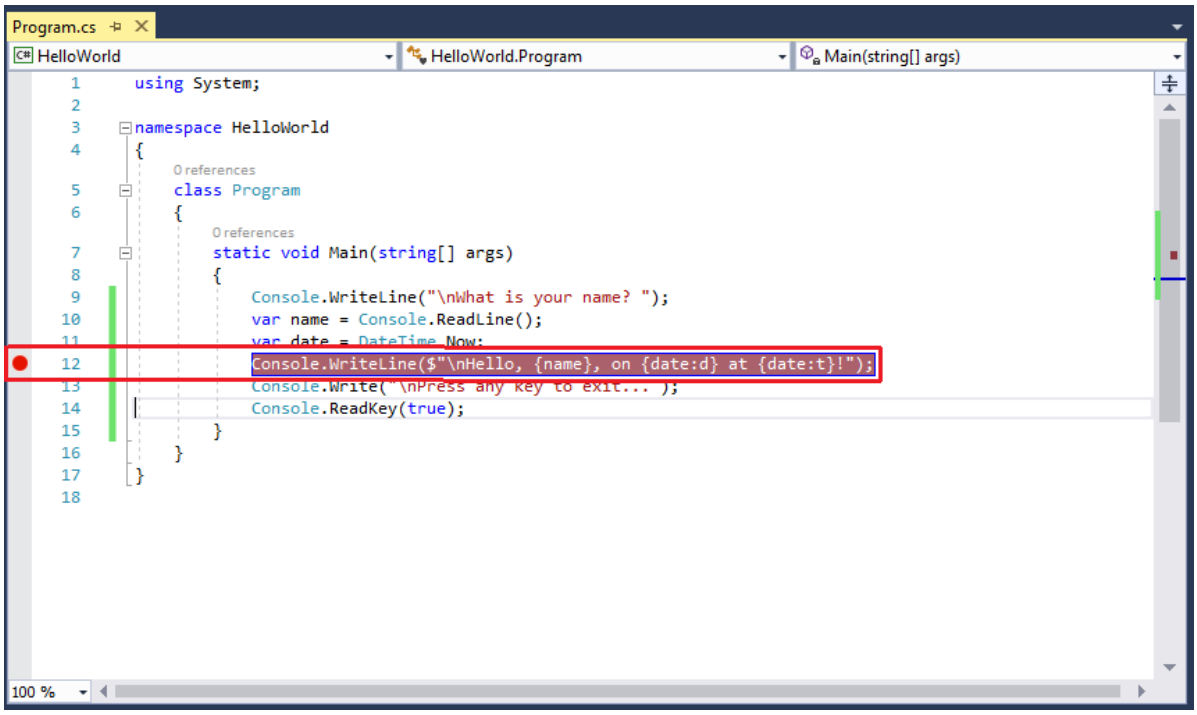
Setting a breakpoint

Run your program in Debug mode and try a few debugging features:

- C#
- Visual Basic

1. A *breakpoint* temporarily interrupts the execution of the application *before* the line with the breakpoint is executed.

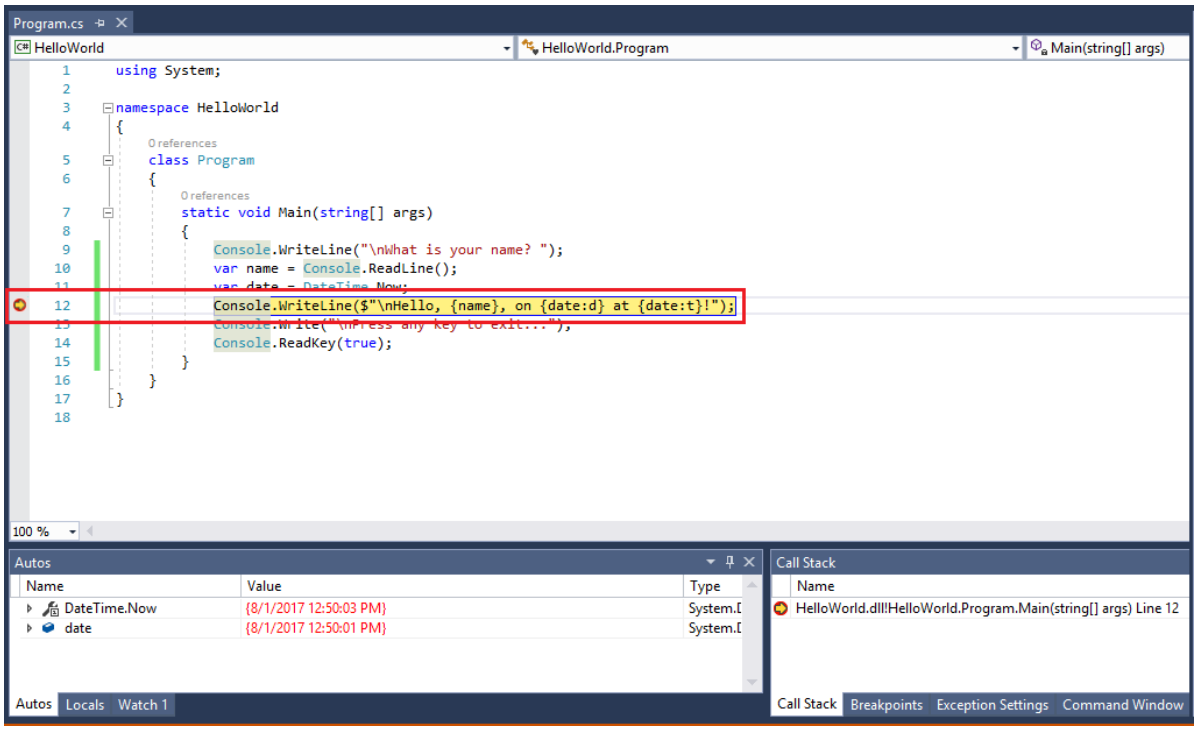
   Set a breakpoint on the line that reads `Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");` by clicking in the left margin of the code window on that line or by choosing the **Debug** > **Toggle Breakpoint** menu item with the line selected. As the following figure shows, Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red circle in its left margin.



2. Run the program in Debug mode by selecting the **HelloWorld** button with the green arrow on the toolbar, pressing F5, or choosing **Debug** > **Start Debugging**.

3. Enter a string in the console window when the program prompts for a name and press Enter.

4. Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes. The **Autos** window displays the values of variables that are used around the current line. The **Locals** window (which you can view by clicking the **Locals** tab) displays the values of variables that are defined in the currently executing method.

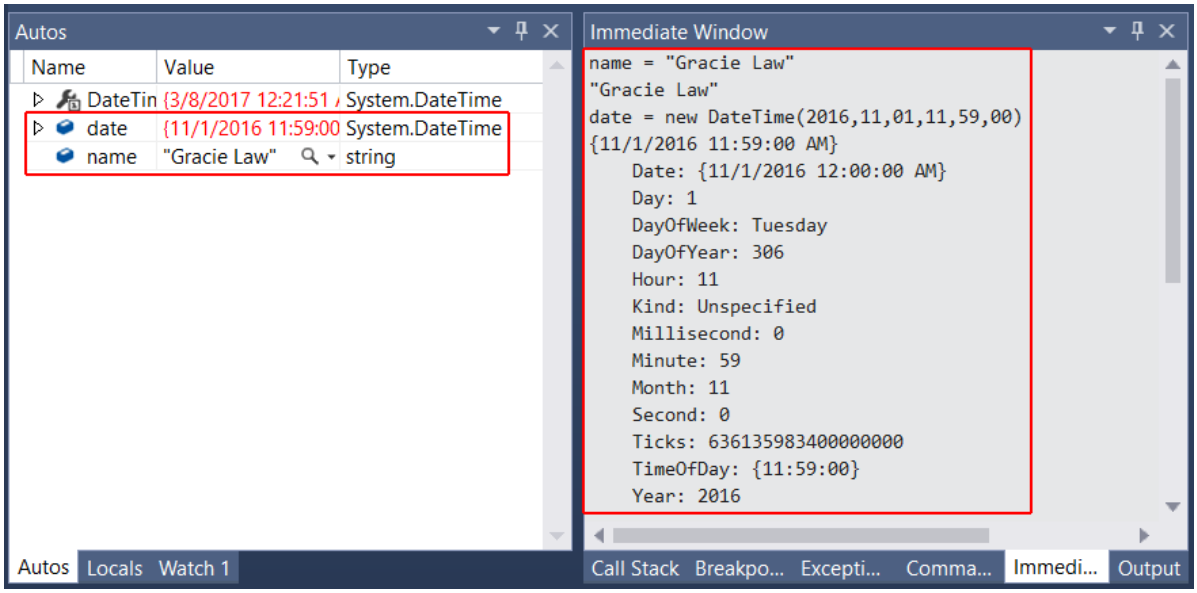


5. You can change the value of the variables to see how it affects your program. If the **Immediate Window** is not visible, display it by choosing the **Debug** > **Windows** > **Immediate** menu item. The **Immediate Window** lets you interact with the application you're debugging.

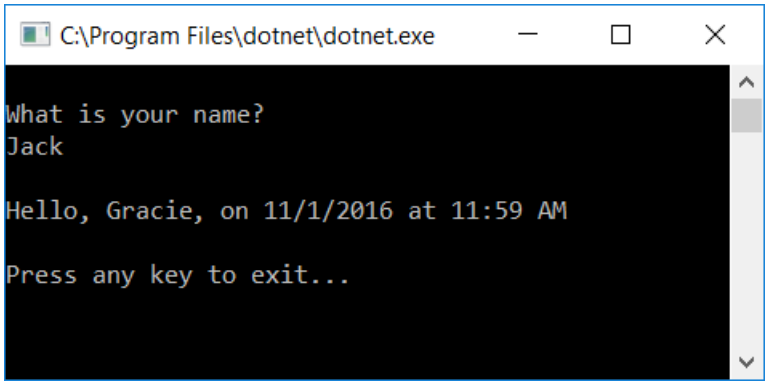
6. You can interactively change the values of variables. Enter `name = "Gracie"` in the **Immediate Window** and press the Enter key.

7. Enter `date = new DateTime(2016,11,01,11,59,00)` in the **Immediate Window** and press the Enter key.

   The **Immediate Window** displays the value of the string variable and the properties of the DateTime value. In addition, the value of the variables is updated in the **Autos** and **Locals** windows.



8. Continue program execution by selecting the **Continue** button in the toolbar or by selecting the **Debug** > **Continue** menu item. The values displayed in the console window correspond to the changes you made in the **Immediate Window**.



9. Press any key to exit the application and end Debug mode.

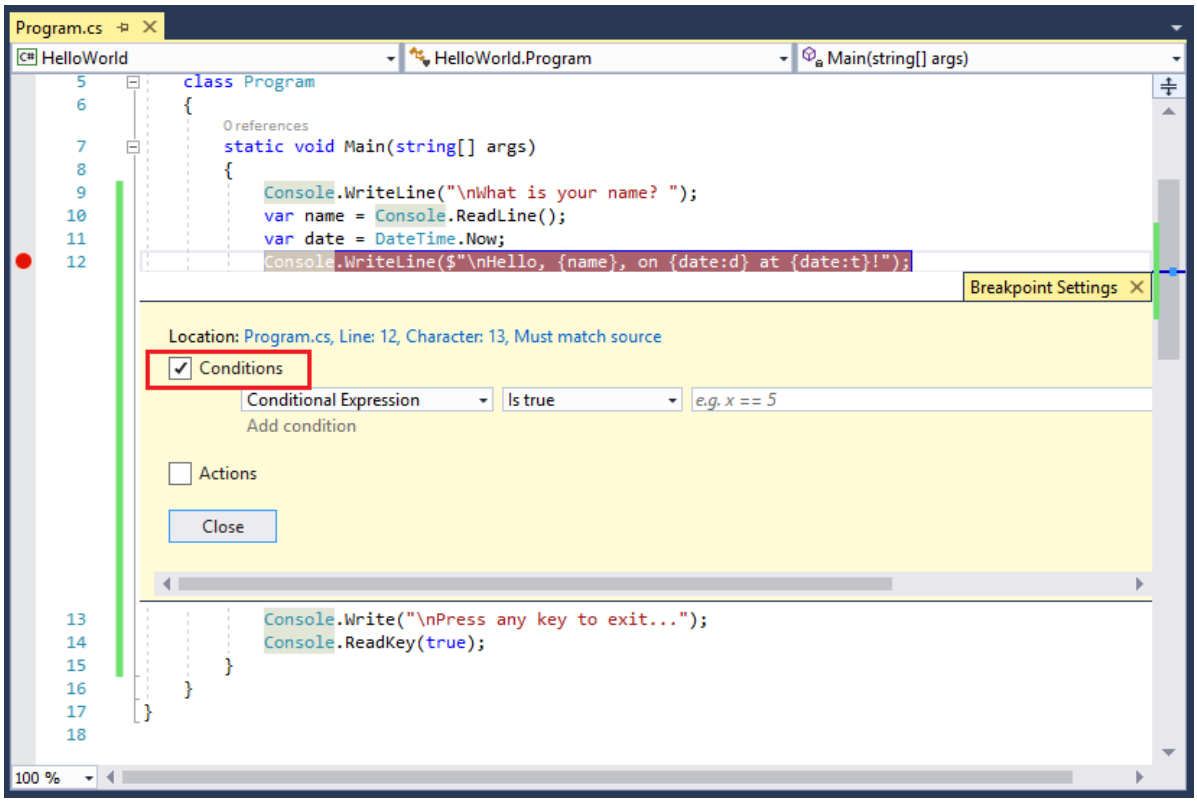
Setting a conditional breakpoint

Your program displays the string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature, the *conditional breakpoint*, which breaks program execution when one or more conditions are met.

To set a conditional breakpoint and test what happens when the user fails to enter a string, do the following:

- C#
- Visual Basic

1. Right-click on the red dot that represents the breakpoint. On the context menu, select **Conditions** to open the **Breakpoint Settings** dialog. Check the box for **Conditions**.



2. For the **Conditional Expression** replace "e.g. x == 5" with the following:

```
String.IsNullOrEmpty(name)
```

   You're testing for a code condition, that the `String.IsNullOrEmpty(name)` method call is `true` either because *name* has not been assigned a value or because its value is an empty string (""). You can also specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times, or a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

3. Select the **Close** button to close the dialog.

4. Run the program in Debug mode.

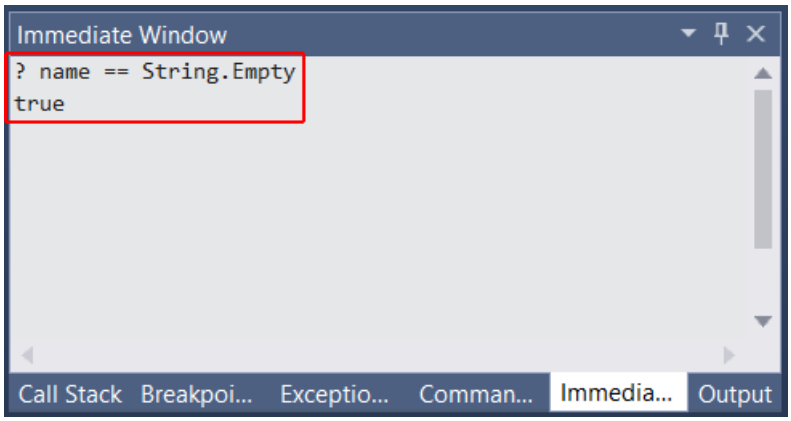
5. In the console window, press the Enter key when prompted to enter your name.

6. Because the condition we specified, `name` is either `null` or String.Empty, has been satisfied, program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes.

7. Select the **Locals** window, which shows the values of variables that are local to the currently executing method, which is the `Main` method in your program. Observe that the value of the `name` variable is `""`, or String.Empty.

8. Confirm the value is an empty string by entering the following statement in the **Immediate Window**. The result is `true`.

```
? name == String.Empty
```



9. Select the **Continue** button on the toolbar to continue program execution.
10. Press any key to close the console window and exit Debug mode.
11. Clear the breakpoint by clicking on the dot in the left margin of the code window or by choosing the **Debug > Toggle Breakpoint** menu item with the row selected.

Stepping through a program

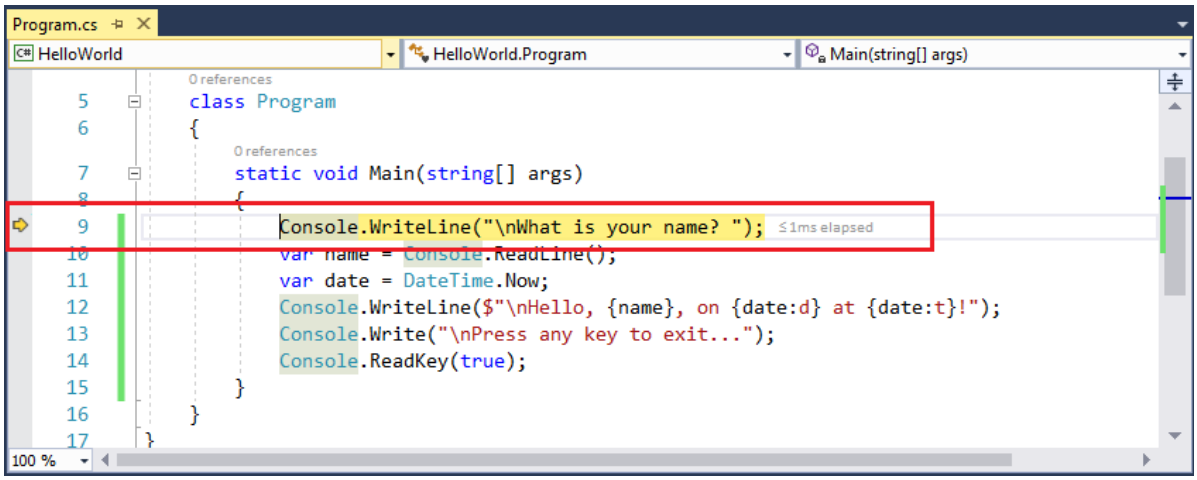
Visual Studio also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and use this feature to follow program flow through a small part of your program code. Since your program is small, you can step through the entire program by doing the following:

- C#
- Visual Basic

1. On the menu bar, choose **Debug** > **Step Into** or press the F11 key. Visual Studio highlights and displays an arrow beside the next line of execution.

At this point, the **Autos** window shows that your program has defined only one variable, `args`. Because you haven't passed any command-line arguments to the program, its value is an empty string array. In addition, Visual Studio has opened a blank console window.

2. Select **Debug** > **Step Into** or press the F11 key. Visual Studio now highlights the next line of execution. As the figure shows, it has taken less than one millisecond to execute the code between the last statement and this one. `args` remains the only declared variable, and the console window remains blank.



3. Select **Debug** > **Step Into** or press the F11 key. Visual Studio highlights the statement that includes the `name` variable assignment. The **Autos** window shows that `name` is `null`, and the console window displays the string "What is your name?".

4. Respond to the prompt by entering a string in the console window and pressing Enter. The console is unresponsive, and the string you enter isn't displayed in the console window, but the Console.ReadLine method will nevertheless capture your input.

5. Select **Debug** > **Step Into** or press the F11 key. Visual Studio highlights the statement that includes the `date` (in C#) or `currentDate` (in Visual Basic) variable assignment. The **Autos** window shows the DateTime.Now property value and the value returned by the call to the Console.ReadLine method. The console window also displays the string entered when the console prompted for input.

6. Select **Debug** > **Step Into** or press the F11 key. The **Autos** window shows the value of the `date` variable after the assignment from the DateTime.Now property. The console window is unchanged.

7. Select **Debug** > **Step Into** or press the F11 key. Visual Studio calls the Console.WriteLine(String, Object, Object) method. The values of the `date` (or `currentDate`) and `name` variables appear in the **Autos** window, and the console window displays the formatted string.

8. Select **Debug** > **Step Out** or press Shift and the F11 key. This stops step-by-step execution. The console window displays a message and waits for you to press a key.

9. Press any key to close the console window and exit Debug mode.

Building a Release version

Once you've tested the Debug build of your application, you should also compile and test the Release version. The Release version incorporates compiler optimizations that can sometimes negatively affect the behavior of an application. For example, compiler optimizations that are designed to improve performance can create race conditions in asynchronous or multithreaded applications.

To build and test the Release version of your console application, change the build configuration on the toolbar from **Debug** to **Release**.

When you press F5 or choose **Build Solution** from the **Build** menu, Visual Studio compiles the Release version of your console application. You can test it as you did the Debug version of the application.

Once you've finished debugging your application, the next step is to publish a deployable version of your application. For information on how to do this, see Publish the Hello World application with Visual Studio 2017.

Publish your .NET Core Hello World application with Visual Studio 2017

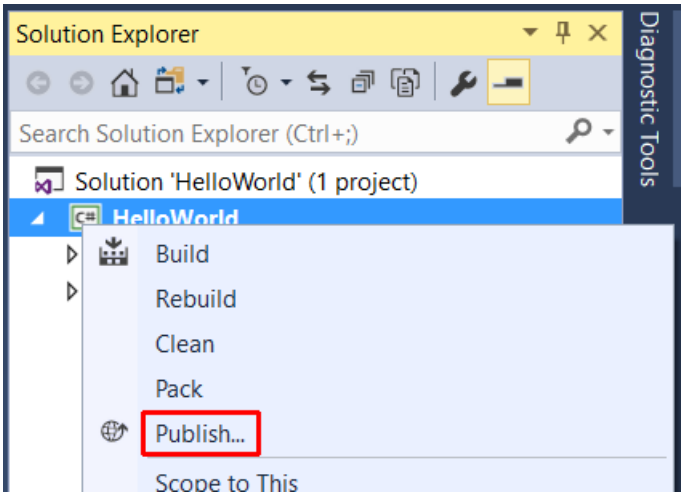
8/20/2019 • 2 minutes to read • Edit Online

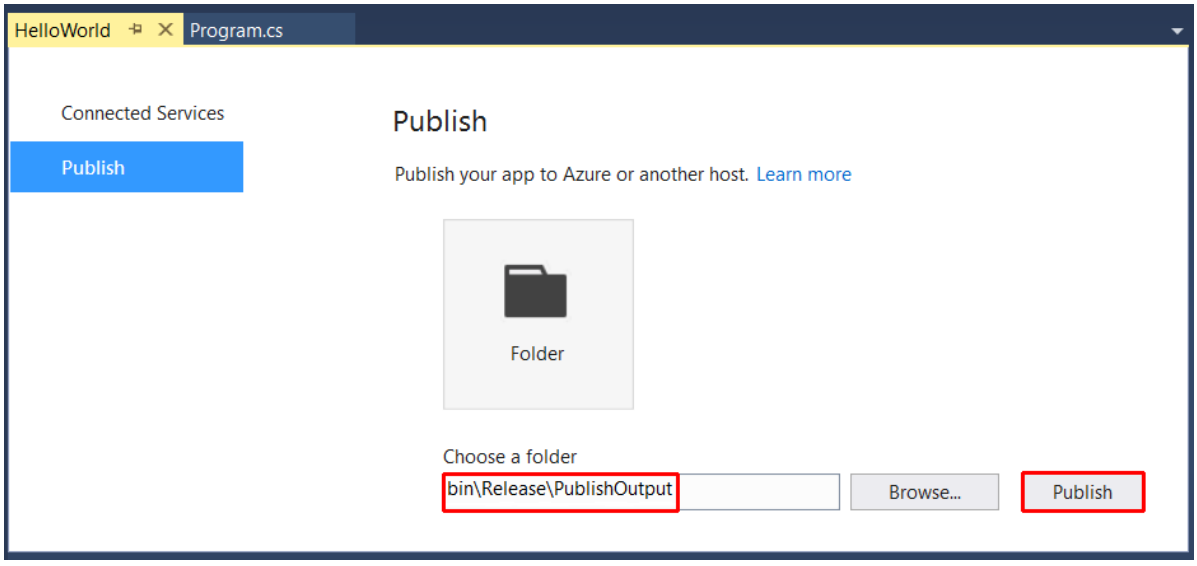
In Build a C# Hello World application with .NET Core in Visual Studio 2017 or Build a Visual Basic Hello World application with .NET Core in Visual Studio 2017, you built a Hello World console application. In Debug your C# Hello World application with Visual Studio 2017, you tested it using the Visual Studio debugger. Now that you're sure that it works as expected, you can publish it so that other users can run it. Publishing creates the set of files that are needed to run your application, and you can deploy the files by copying them to a target machine.

To publish and run your application:

1. Make sure that Visual Studio is building the Release version of your application. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.

2. Right-click on the **HelloWorld** project (not the HelloWorld solution) and select **Publish** from the menu. You can also select **Publish HelloWorld** from the main Visual Studio **Build** menu.





3. Open a console window. For example in the **Type here to search** text box in the Windows taskbar, enter

`Command Prompt` (or `cmd` for short), and open a console window by either selecting the **Command Prompt** desktop app or pressing Enter if it's selected in the search results.

4. Navigate to the published application in the `bin\release\PublishOutput` subdirectory of your application's project directory. As the following figure shows, the published output includes the following four files:

   - *HelloWorld.deps.json*

     The application's runtime dependencies file. It defines the .NET Core components and the libraries (including the dynamic link library that contains your application) needed to run your application. For more information, see Runtime Configuration Files.

   - *HelloWorld.dll*

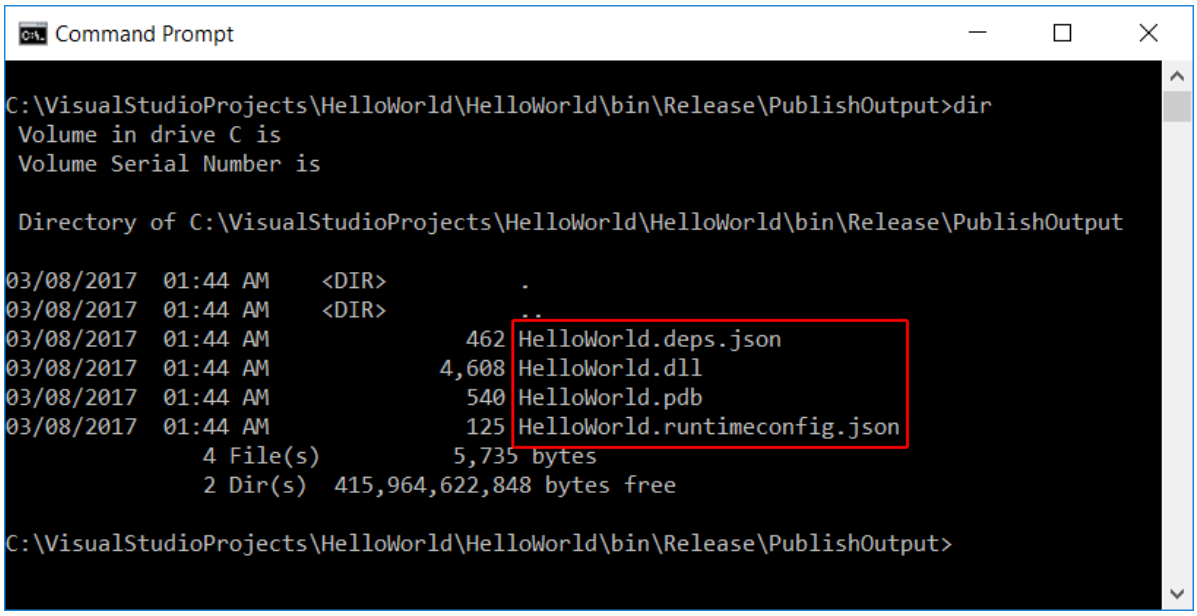
     The file that contains your application. It is a dynamic link library that can be executed by entering the `dotnet HelloWorld.dll` command in a console window.

   - *HelloWorld.pdb* (optional for deployment)

     A file that contains debug symbols. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

   - *HelloWorld.runtimeconfig.json*

     The application's runtime configuration file. It identifies the version of .NET Core that your application was built to run on. For more information, see Runtime Configuration Files.



The publishing process creates a framework-dependent deployment, which is a type of deployment where the published application will run on any platform supported by .NET Core with .NET Core installed on the system. Users can run your application by issuing the `dotnet HelloWorld.dll` command from a console window.

For more information on publishing and deploying .NET Core applications, see .NET Core Application Deployment.

Build a .NET Standard library with C# and the .NET Core SDK in Visual Studio 2017

5/15/2019 • 2 minutes to read • Edit Online

A *class library* defines types and methods that are called by an application. A class library that targets the .NET Standard 2.0 allows your library to be called by any .NET implementation that supports that version of the .NET Standard. When you finish your class library, you can decide whether you want to distribute it as a third-party component or whether you want to include it as a bundled component with one or more applications.

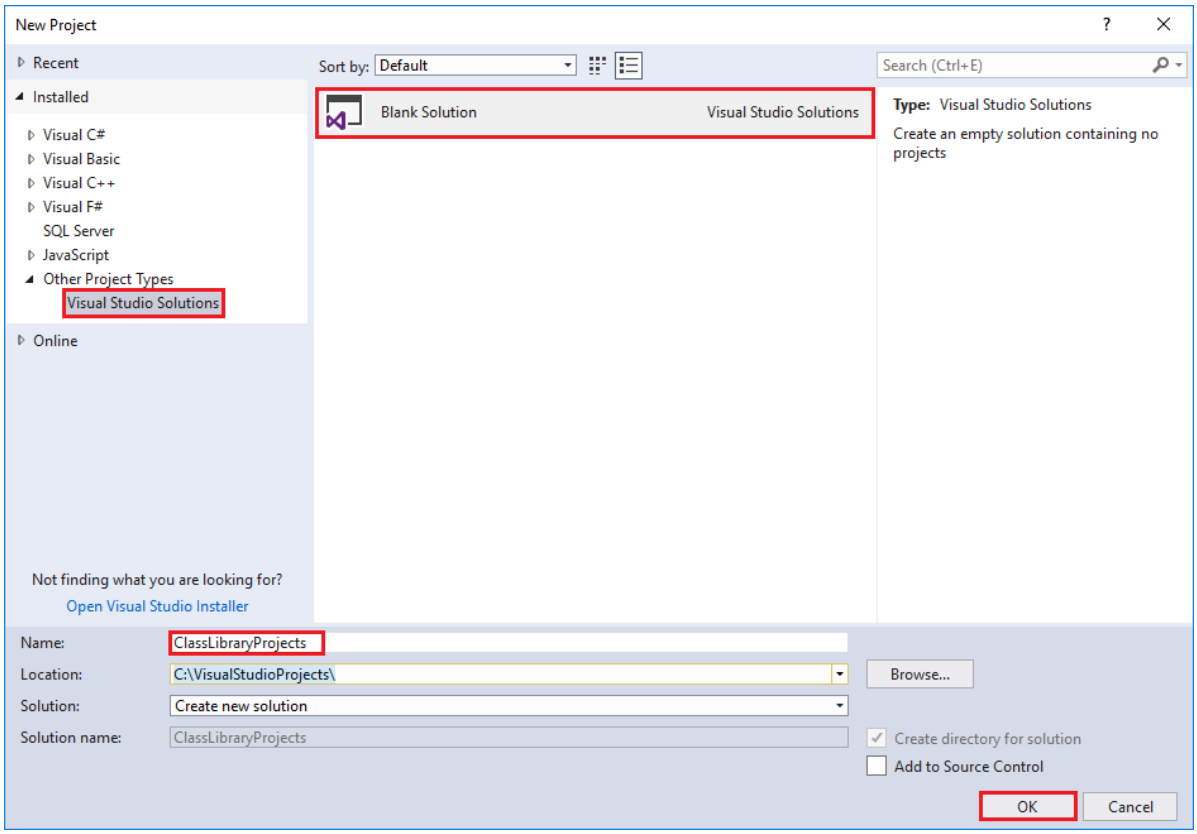
> **NOTE**
>
> For a list of the .NET Standard versions and the platforms they support, see .NET Standard.

In this topic, you'll create a simple utility library that contains a single string-handling method. You'll implement it as an extension method so that you can call it as if it were a member of the String class.

Creating a class library solution

Start by creating a solution for your class library project and its related projects. A Visual Studio Solution just serves as a container for one or more projects. To create the solution:

1. On the Visual Studio menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog, expand the **Other Project Types** node, and select **Visual Studio Solutions**. Name the solution "ClassLibraryProjects" and select the **OK** button.

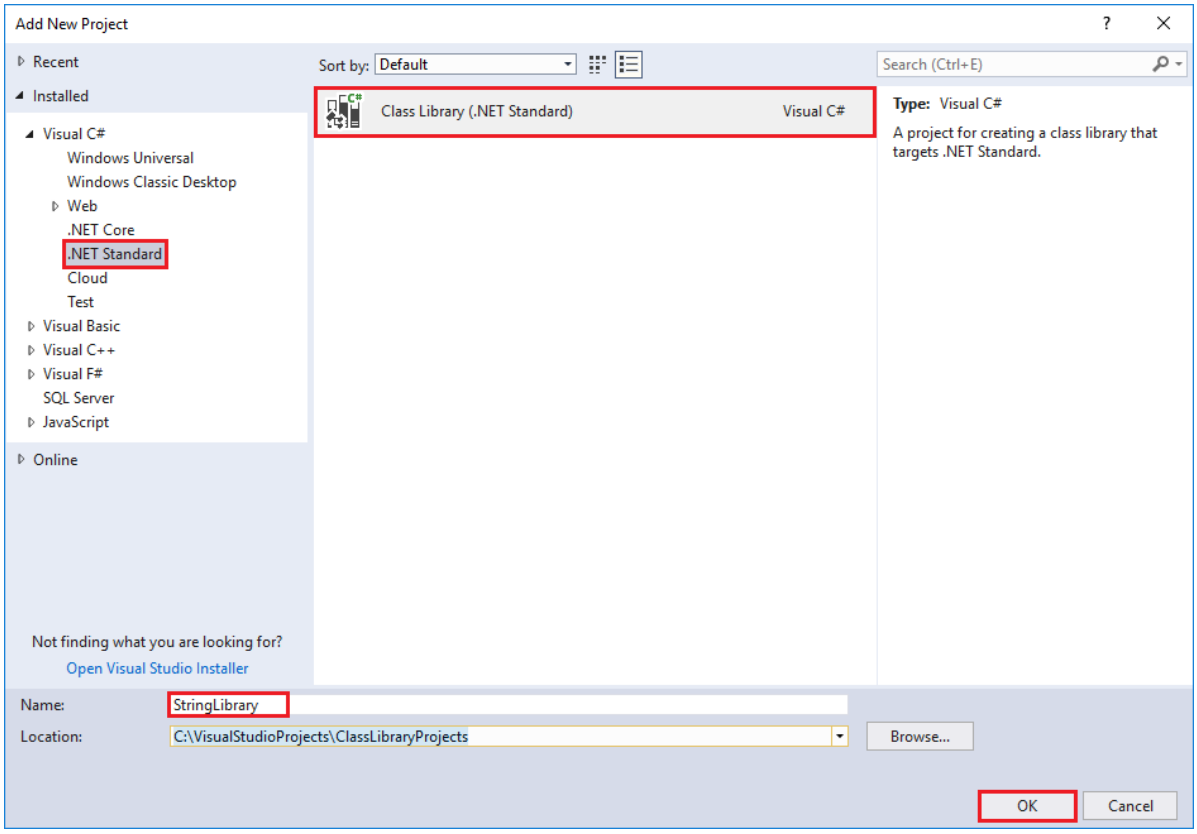


Creating the class library project

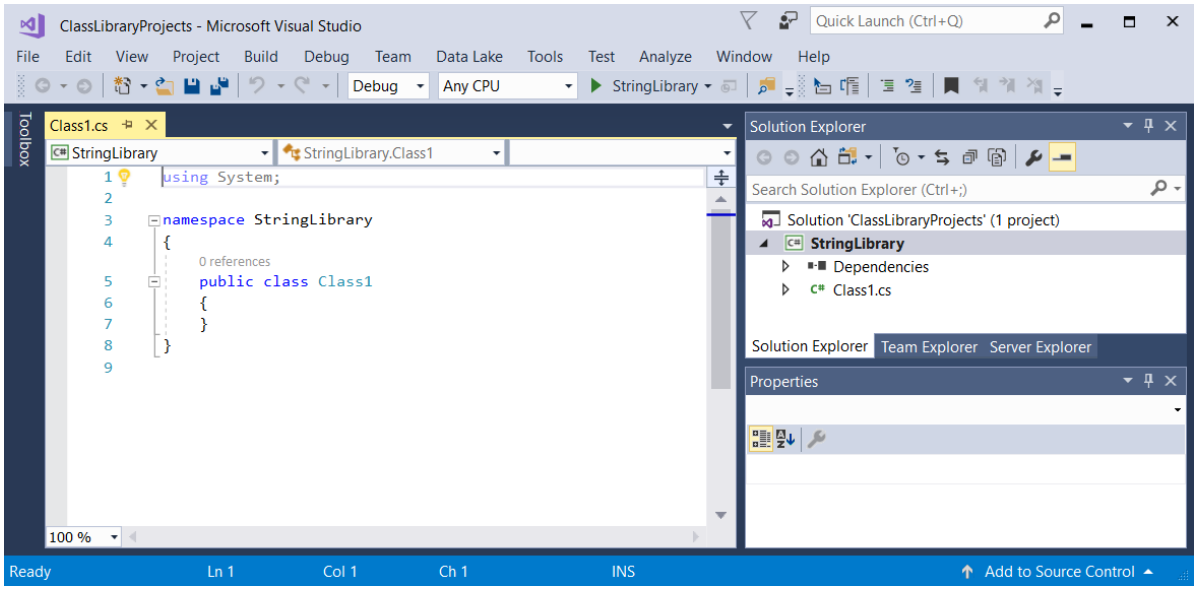
Create your class library project:

1. In **Solution Explorer**, right-click on the **ClassLibraryProjects** solution file and from the context menu, select **Add** > **New Project**.

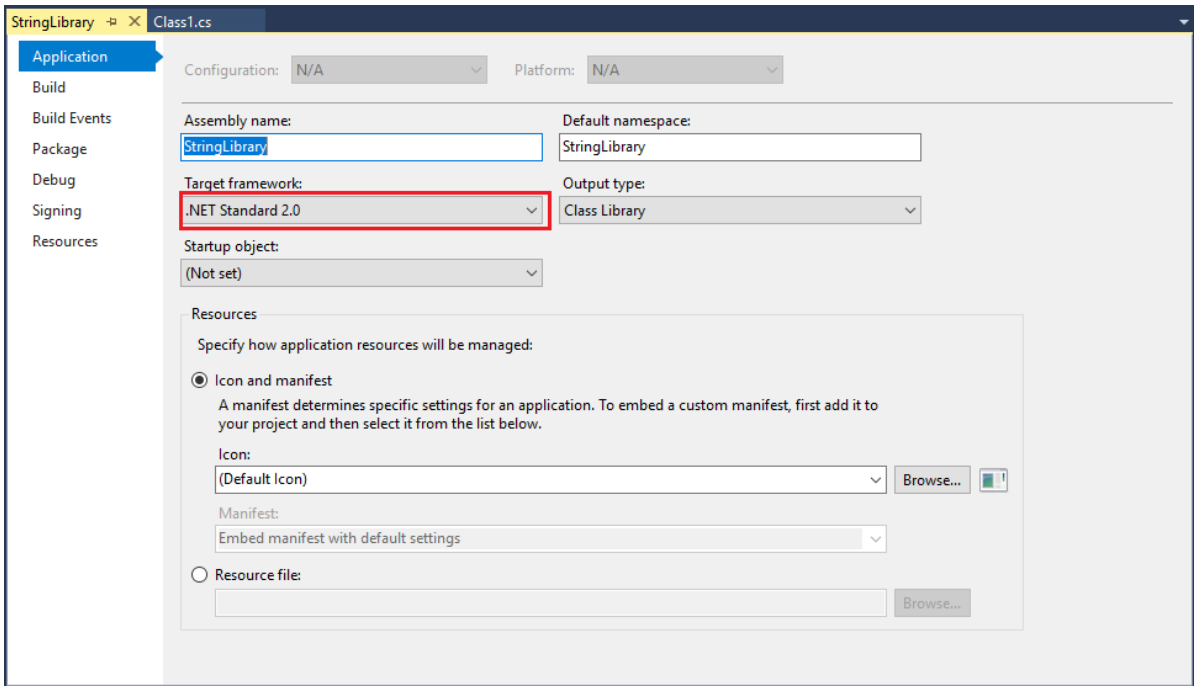
2. In the **Add New Project** dialog, expand the **Visual C#** node, then select the **.NET Standard** node followed by the **Class Library (.NET Standard)** project template. In the **Name** text box, enter "StringLibrary" as the name of the project. Select **OK** to create the class library project.



The code window then opens in the Visual Studio development environment.



3. Check to make sure that our library targets the correct version of the .NET Standard. Right-click on the library project in the **Solution Explorer** windows, then select **Properties**. The **Target Framework** text box shows that we're targeting .NET Standard 2.0.

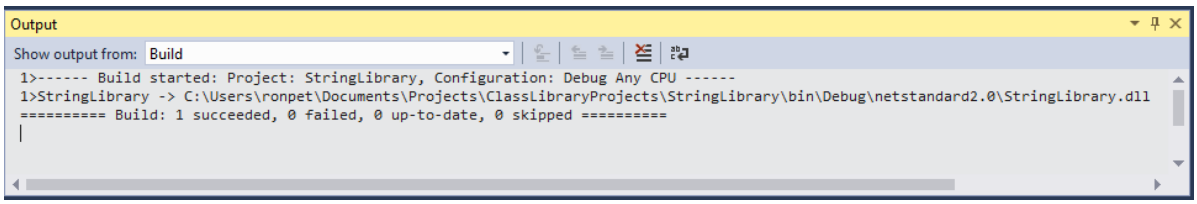


4. Replace the code in the code window with the following code and save the file:

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this String str)
        {
            if (String.IsNullOrWhiteSpace(str))
                return false;

            Char ch = str[0];
            return Char.IsUpper(ch);
        }
    }
}
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`, which returns a Boolean value that indicates whether the current string instance begins with an uppercase character. The Unicode standard distinguishes uppercase characters from lowercase characters. The Char.IsUpper(Char) method returns `true` if a character is uppercase.

5. On the menu bar, select **Build** > **Build Solution**. The project should compile without error.



Next step

You've successfully built the library. Because you haven't called any of its methods, you don't know whether it works as expected. The next step in developing your library is to test it by using a Unit Test Project.

Build a .NET Standard library with Visual Basic and the .NET Core SDK in Visual Studio 2017

4/8/2019 • 2 minutes to read • Edit Online

A *class library* defines types and methods that are called by an application. A class library that targets the .NET Standard 2.0 allows your library to be called by any .NET implementation that supports that version of the .NET Standard. When you finish your class library, you can decide whether you want to distribute it as a third-party component or whether you want to include it as a bundled component with one or more applications.

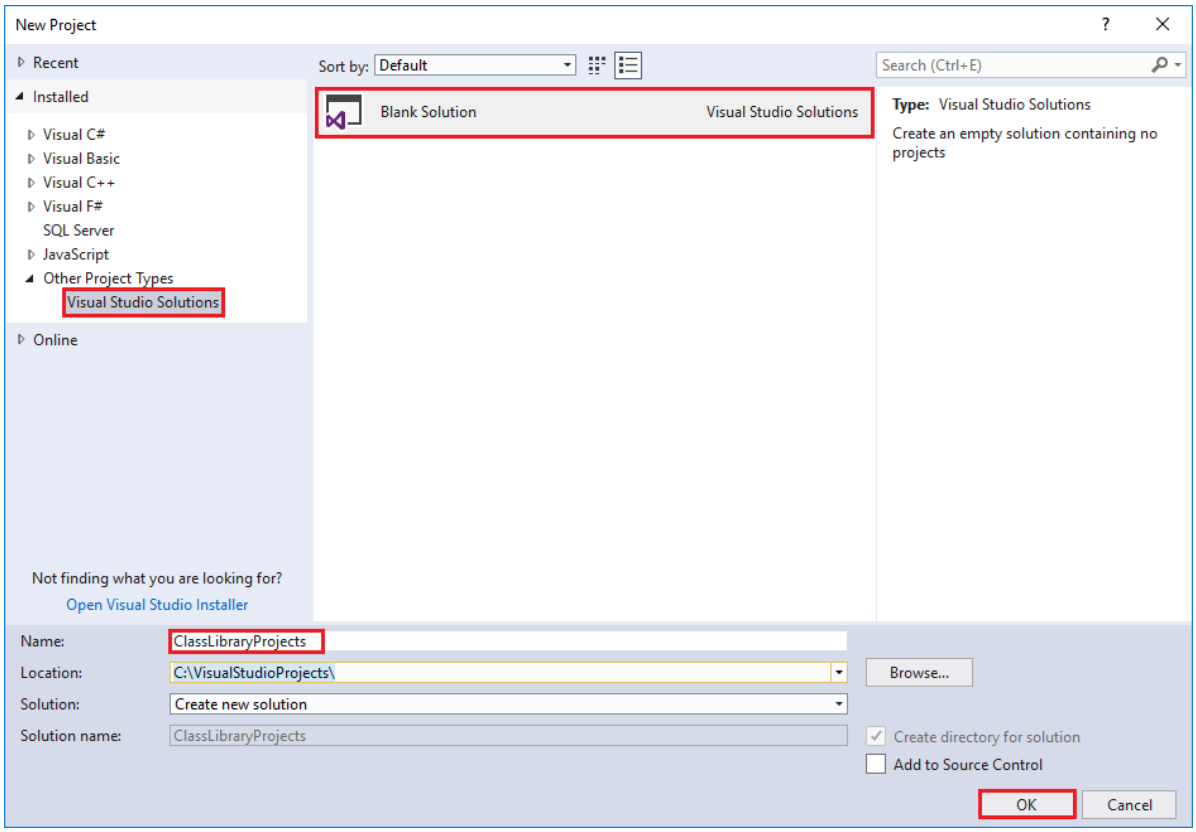
> **NOTE**
> For a list of the .NET Standard versions and the platforms they support, see .NET Standard.

In this topic, you'll create a simple utility library that contains a single string-handling method. You'll implement it as an extension method so that you can call it as if it were a member of the String class.

Creating a class library solution

Start by creating a solution for your class library project and its related projects. A Visual Studio Solution just serves as a container for one or more projects. To create the solution:

1. On the Visual Studio menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog, expand the **Other Project Types** node, and select **Visual Studio Solutions**. Name the solution "ClassLibraryProjects" and select the **OK** button.

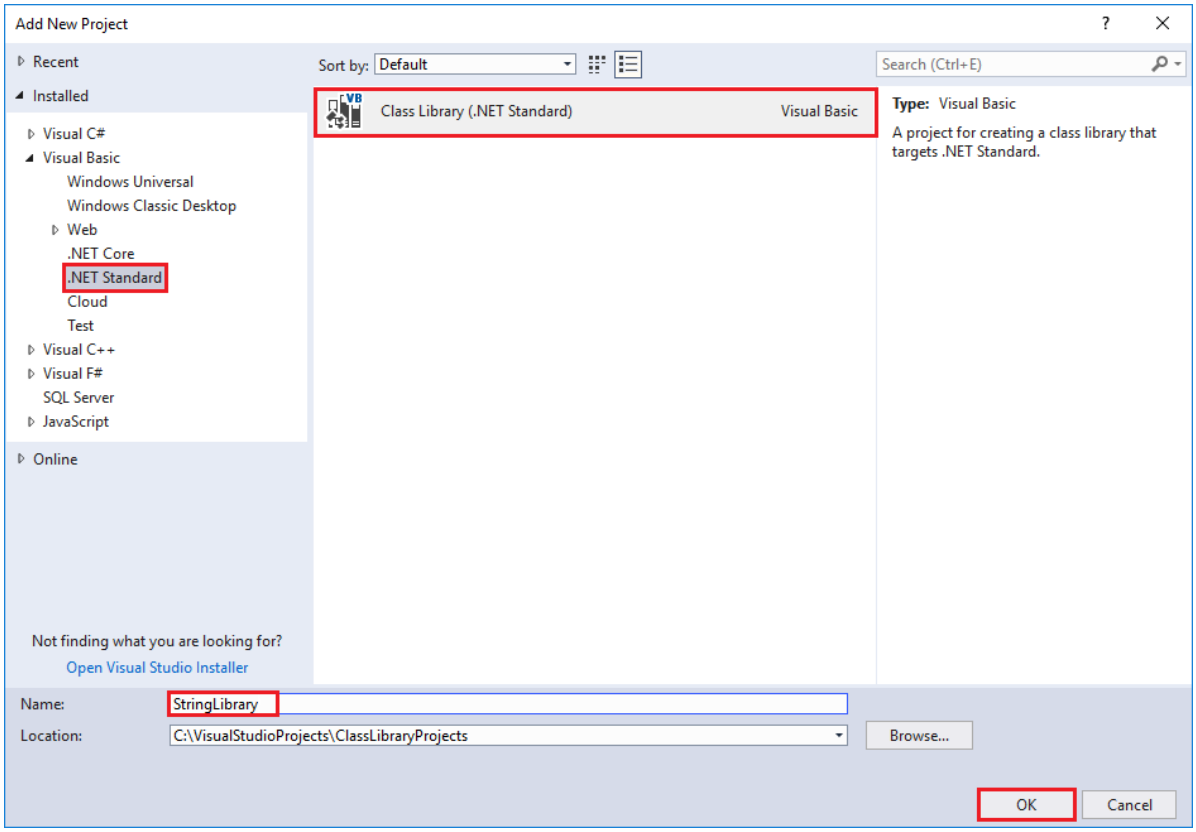


Creating the class library project

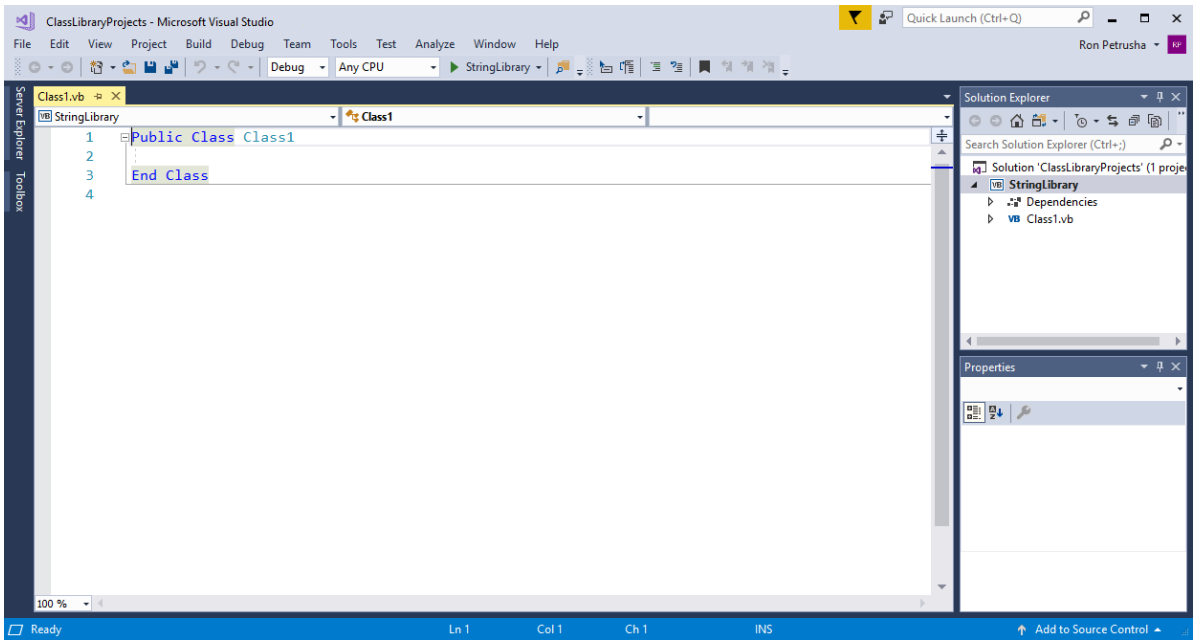
Create your class library project:

1. In **Solution Explorer**, right-click on the **ClassLibraryProjects** solution file and from the context menu, select **Add** > **New Project**.

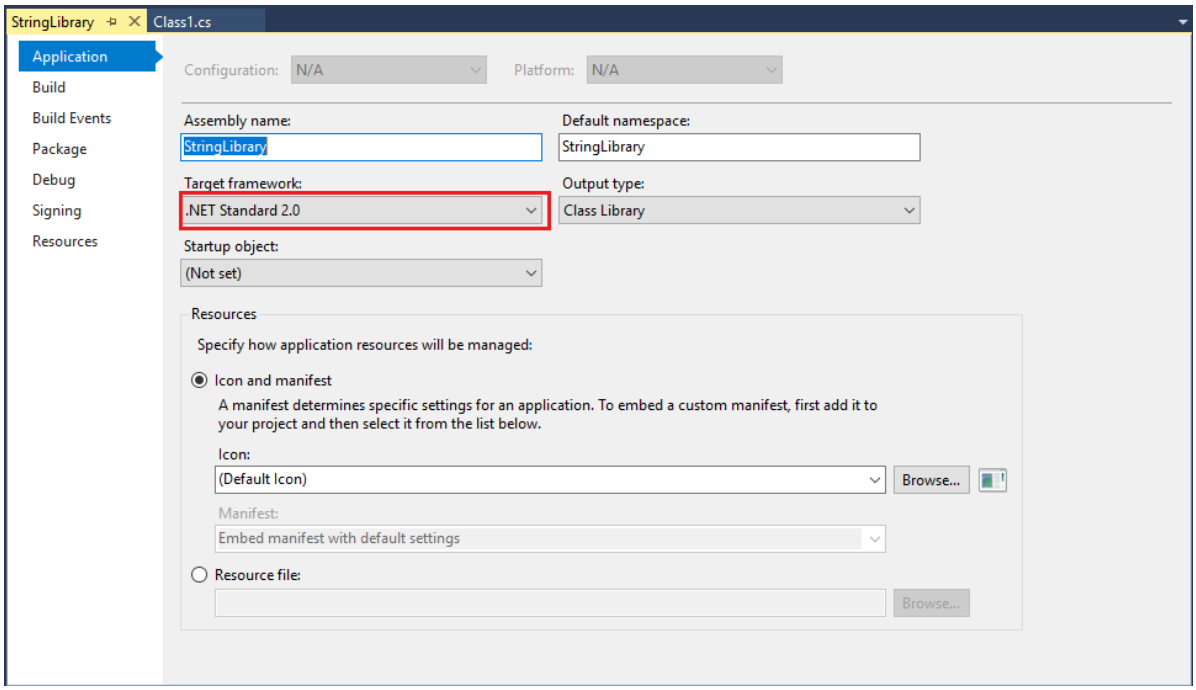
2. In the **Add New Project** dialog, expand the **Visual Basic** node, then select the **.NET Standard** node followed by the **Class Library (.NET Standard)** project template. In the **Name** text box, enter "StringLibrary" as the name of the project. Select **OK** to create the class library project.



The code window then opens in the Visual Studio development environment.



3. Check to make sure that the library targets the correct version of the .NET Standard. Right-click on the library project in the **Solution Explorer** windows, then select **Properties**. The **Target Framework** text box shows that we're targeting .NET Standard 2.0.



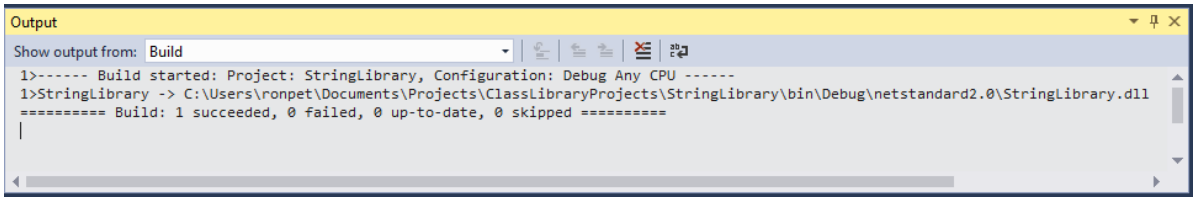
4. Also in the **Properties** dialog, clear the text in the **Root namespace** text box. For each project, Visual Basic automatically creates a namespace that corresponds to the project name, and any namespaces defined in source code files are parents of that namespace. We want to define a top-level namespace by using the `namespace` keyword.

5. Replace the code in the code window with the following code and save the file:

```
Imports System.Runtime.CompilerServices

Namespace UtilityLibraries
    Public Module StringLibrary
        <Extension>
        Public Function StartsWithUpper(str As String) As Boolean
            If String.IsNullOrWhiteSpace(str) Then
                Return False
            End If

            Dim ch As Char = str(0)
            Return Char.IsUpper(ch)
        End Function
    End Module
End Namespace
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`, which returns a Boolean value that indicates whether the current string instance begins with an uppercase character. The Unicode standard distinguishes uppercase characters from lowercase characters. The Char.IsUpper(Char) method returns `true` if a character is uppercase.

1. On the menu bar, select **Build** > **Build Solution**. The project should compile without error.



Next step

You've successfully built the library. Because you haven't called any of its methods, you don't know whether it works as expected. The next step in developing your library is to test it by using a Unit Test Project.

Test a .NET Standard library with .NET Core in Visual Studio 2017

6/26/2019 • 9 minutes to read • Edit Online

In Build a .NET Standard library with C# and .NET Core in Visual Studio 2017 or Build a .NET Standard library with Visual Basic and .NET Core in Visual Studio 2017, you created a simple class library that adds an extension method to the String class. Now, you'll create a unit test to make sure that it works as expected. You'll add your unit test project to the solution you created in the previous article.

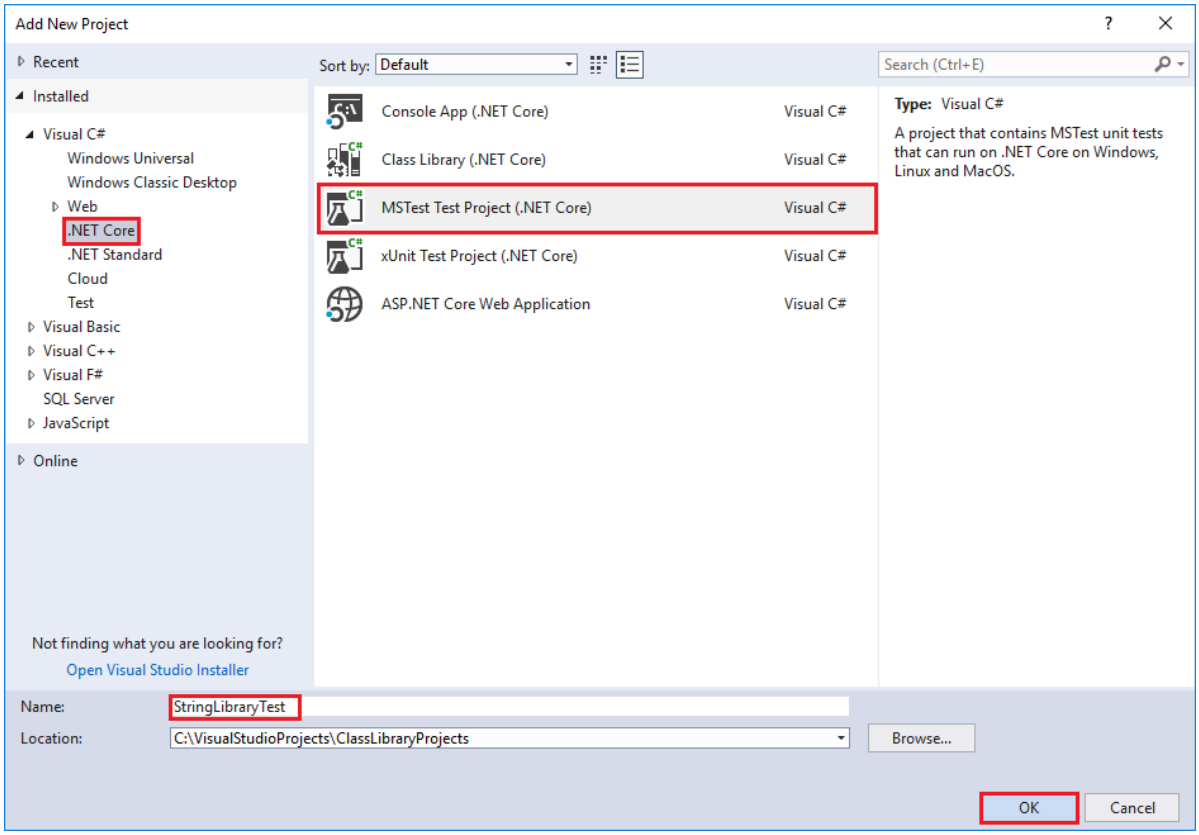
Creating a unit test project

To create the unit test project, do the following:

- C#
- Visual Basic

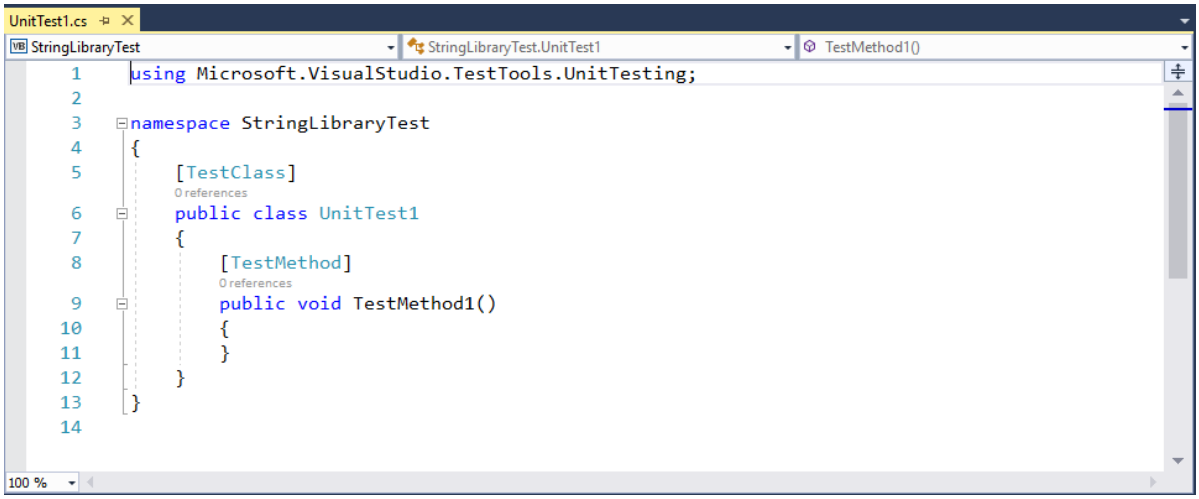
1. In **Solution Explorer**, open the context menu for the **ClassLibraryProjects** solution node and select **Add** > **New Project**.

2. In the **Add New Project** dialog, select the **Visual C#** node. Then select the **.NET Core** node followed by the **MSTest Test Project (.NET Core)** project template. In the **Name** text box, enter "StringLibraryTest" as the name of the project. Select **OK** to create the unit test project.



> **NOTE**
> In addition to an MSTest Test project, you can also use Visual Studio to create an xUnit test project for .NET Core.

3. Visual Studio creates the project and opens the *UnitTest1.cs* file in the code window.

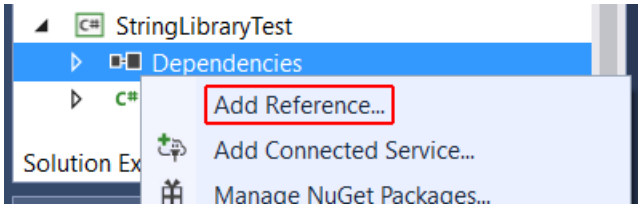



```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

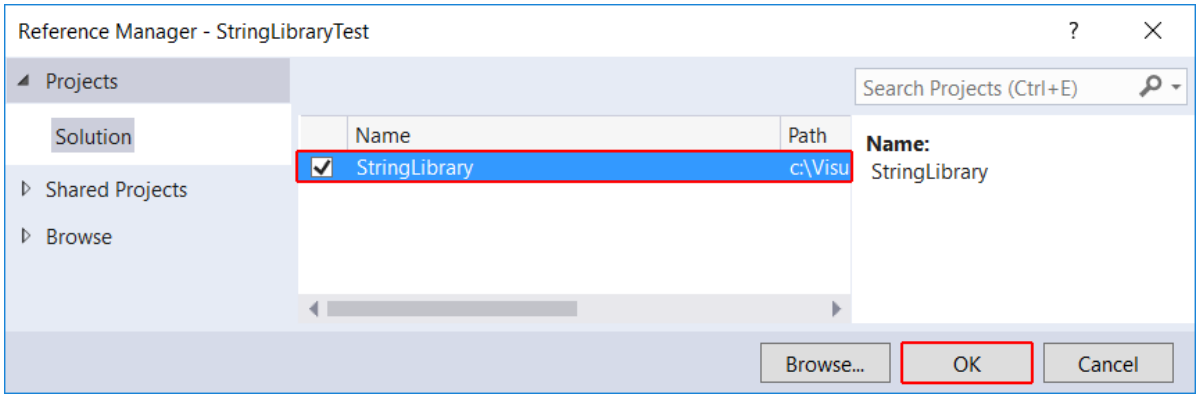
The source code created by the unit test template does the following:

- It imports the Microsoft.VisualStudio.TestTools.UnitTesting namespace, which contains the types used for unit testing.
- It applies the TestClassAttribute attribute to the `UnitTest1` class. Each test method in a test class tagged with the [TestMethod] attribute is executed automatically when the unit test is run.
- It applies the TestMethodAttribute attribute to define `TestMethod1` as a test method for automatic execution when the unit test is run.

4. In **Solution Explorer**, right-click the **Dependencies** node of the **StringLibraryTest** project and select **Add Reference** from the context menu.



5. In the **Reference Manager** dialog, expand the **Projects** node and check the box next to **StringLibrary**. Adding a reference to the `StringLibrary` assembly allows the compiler to find **StringLibrary** methods. Select the **OK** button. This adds a reference to your class library project, `StringLibrary`.



Adding and running unit test methods

When Visual Studio runs a unit test, it executes each method marked with the TestMethodAttribute attribute in a unit test class, the class to which the TestClassAttribute attribute is applied. A test method ends when the first failure is encountered or when all tests contained in the method have succeeded.

The most common tests call members of the Assert class. Many assert methods include at least two parameters, one of which is the expected test result and the other of which is the actual test result. Some of its most frequently called methods are shown in the table below.

ASSERT METHODS	FUNCTION
`Assert.AreEqual`	Verifies that two values or objects are equal. The assert fails if the values or objects are not equal.
`Assert.AreSame`	Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects.
`Assert.IsFalse`	Verifies that a condition is `false`. The assert fails if the condition is `true`.
`Assert.IsNotNull`	Verifies that an object is not `null`. The assert fails if the object is `null`.

You can also apply the ExpectedExceptionAttribute attribute to a test method. It indicates the type of exception a test method is expected to throw. The test fails if the specified exception is not thrown.

In testing the `StringLibrary.StartsWithUpper` method, you want to provide a number of strings that begin with an uppercase character. You expect the method to return `true` in these cases, so you can call the IsTrue method. Similarly, you want to provide a number of strings that begin with something other than an uppercase character. You expect the method to return `false` in these cases, so you can call the IsFalse method.

Since your library method handles strings, you also want to make sure that it successfully handles an empty string (`String.Empty`), a valid string that has no characters and whose Length is 0, and a `null` string that has not been initialized. If `StartsWithUpper` is called as an extension method on a String instance, it cannot be passed a `null` string. However, you can also call it directly as a static method and pass a single String argument.

You'll define three methods, each of which calls its Assert method repeatedly for each element in a string array. Because the test method fails as soon as it encounters the first failure, you'll call a method overload that allows you to pass a string that indicates the string value used in the method call.

To create the test methods:

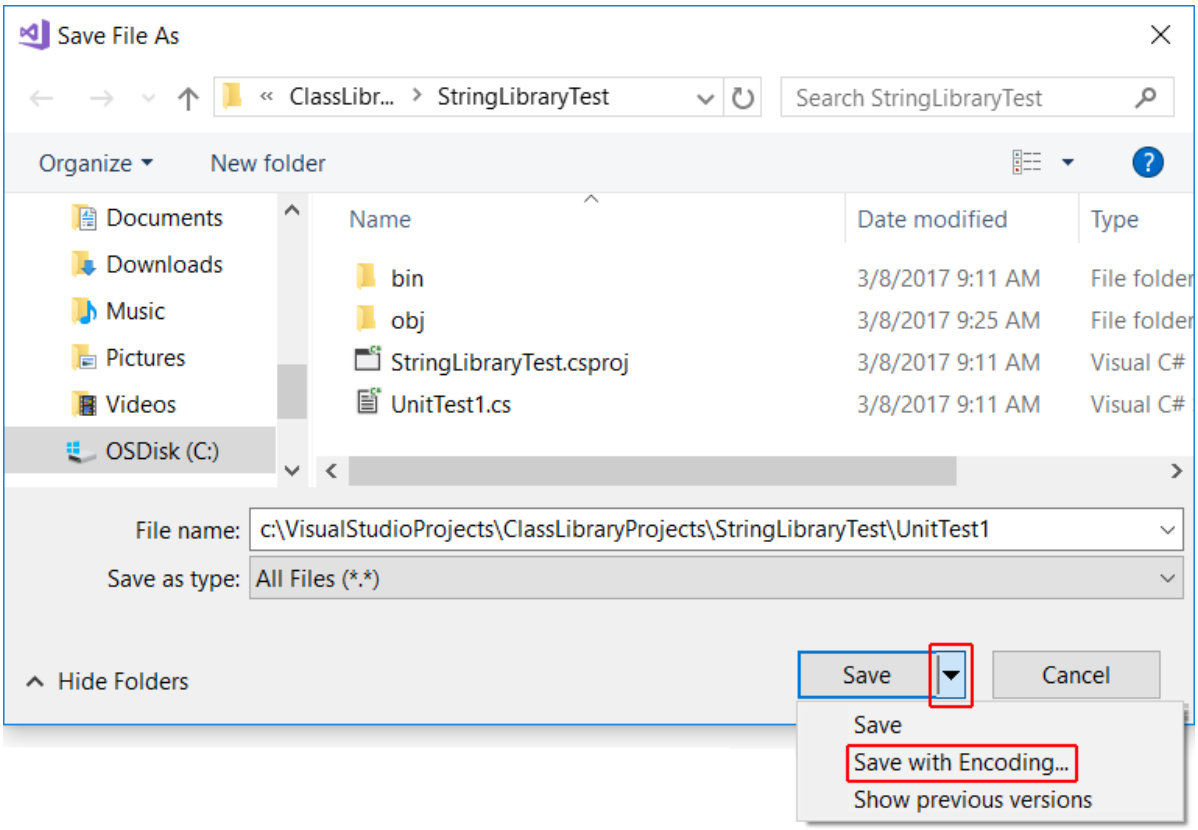
- C#
- Visual Basic

1. In the *UnitTest1.cs* code window, replace the code with the following code:

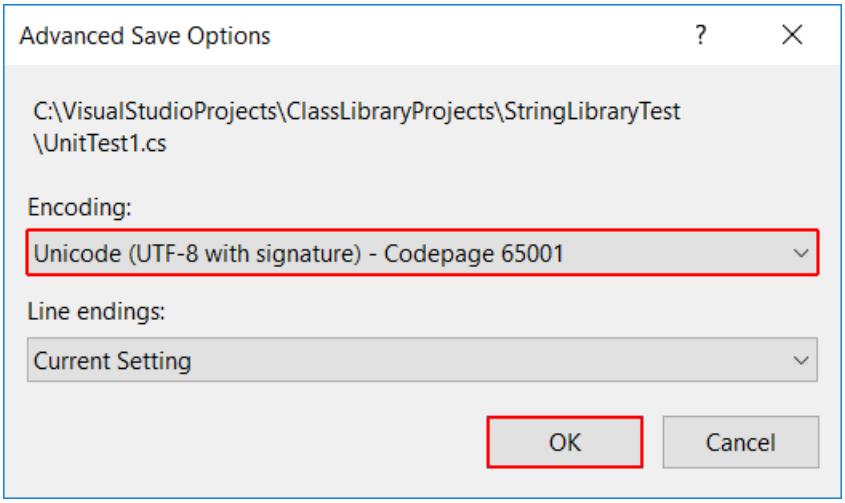
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

Note that your test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C), and the test of lowercase characters in the `TestDoesNotStartWithUpper` method includes the Greek small letter alpha (U+03B1) and the Cyrillic small letter Ghe (U+0433).

2. On the menu bar, select **File** > **Save UnitTest1.cs As**. In the **Save File As** dialog, select the arrow beside the **Save** button, and select **Save with Encoding**.

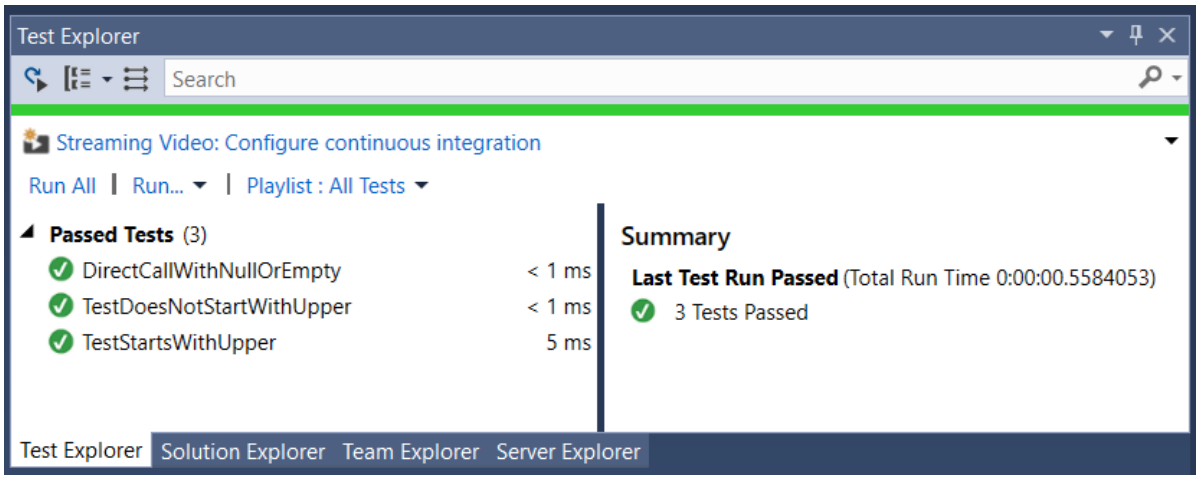


1. In the **Confirm Save As** dialog, select the **Yes** button to save the file.
2. In the **Advanced Save Options** dialog, select **Unicode (UTF-8 with signature) - Codepage 65001** from the **Encoding** drop-down list and select **OK**.



If you fail to save your source code as a UTF8-encoded file, Visual Studio may save it as an ASCII file. When that happens, the runtime doesn't accurately decode the UTF8 characters outside of the ASCII range, and the test results won't be accurate.

3. On the menu bar, select **Test** > **Run** > **All Tests**. The **Test Explorer** window opens and shows that the tests ran successfully. The three tests are listed in the **Passed Tests** section, and the **Summary** section reports the result of the test run.



Handling test failures

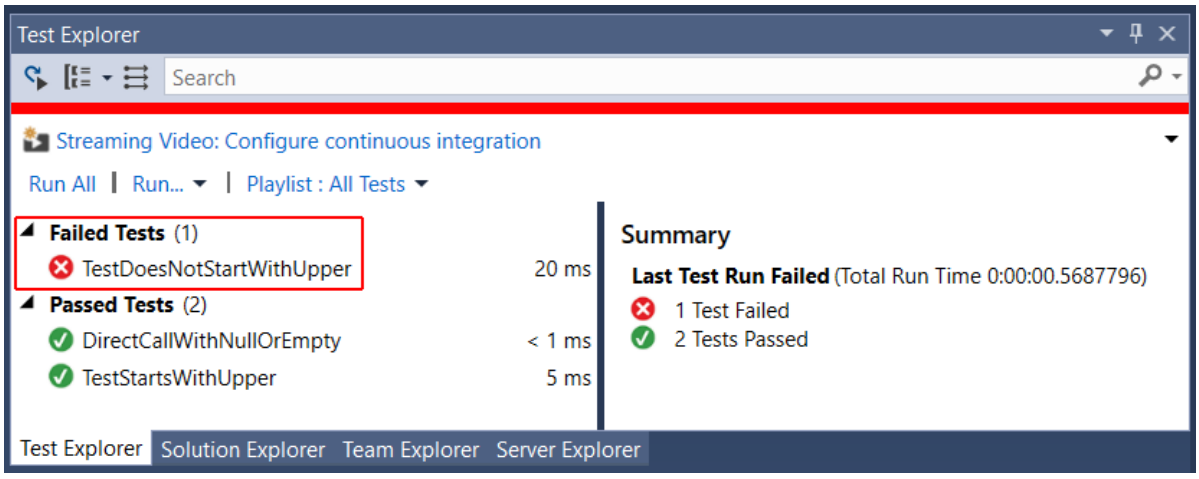
Your test run had no failures, but change it slightly so that one of the test methods fails:

1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error". You don't need to save the file because Visual Studio automatically saves open files when a solution is built to run tests.

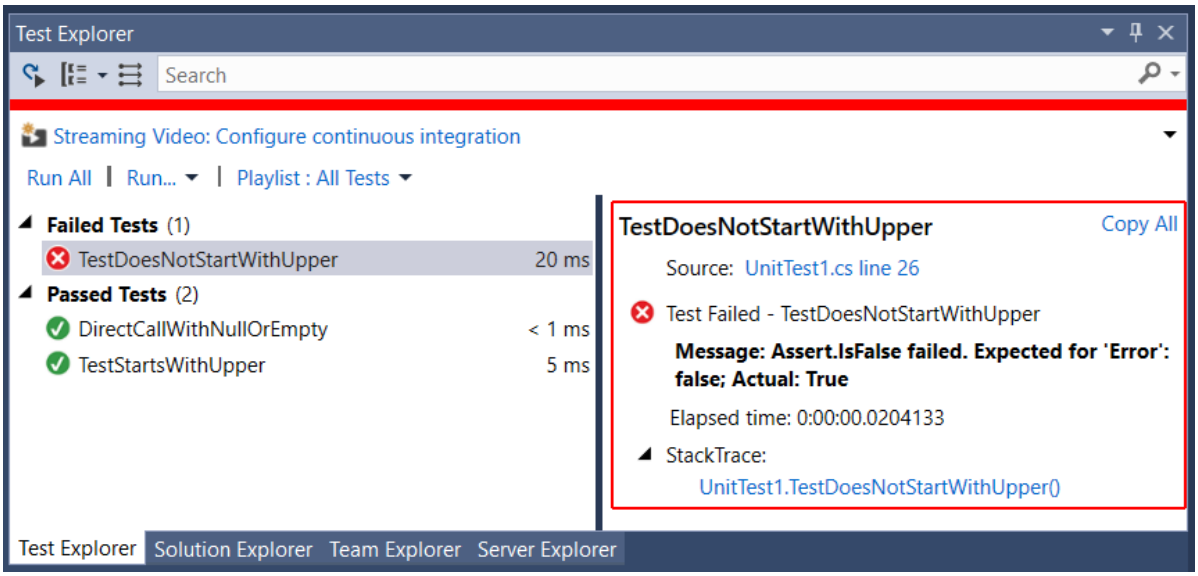
```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

```
Dim words() As String = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " }
```

2. Run the test by selecting **Test** > **Run** > **All Tests** from the menu bar. The **Test Explorer** window indicates that two tests succeeded and one failed.



3. In the **Failed Tests** section, select the failed test, `TestDoesNotStartWith`. The **Test Explorer** window displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, all strings in the array after "Error" were not tested.



4. Undo the modification you did in step 1 and remove the string "Error". Rerun the test and the tests will pass.

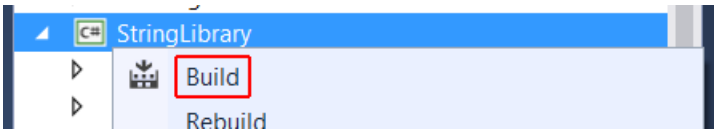
Testing the Release version of the library

You've been running your tests against the Debug version of the library. Now that your tests have all passed and you've adequately tested your library, you should run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

To test the Release build:

1. In the Visual Studio toolbar, change the build configuration from **Debug** to **Release**.

2. In **Solution Explorer**, right-click the **StringLibrary** project and select **Build** from the context menu to recompile the library.



3. Run the unit tests by choosing **Test** > **Run** > **All Tests** from the menu bar. The tests pass.

Now that you've finished testing your library, the next step is to make it available to callers. You can bundle it with one or more applications, or you can distribute it as a NuGet package. For more information, see Consuming a .NET Standard Class Library.

Consume a .NET Standard library in Visual Studio 2017

8/20/2019 • 5 minutes to read • Edit Online

Once you've created a .NET Standard class library by following the steps in Building a C# class library with .NET Core in Visual Studio 2017 or Building a Visual Basic class library with .NET Core in Visual Studio 2017, tested it in Testing a class library with .NET Core in Visual Studio 2017, and built a Release version of the library, the next step is to make it available to callers. You can do this in two ways:

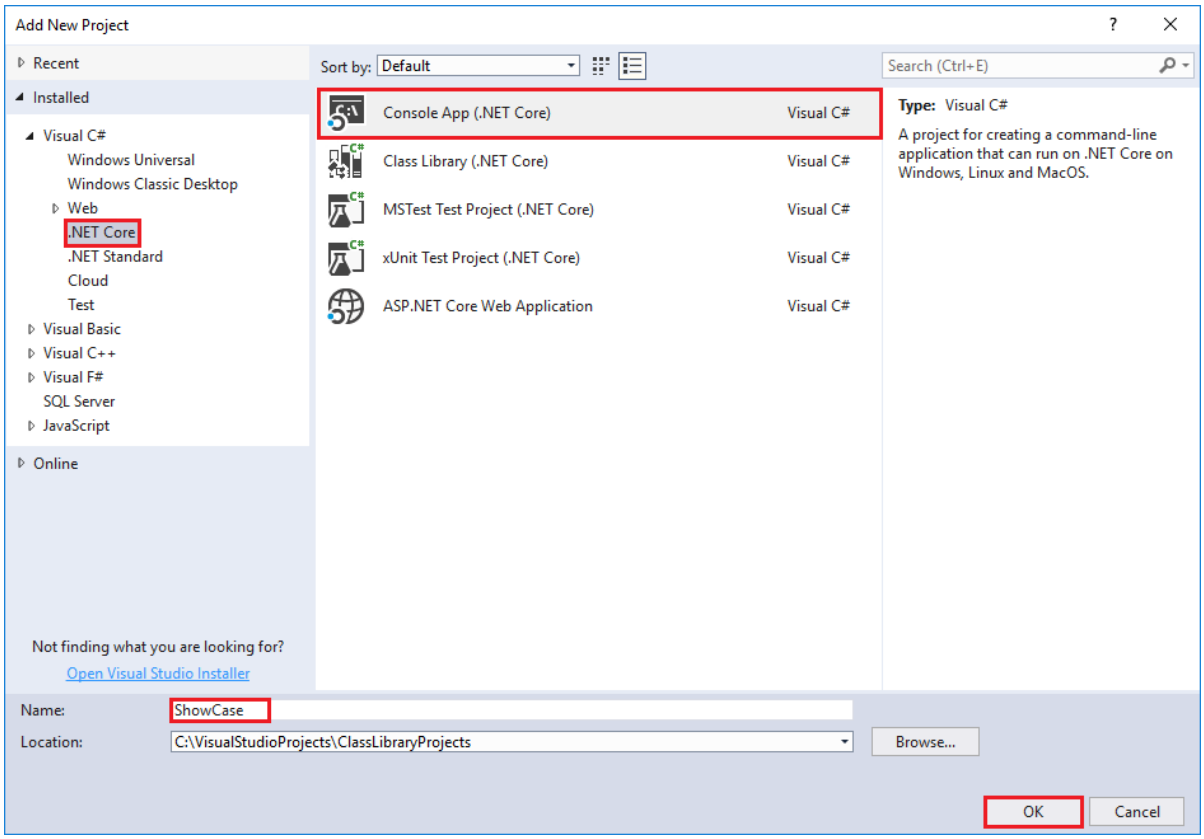
- If the library will be used by a single solution (for example, if it's a component in a single large application), you can include it as a project in your solution.
- If the library will be generally accessible, you can distribute it as a NuGet package.

Including a library as a project in a solution

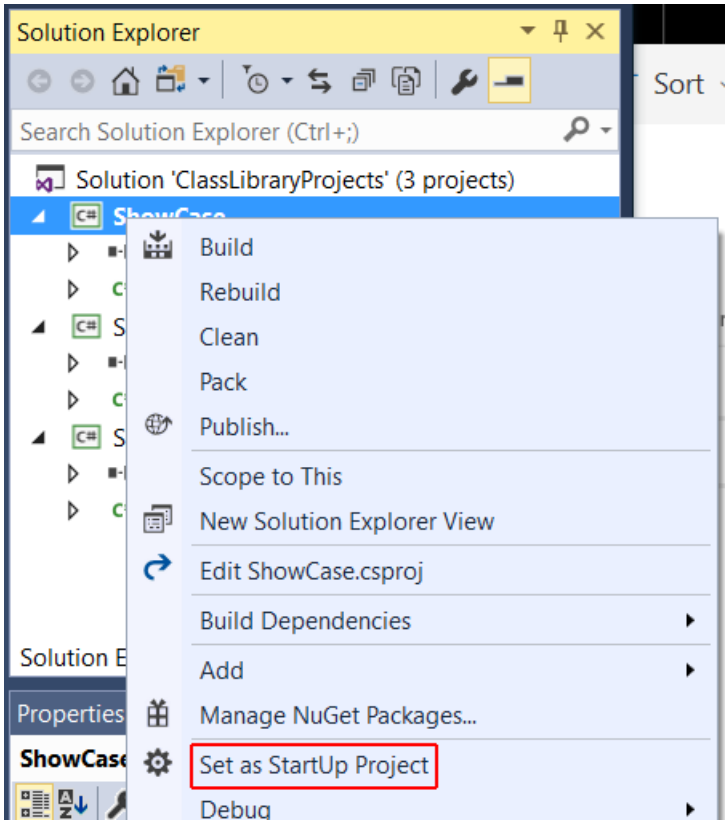
Just as you included unit tests in the same solution as your class library, you can include your application as part of that solution. For example, you can use your class library in a console application that prompts the user to enter a string and reports whether its first character is uppercase:

- C#
- Visual Basic

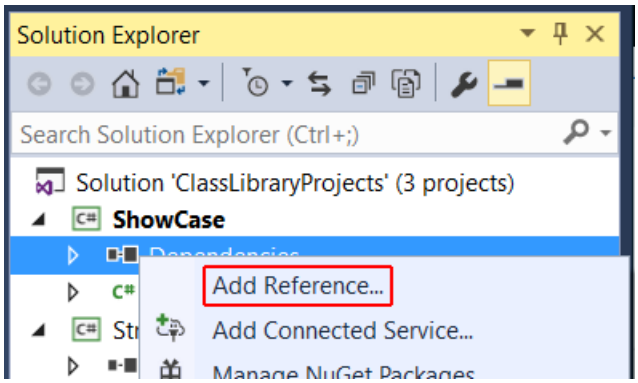
1. Open the `ClassLibraryProjects` solution you created in the Building a C# Class Library with .NET Core in Visual Studio 2017 topic. In **Solution Explorer**, right-click the **ClassLibraryProjects** solution and select **Add** > **New Project** from the context menu.
2. In the **Add New Project** dialog, expand the **Visual C#** node and select the **.NET Core** node followed by the **Console App (.NET Core)** project template. In the **Name** text box, type "ShowCase", and select the **OK** button.



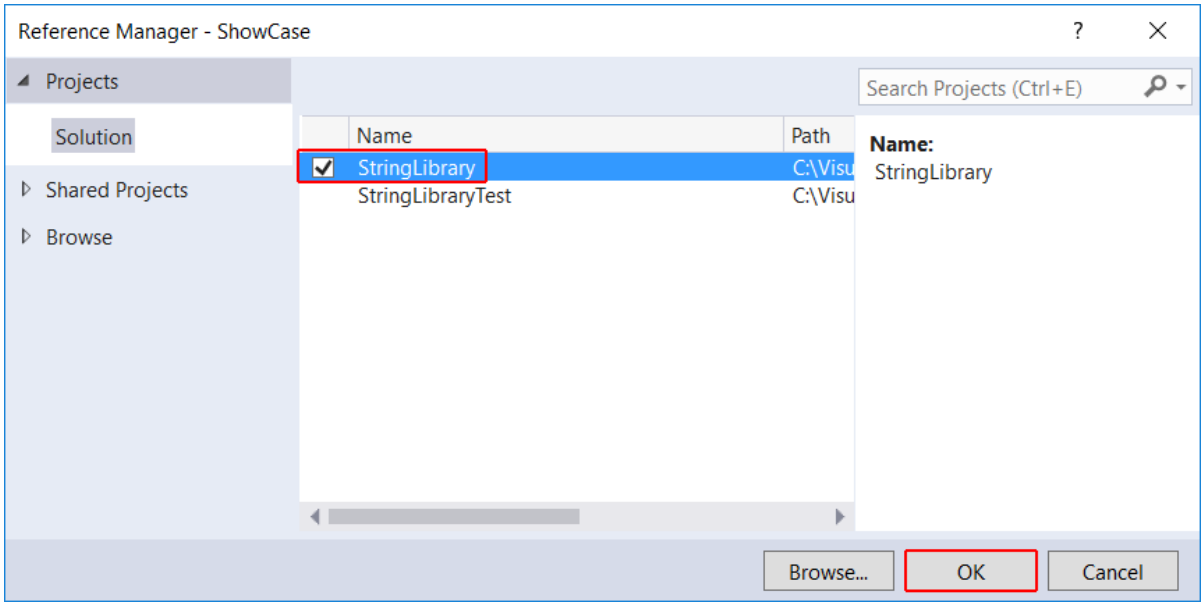
3. In **Solution Explorer**, right-click the **ShowCase** project and select **Set as StartUp Project** in the context menu.



4. Initially, your project doesn't have access to your class library. To allow it to call methods in your class library, you create a reference to the class library. In **Solution Explorer**, right-click the `ShowCase` project's **Dependencies** node and select **Add Reference**.



5. In the **Reference Manager** dialog, select **StringLibrary**, your class library project, and select the **OK** button.



6. In the code window for the *Program.cs* file, replace all of the code with the following code:

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
               ResetConsole();

            string input = Console.ReadLine();
            if (String.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{(input.StartsWithUpper() ? "Yes" : "No")}\n");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0) {
               Console.WriteLine("Press any key to continue...");
               Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine("\nPress <Enter> only to exit; otherwise, enter a string and press
<Enter>:\n");
            row = 3;
        }
    }
}
```

The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it is greater than or equal to 25, the code clears the console window and displays a message to the user.

The program prompts the user to enter a string. It indicates whether the string starts with an uppercase character. If the user presses the Enter key without entering a string, the application terminates, and the console window closes.

7. If necessary, change the toolbar to compile the **Debug** release of the `ShowCase` project. Compile and run the program by selecting the green arrow on the **ShowCase** button.

You can debug and publish the application that uses this library by following the steps in Debugging your Hello World application with Visual Studio 2017 and Publishing your Hello World Application with Visual Studio 2017.

Distributing the library in a NuGet package

You can make your class library widely available by publishing it as a NuGet package. Visual Studio does not support the creation of NuGet packages. To create one, you use the `dotnet` command line utility:

1. Open a console window. For example in the **Ask me anything** text box in the Windows taskbar, enter `Command Prompt` (or `cmd` for short), and open a console window by either selecting the **Command Prompt** desktop app or pressing Enter if it's selected in the search results.

2. Navigate to your library's project directory. Unless you've reconfigured the typical file location, it's in the *Documents\Visual Studio 2017\Projects\ClassLibraryProjects\StringLibrary* directory. The directory contains your source code and a project file, *StringLibrary.csproj*.

3. Issue the command `dotnet pack --no-build`. The `dotnet` utility generates a package with a *.nupkg* extension.

   > **TIP**
   >
   > If the directory that contains *dotnet.exe* is not in your PATH, you can find its location by entering `where dotnet.exe` in the console window.

For more information on creating NuGet packages, see How to Create a NuGet Package with Cross Platform Tools.

Prerequisites for .NET Core on Windows

7/19/2019 • 3 minutes to read • Edit Online

This article shows the supported OS versions in order to run .NET Core applications on Windows. The supported OS versions and dependencies that follow apply to the three ways of developing .NET Core apps on Windows:

- Command line
- Visual Studio
- Visual Studio Code

Also, if you're developing on Windows using Visual Studio 2017, the Prerequisites with Visual Studio 2017 section goes in more detail about minimum versions supported for .NET Core development.

.NET Core supported operating systems

The following articles have a complete list of .NET Core supported operating systems per version:

- .NET Core 3.0 (Preview)
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 1.0

For download links and more information, see .NET downloads to download the latest version or .NET downloads archive for older versions.

.NET Core dependencies

.NET Core 1.1 and earlier versions require the Visual C++ Redistributable when running on Windows versions earlier than Windows 10 and Windows Server 2016. This dependency is automatically installed by the .NET Core installer.

Microsoft Visual C++ 2015 Redistributable Update 3 must be manually installed when:

- Installing .NET Core with the installer script.
- Deploying a self-contained .NET Core application.
- Building the product from source.
- Installing .NET Core via a *.zip* file. This can include build/CI/CD servers.

> **NOTE**
>
> **For Windows 8.1 and earlier versions, or Windows Server 2012 R2 and earlier versions:**
>
> Make sure that your Windows installation is up-to-date and includes KB2999226, which can be installed through Windows Update. If you don't have this update installed, you'll see an error like the following when you launch a .NET Core application:
>
> ```
> The program can't start because api-ms-win-crt-runtime-l1-1-0.dll is missing from your computer. Try reinstalling the program to fix this problem.
> ```
>
> **For Windows 7 or Windows Server 2008 R2:**
>
> In addition to KB2999226, make sure you also have KB2533623 installed. If you don't have this update installed, you'll see an error similar to the following when you launch a .NET Core application:
>
> `The library hostfxr.dll was found, but loading it from C:\<path_to_app>\hostfxr.dll failed`.

Prerequisites for .NET Core 3.0 Preview 3

.NET Core 3.0 Preview 3 has the same prerequisites as other versions of .NET Core. However, if you want to use Visual Studio to create .NET Core 3.0 projects, you must use the Visual Studio 2019. Visual Studio 2019 can be installed side-by-side with other versions of Visual Studio without conflict.

Prerequisites with Visual Studio 2017

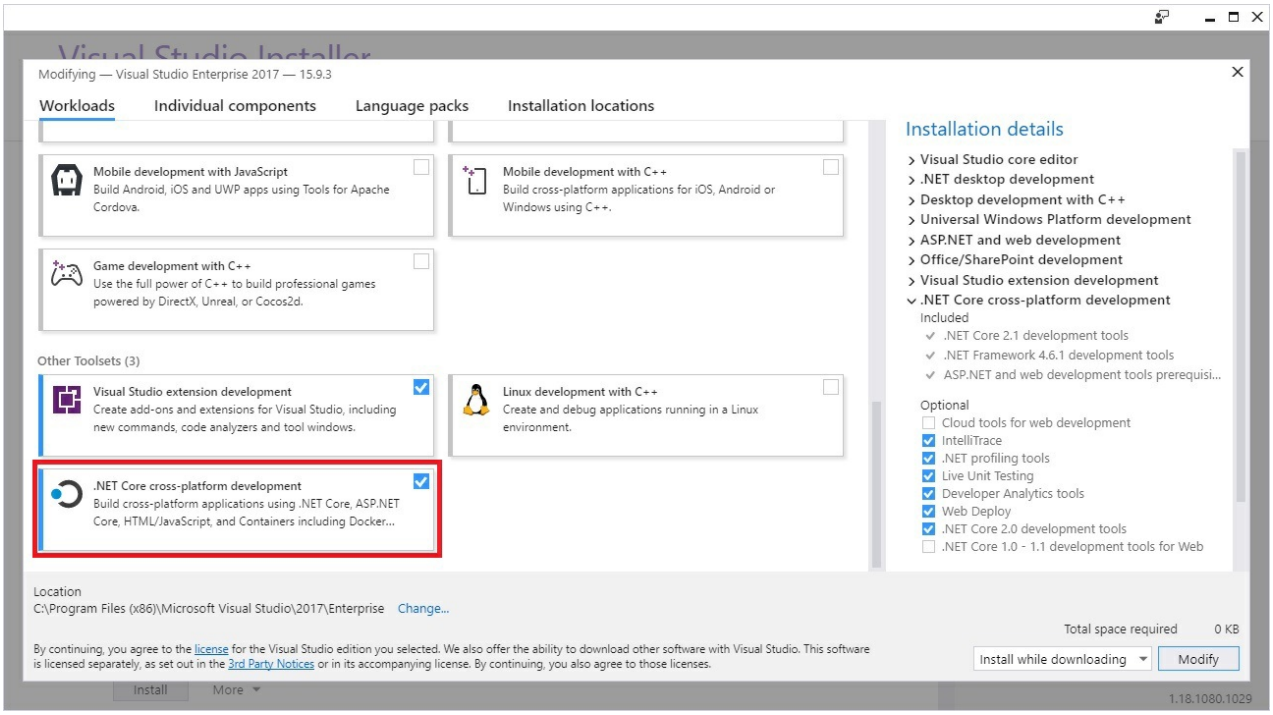
You can use any editor to develop .NET Core applications using the .NET Core SDK. Visual Studio 2017 provides an integrated development environment for .NET Core apps on Windows.

You can read more about the changes in Visual Studio 2017 in the release notes.

- .NET Core 2.x
- .NET Core 1.x

To develop .NET Core apps in Visual Studio 2017 using the .NET Core 2.2 SDK:

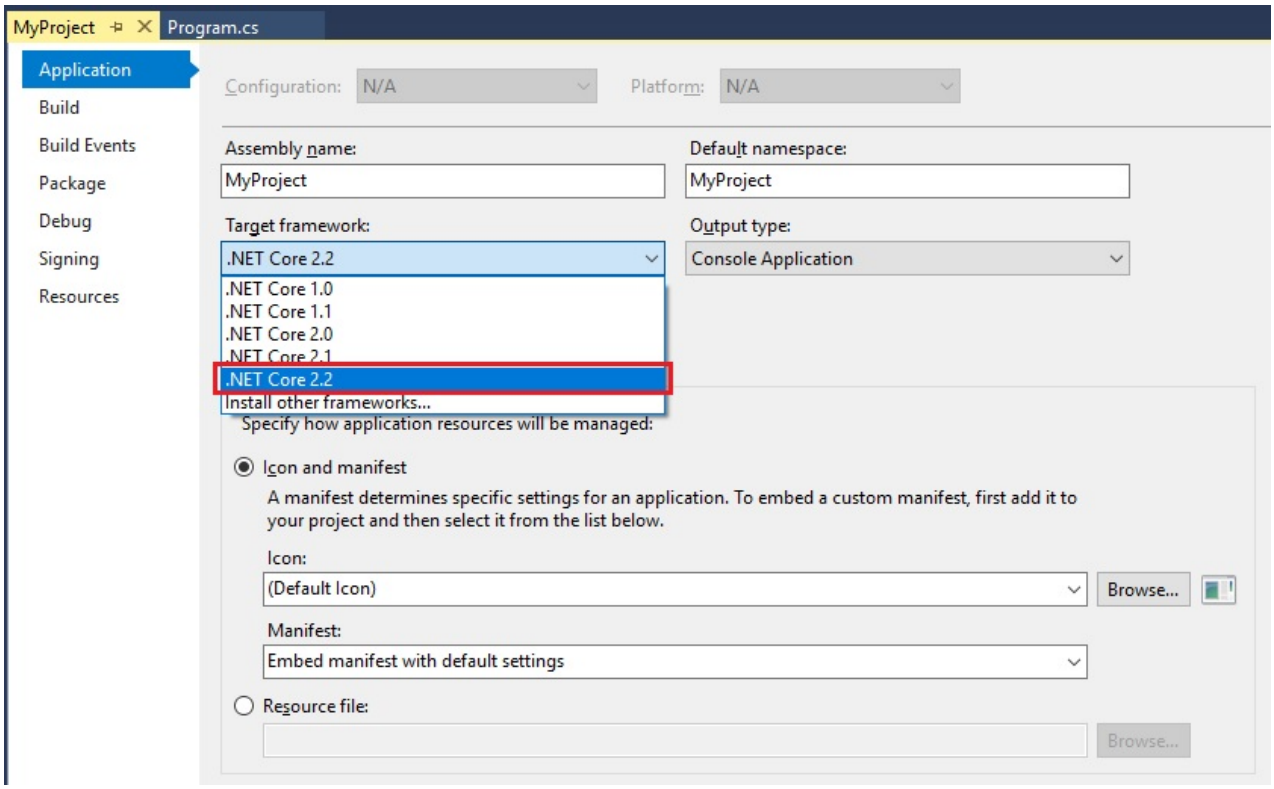
1. Download and install Visual Studio 2017 version 15.9.0 or higher with the **.NET Core cross-platform development** workload (in the **Other Toolsets** section) selected.



After the **.NET Core cross-platform development** toolset is installed, Visual Studio usually installs a previous version of the .NET Core SDK. For example, Visual Studio 2017 15.9 uses .NET Core 2.1 SDK by default after the workload is installed.

To update Visual Studio to use .NET Core 2.2 SDK:

1. Install the .NET Core 2.2 SDK.
2. If you want your project to use the latest .NET Core runtime, retarget existing or new .NET Core projects to .NET Core 2.2 using the following instructions:
   - On the **Project** menu, choose **Properties**.
   - In the **Target framework** selection menu, set the value to **.NET Core 2.2**.



Once you have Visual Studio configured with .NET Core 2.2 SDK, you can do the following actions:

- Open, build, and run existing .NET Core 1.x and 2.x projects.
- Retarget .NET Core 1.x and 2.x projects to .NET Core 2.2, build, and run.
- Create new .NET Core 2.2 projects.

TIP

To verify your Visual Studio version:

- On the **Help** menu, choose **About Microsoft Visual Studio**.
- In the **About Microsoft Visual Studio** dialog, verify the version number.
  - For .NET Core 3.0 Preview 3 apps, Visual Studio 2019 version 16.0 or higher.
  - For .NET Core 2.2 apps, Visual Studio 2017 version 15.9 or higher.
  - For .NET Core 2.1 apps, Visual Studio 2017 version 15.7 or higher.
  - For .NET Core 1.x apps, Visual Studio 2017 version 15.0 or higher.

Prerequisites for .NET Core on macOS

7/24/2019 • 3 minutes to read • Edit Online

This article shows you the supported macOS versions and .NET Core dependencies that you need to develop, deploy, and run .NET Core applications on macOS machines. The supported OS versions and dependencies that follow apply to the three ways of developing .NET Core apps on a Mac: via the command-line with your favorite editor, Visual Studio Code, and Visual Studio for Mac.

Supported macOS versions

- .NET Core 2.x
- .NET Core 1.x
- .NET Core 3.0

.NET Core 2.x is supported on the following versions of macOS:

- macOS 10.12 "Sierra" and later versions

See .NET Core 2.1 Supported OS Versions and .NET Core 2.2 Supported OS Versions for the complete list of .NET Core 2.1 and .NET Core 2.2 supported operating systems, distributions and versions, out of support OS versions, and lifecycle policy links.

For download links and more information, see .NET Core 2.2 downloads or .NET Core 2.1 downloads.

.NET Core dependencies

- .NET Core 2.x
- .NET Core 1.x
- .NET Core 3.0

Download and install the .NET Core SDK from .NET Downloads. If you have problems with the installation on macOS, consult the Known issues topic for the version you have installed.

Increase the maximum open file limit (.NET Core versions before .NET Core SDK 2.0.2)

In older .NET Core versions (before .NET Core SDK 2.0.2), the default open file limit on macOS may not be sufficient for some .NET Core workloads, such as restoring projects or running unit tests.

You can increase this limit by following these steps:

1. Using a text editor, create a new file */Library/LaunchDaemons/limit.maxfiles.plist*, and save the file with this content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>Label</key>
    <string>limit.maxfiles</string>
    <key>ProgramArguments</key>
    <array>
      <string>launchctl</string>
      <string>limit</string>
      <string>maxfiles</string>
      <string>2048</string>
      <string>4096</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>ServiceIPC</key>
    <false/>
  </dict>
</plist>
```

2. In a terminal window, run the following command:

```
echo 'ulimit -n 2048' | sudo tee -a /etc/profile
```

3. Reboot your Mac to apply these settings.

Visual Studio for Mac

You can use any editor to develop .NET Core applications using the .NET Core SDK. However, if you want to develop .NET Core applications on a Mac in an integrated development environment, you can use Visual Studio for Mac.

.NET Core development on macOS with Visual Studio for Mac requires:

- A supported version of the macOS operating system
- OpenSSL (.NET Core 1.x only; .NET Core 2.x uses security services available natively in macOS)
- .NET Core SDK for Mac
- Visual Studio for Mac

Prerequisites for .NET Core on Linux

4/16/2019 • 4 minutes to read • Edit Online

This article shows the dependencies needed to develop .NET Core applications on Linux. The supported Linux distributions/versions, and dependencies that follow apply to the two ways of developing .NET Core apps on Linux:

- Command-line with your favorite editor
- Visual Studio Code

> **NOTE**
> The .NET Core SDK package is not required for production servers/environments. Only the .NET Core runtime package is needed for apps deployed to production environments. The .NET Core runtime is deployed with apps as part of a self-contained deployment, however, it must be deployed for Framework-dependent deployed apps separately. For more information about framework-dependent and self-contained deployment types, see .NET Core application deployment. Also see Self-contained Linux applications for specific guidelines.

Supported Linux versions

- .NET Core 2.x
- .NET Core 1.x
- .NET Core 3.0 Preview 1

.NET Core 2.x treats Linux as a single operating system. There is a single Linux build (per chip architecture) for supported Linux distributions.

For download links and more information, see .NET Core 2.2 downloads or .NET Core 2.1 downloads.

.NET Core 2.x is supported on the following Linux distributions/versions:

- Red Hat Enterprise Linux 7, 6 - 64-bit (`x86_64` or `amd64`)
- CentOS 7 - 64-bit (`x86_64` or `amd64`)
- Oracle Linux 7 - 64-bit (`x86_64` or `amd64`)
- Fedora 28, 27 - 64-bit (`x86_64` or `amd64`)
- Debian 9 (64-bit, `arm32`), 8.7 or later versions - 64-bit (`x86_64` or `amd64`)
- Ubuntu 18.04 (64-bit, `arm32`), 16.04, 14.04 - 64-bit (`x86_64` or `amd64`)
- Linux Mint 18, 17 - 64-bit (`x86_64` or `amd64`)
- openSUSE 42.3 or later versions - 64-bit (`x86_64` or `amd64`)
- SUSE Enterprise Linux (SLES) 12 Service Pack 2 or later - 64-bit (`x86_64` or `amd64`)
- Alpine Linux 3.7 or later versions - 64-bit (`x86_64` or `amd64`)

See .NET Core 2.1 Supported OS Versions and .NET Core 2.2 Supported OS Versions for the complete list of .NET Core 2.1 and .NET Core 2.2 supported operating systems, distributions and versions, out of support OS versions, and lifecycle policy links.

Linux distribution dependencies

The following are intended to be examples. The exact versions and names may vary slightly on your Linux

distribution of choice.

Ubuntu

Ubuntu distributions require the following libraries installed:

- liblttng-ust0
- libcurl3 (for 14.x and 16.x)
- libcurl4 (for 18.x)
- libssl1.0.0
- libkrb5-3
- zlib1g
- libicu52 (for 14.x)
- libicu55 (for 16.x)
- libicu57 (for 17.x)
- libicu60 (for 18.x)

For versions earlier than .NET Core 2.1, following dependencies are also required:

- libunwind8
- libuuid1

CentOS and Fedora

CentOS distributions require the following libraries installed:

- lttng-ust
- libcurl
- openssl-libs
- krb5-libs
- libicu
- zlib

Fedora users: If your openssl's version >= 1.1, you'll need to install compat-openssl10.

For versions earlier than .NET Core 2.1, following dependencies are also required:

- libunwind
- libuuid

For more information about the dependencies, see Self-contained Linux applications.

Installing .NET Core dependencies with the native installers

.NET Core native installers are available for supported Linux distributions/versions. The native installers require admin (sudo) access to the server. The advantage of using a native installer is that all of the .NET Core native dependencies are installed. Native installers also install the .NET Core SDK system-wide.

On Linux, there are two installer package choices:

- Using a feed-based package manager, such as apt-get for Ubuntu, or yum for CentOS/RHEL.
- Using the packages themselves, DEB or RPM.

Scripting Installs with the .NET Core installer script

The dotnet-install scripts are used to perform a non-admin install of the CLI toolchain and the shared runtime. You can download the script from https://dot.net/v1/dotnet-install.sh.

The script defaults to installing the latest "LTS" version, which is currently .NET Core 1.1. To install .NET Core 2.1, run the script with the following switch:

```
./dotnet-install.sh -c Current
```

The installer bash script is used in automation scenarios and non-admin installations. This script also reads PowerShell switches, so they can be used with the script on Linux/OS X systems.

Troubleshoot

If you have problems with a .NET Core installation on a supported Linux distribution/version, consult the following topics for your installed distributions/versions:

- .NET Core 3.0 known issues
- .NET Core 2.2 known issues
- .NET Core 2.1 known issues
- .NET Core 1.1 known issues
- .NET Core 1.0 known issues

What's new in .NET Core 3.0 (Preview 8)

8/27/2019 • 22 minutes to read • Edit Online

This article describes what is new in .NET Core 3.0 (through preview 8). One of the biggest enhancements is support for Windows desktop applications (Windows only). By using the .NET Core 3.0 SDK component Windows Desktop, you can port your Windows Forms and Windows Presentation Foundation (WPF) applications. To be clear, the Windows Desktop component is only supported and included on Windows. For more information, see the Windows desktop section later in this article.

.NET Core 3.0 adds support for C# 8.0. It's highly recommended that you use the latest release of Visual Studio Preview, or Visual Studio Code with the OmniSharp extension.

Download and get started with .NET Core 3.0 preview 8 right now on Windows, macOS, or Linux.

For more information about each preview release, see the following announcements:

- .NET Core 3.0 Preview 8 announcement
- .NET Core 3.0 Preview 7 announcement
- .NET Core 3.0 Preview 6 announcement
- .NET Core 3.0 Preview 5 announcement
- .NET Core 3.0 Preview 4 announcement
- .NET Core 3.0 Preview 3 announcement
- .NET Core 3.0 Preview 2 announcement
- .NET Core 3.0 Preview 1 announcement

Production supported preview

.NET Core Preview 8 is considered production ready by Microsoft and is fully supported. Starting with preview 7, releases will focus on polishing .NET Core 3.0 instead of adding new features. For more information about what has changed in preview 8, see the preview 8 announcement.

If you're using a previous preview release, you must move to Preview 8 for continued "Go Live" support.

.NET Core SDK Windows Installer

The MSI installer for Windows has changed starting with .NET Core 3.0. The SDK installers will now upgrade SDK feature-band releases in place. Feature bands are defined in the *hundreds* groups in the *patch* section of the version number. For example, **3.0.*101*** and **3.0.*201*** are versions in two different feature bands while **3.0.*101*** and **3.0.*199*** are in the same feature band. And, when .NET Core SDK **3.0.*101*** is installed, .NET Core SDK **3.0.*100*** will be removed from the machine if it exists. When .NET Core SDK **3.0.*200*** is installed on the same machine, .NET Core SDK **3.0.*101*** won't be removed.

For more information about versioning, see Overview of how .NET Core is versioned.

C# 8.0 preview

.NET Core 3.0 supports C# 8 preview. For more information about C# 8.0 features, see What's new in C# 8.0.

.NET Standard 2.1

Even though .NET Core 3.0 supports **.NET Standard 2.1**, the default `dotnet new classlib` template generates a

project that targets **.NET Standard 2.0**. To target **.NET Standard 2.1**, edit your project file and change the `TargetFramework` property to `netstandard2.1`:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

If you're using Visual Studio, you need Visual Studio 2019, as Visual Studio 2017 doesn't support **.NET Standard 2.1** or **.NET Core 3.0**.

Improved .NET Core Version APIs

Starting with .NET Core 3.0, the version APIs provided with .NET Core now return the information you expect. For example:

```
System.Console.WriteLine($"Environment.Version: {System.Environment.Version}");

// Old result
//   Environment.Version: 4.0.30319.42000
//
// New result
//   Environment.Version: 3.0.0
```

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");

// Old result
//   RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71
//
// New result
//   RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-27615-11
```

WARNING

Breaking change. This is technically a breaking change because the versioning scheme has changed.

.NET Platform-Dependent Intrinsics

APIs have been added that allow access to certain perf-oriented CPU instructions, such as the **SIMD** or **Bit Manipulation instruction** sets. These instructions can help achieve significant performance improvements in certain scenarios, such as processing data efficiently in parallel.

Where appropriate, the .NET libraries have begun using these instructions to improve performance.

For more information, see .NET Platform Dependent Intrinsics.

Default executables

.NET Core now builds framework-dependent executables by default. This behavior is new for applications that use a globally installed version of .NET Core. Previously, only self-contained deployments would produce an executable.

During `dotnet build` or `dotnet publish`, an executable is created that matches the environment and platform of the SDK you're using. You can expect the same things with these executables as you would other native executables, such as:

- You can double-click on the executable.
- You can launch the application from a command prompt directly, such as `myapp.exe` on Windows, and `./myapp` on Linux and macOS.

Single-file executables

The `dotnet publish` command supports packaging your app into a platform-specific single-file executable. The executable is self-extracting and contains all dependencies (including native) that are required to run your app. When the app is first run, the application is extracted to a directory based on the app name and build identifier. Startup is faster when the application is run again. The application doesn't need to extract itself a second time unless a new version was used.

To publish a single-file executable, set the `PublishSingleFile` in your project or on the command line with the `dotnet publish` command:

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

-or-

```
dotnet publish -r win10-x64 /p:PublishSingleFile=true
```

For more information about single-file publishing, see the single-file bundler design document.

Assembly linking

The .NET core 3.0 SDK comes with a tool that can reduce the size of apps by analyzing IL and trimming unused assemblies.

Self-contained apps include everything needed to run your code, without requiring .NET to be installed on the host computer. However, many times the app only requires a small subset of the framework to function, and other unused libraries could be removed.

.NET Core now includes a setting that will use the IL linker tool to scan the IL of your app. this tool detects what code is required, and then trims unused libraries. This tool can significantly reduce the deployment size of some apps.

To enable this tool, add the `<PublishTrimmed>` setting in your project and publish a self-contained app:

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

```
dotnet publish -r <rid> -c Release
```

As an example, the basic "hello world" new console project template that is included, when published, hits about 70 MB in size. By using `<PublishTrimmed>`, that size is reduced to about 30 MB.

It's important to consider that applications or frameworks (including ASP.NET Core and WPF) that use reflection or related dynamic features, will often break when trimmed. This breakage occurs because the linker doesn't know about this dynamic behavior and can't determine which framework types are required for reflection. The IL Linker tool can be configured to be aware of this scenario.

Above all else, be sure to test your app after trimming.

For more information about the IL Linker tool, see the documentation or visit the mono/linker repo.

Tiered compilation

Tiered compilation (TC) is on by default with .NET Core 3.0. This feature enables the runtime to more adaptively use the Just-In-Time (JIT) compiler to get better performance.

The main benefit of TC is to enable (re-)jitting methods with a lower-quality-but-faster tier or a higher-quality-but-slower tier. This helps increase performance of an application as it goes through various stages of execution, from startup through steady-state. This contrasts with the non-TC approach, where every method is compiled a single way (the same as the high-quality tier), which is biased to steady-state over startup performance.

To enable Quick JIT (tier 0 jitted code), use this setting in your project file:

```
<PropertyGroup>
  <TieredCompilationQuickJit>true</TieredCompilationQuickJit>
</PropertyGroup>
```

To disable TC completely, use this setting in your project file:

```
<TieredCompilation>false</TieredCompilation>
```

ReadyToRun images

You can improve the startup time of your .NET Core application by compiling your application assemblies as ReadyToRun (R2R) format. R2R is a form of ahead-of-time (AOT) compilation.

R2R binaries improve startup performance by reducing the amount of work the just-in-time (JIT) compiler needs to do as your application loads. The binaries contain similar native code compared to what the JIT would produce. However, R2R binaries are larger because they contain both intermediate language (IL) code, which is still needed for some scenarios, and the native version of the same code. R2R is only available when you publish a self-contained app that targets specific runtime environments (RID) such as Linux x64 or Windows x64.

To compile your project as ReadyToRun, do the following:

1. Add the `<PublishReadyToRun>` setting to your project

   ```
   <PropertyGroup>
     <PublishReadyToRun>true</PublishReadyToRun>
   </PropertyGroup>
   ```

2. Publish a self-contained app. For example, this command creates a self-contained app for the 64-bit version of Windows:

   ```
   dotnet publish -c Release -r win-x64 --self-contained true
   ```

Cross platform/architecture restrictions

The ReadyToRun compiler doesn't currently support cross-targeting. You must compile on a given target. For example, if you want R2R images for Windows x64, you need to run the publish command on that environment.

Exceptions to cross-targeting:

- Windows x64 can be used to compile Windows ARM32, ARM64, and x86 images.
- Windows x86 can be used to compile Windows ARM32 images.
- Linux x64 can be used to compile Linux ARM32 and ARM64 images.

Build copies dependencies

The `dotnet build` command now copies NuGet dependencies for your application from the NuGet cache to the build output folder. Previously, dependencies were only copied as part of `dotnet publish`.

There are some operations, like linking and razor page publishing that will still require publishing.

Local tools

.NET Core 3.0 introduces local tools. Local tools are similar to global tools but are associated with a particular location on disk. Local tools aren't available globally and are distributed as NuGet packages.

> **WARNING**
> If you tried local tools in .NET Core 3.0 Preview 1, such as running `dotnet tool restore` or `dotnet tool install`, delete the local tools cache folder. Otherwise, local tools won't work on any newer release. This folder is located at:
>
> On macOS, Linux: `rm -r $HOME/.dotnet/toolResolverCache`
>
> On Windows: `rmdir /s %USERPROFILE%\.dotnet\toolResolverCache`

Local tools rely on a manifest file name `dotnet-tools.json` in your current directory. This manifest file defines the tools to be available at that folder and below. You can distribute the manifest file with your code to ensure that anyone who works with your code can restore and use the same tools.

For both global and local tools, a compatible version of the runtime is required. Many tools currently on NuGet.org target .NET Core Runtime 2.1. To install these tools globally or locally, you would still need to install the NET Core 2.1 Runtime.

Major-version Roll Forward

.NET Core 3.0 introduces an opt-in feature that allows your app to roll forward to the latest major version of .NET Core. Additionally, a new setting has been added to control how roll forward is applied to your app. This can be configured in the following ways:

- Project file property: `RollForward`
- Runtime configuration file property: `rollForward`
- Environment variable: `DOTNET_ROLL_FORWARD`
- Command-line argument: `--roll-forward`

One of the following values must be specified. If the setting is omitted, **Minor** is the default.

- **LatestPatch**
  Roll forward to the highest patch version. This disables minor version roll forward.
- **Minor**
  Roll forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the **LatestPatch** policy is used.

- **Major**
  Roll forward to lowest higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the **Minor** policy is used.
- **LatestMinor**
  Roll forward to highest minor version, even if requested minor version is present. Intended for component hosting scenarios.
- **LatestMajor**
  Roll forward to highest major and highest minor version, even if requested major is present. Intended for component hosting scenarios.
- **Disable**
  Don't roll forward. Only bind to specified version. This policy isn't recommended for general use because it disables the ability to roll forward to the latest patches. This value is only recommended for testing.

Besides the **Disable** setting, all settings will use the highest available patch version.

Windows desktop

.NET Core 3.0 supports Windows desktop applications using Windows Presentation Foundation (WPF) and Windows Forms. These frameworks also support using modern controls and Fluent styling from the Windows UI XAML Library (WinUI) via XAML islands.

The Windows Desktop component is part of the Windows .NET Core 3.0 SDK.

You can create a new WPF or Windows Forms app with the following `dotnet` commands:

```
dotnet new wpf
dotnet new winforms
```

Visual Studio 2019 adds **New Project** templates for .NET Core 3.0 Windows Forms and WPF.

For more information about how to port an existing .NET Framework application, see Port WPF projects and Port Windows Forms projects.

COM-callable components - Windows Desktop

On Windows, you can now create COM-callable managed components. This capability is critical to use .NET Core with COM add-in models and also to provide parity with .NET Framework.

Unlike .NET Framework where the *mscoree.dll* was used as the COM server, .NET Core will add a native launcher dll to the *bin* directory when you build your COM component.

For an example of how to create a COM component and consume it, see the COM Demo.

MSIX Deployment - Windows Desktop

MSIX is a new Windows application package format. It can be used to deploy .NET Core 3.0 desktop applications to Windows 10.

The Windows Application Packaging Project, available in Visual Studio 2019, allows you to create MSIX packages with self-contained .NET Core applications.

The .NET Core project file must specify the supported runtimes in the `<RuntimeIdentifiers>` property:

```
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

WinForms high DPI

.NET Core Windows Forms applications can set high DPI mode with Application.SetHighDpiMode(HighDpiMode). The `SetHighDpiMode` method sets the corresponding high DPI mode unless the setting has been set by other means like `App.Manifest` or P/Invoke before `Application.Run`.

The possible `highDpiMode` values, as expressed by the System.Windows.Forms.HighDpiMode enum are:

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`
- `DpiUnawareGdiScaled`

For more information about high DPI modes, see High DPI Desktop Application Development on Windows.

Ranges and indices

The new System.Index type can be used for indexing. You can create one from an `int` that counts from the beginning, or with a prefix `^` operator (C#) that counts from the end:

```
Index i1 = 3;  // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

There's also the System.Range type, which consists of two `Index` values, one for the start and one for the end, and can be written with a `x..y` range expression (C#). You can then index with a `Range`, which produces a slice:

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

For more information, see the ranges and indices tutorial.

Async streams

The IAsyncEnumerable<T> type is a new asynchronous version of IEnumerable<T>. The language lets you `await foreach` over `IAsyncEnumerable<T>` to consume their elements, and use `yield return` to them to produce elements.

The following example demonstrates both production and consumption of async streams. The `foreach` statement is async and itself uses `yield return` to produce an async stream for callers. This pattern (using `yield return`) is the recommended model for producing async streams.

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
    }
}
```

In addition to being able to `await foreach`, you can also create async iterators, for example, an iterator that returns an `IAsyncEnumerable/IAsyncEnumerator` that you can both `await` and `yield` in. For objects that need to be disposed, you can use `IAsyncDisposable`, which various BCL types implement, such as `Stream` and `Timer`.

For more information, see the async streams tutorial.

IEEE Floating-point improvements

Floating point APIs are being updated to comply with IEEE 754-2008 revision. The goal of these changes is to expose all **required** operations and ensure that they're behaviorally compliant with the IEEE spec. For more information about floating-point improvements, see the Floating-Point Parsing and Formatting improvements in .NET Core 3.0 blog post.

Parsing and formatting fixes include:

- Correctly parse and round inputs of any length.
- Correctly parse and format negative zero.
- Correctly parse `Infinity` and `NaN` by doing a case-insensitive check and allowing an optional preceding `+` where applicable.

New System.Math APIs include:

- BitIncrement(Double) and BitDecrement(Double)
  Corresponds to the `nextUp` and `nextDown` IEEE operations. They return the smallest floating-point number that compares greater or lesser than the input (respectively). For example, `Math.BitIncrement(0.0)` would return `double.Epsilon`.

- MaxMagnitude(Double, Double) and MinMagnitude(Double, Double)
  Corresponds to the `maxNumMag` and `minNumMag` IEEE operations, they return the value that is greater or lesser in magnitude of the two inputs (respectively). For example, `Math.MaxMagnitude(2.0, -3.0)` would return `-3.0`.

- ILogB(Double)
  Corresponds to the `logB` IEEE operation that returns an integral value, it returns the integral base-2 log of the input parameter. This method is effectively the same as `floor(log2(x))`, but done with minimal rounding error.

- ScaleB(Double, Int32)
  Corresponds to the `scaleB` IEEE operation that takes an integral value, it returns effectively `x * pow(2, n)`, but is done with minimal rounding error.

- Log2(Double)
  Corresponds to the `log2` IEEE operation, it returns the base-2 logarithm. It minimizes rounding error.

- FusedMultiplyAdd(Double, Double, Double)
  Corresponds to the `fma` IEEE operation, it performs a fused multiply add. That is, it does `(x * y) + z` as a single operation, thereby minimizing the rounding error. An example would be `FusedMultiplyAdd(1e308, 2.0, -1e308)` which returns `1e308`. The regular `(1e308 * 2.0) - 1e308` returns `double.PositiveInfinity`.

- CopySign(Double, Double)
  Corresponds to the `copySign` IEEE operation, it returns the value of `x`, but with the sign of `y`.

Fast built-in JSON support

.NET users have largely relied on **Json.NET** and other popular JSON libraries, which continue to be good choices. **Json.NET** uses .NET strings as its base datatype, which is UTF-16 under the hood.

The new built-in JSON support is high-performance, low allocation, and based on `Span<byte>`. Three new main JSON-related types have been added to .NET Core 3.0 the System.Text.Json namespace. These types don't *yet*

support plain old CLR object (POCO) serialization and deserialization.

Utf8JsonReader

System.Text.Json.Utf8JsonReader is a high-performance, low allocation, forward-only reader for UTF-8 encoded JSON text, read from a `ReadOnlySpan<byte>`. The `Utf8JsonReader` is a foundational, low-level type, that can be used to build custom parsers and deserializers. Reading through a JSON payload using the new `Utf8JsonReader` is 2x faster than using the reader from **Json.NET**. It doesn't allocate until you need to actualize JSON tokens as (UTF-16) strings.

Here is an example of reading through the **launch.json** file created by Visual Studio Code:

```
public static void PrintJson(ReadOnlySpan<byte> dataUtf8)
{
    var json = new Utf8JsonReader(dataUtf8, isFinalBlock: true, state: default);

    while (json.Read())
    {
        JsonTokenType tokenType = json.TokenType;
        ReadOnlySpan<byte> valueSpan = json.ValueSpan;
        switch (tokenType)
        {
            case JsonTokenType.StartObject:
            case JsonTokenType.EndObject:
                break;
            case JsonTokenType.StartArray:
            case JsonTokenType.EndArray:
                break;
            case JsonTokenType.PropertyName:
                break;
            case JsonTokenType.String:
                Console.WriteLine($"STRING: {json.GetString()}");
                break;
            case JsonTokenType.Number:
                if (!json.TryGetInt32(out int valueInteger))
                {
                    throw new FormatException();
                }
                break;
            case JsonTokenType.True:
            case JsonTokenType.False:
                Console.WriteLine($"BOOL: {json.GetBoolean()}");
                break;
            case JsonTokenType.Null:
                break;
            default:
                throw new ArgumentException();
        }
    }

    dataUtf8 = dataUtf8.Slice((int)json.BytesConsumed);
    JsonReaderState state = json.CurrentState;
}
```

```
// Calling code
Console.WriteLine("Read with Utf8JsonReader");
PrintJson(File.ReadAllBytes("launch.json").AsSpan());
```

Utf8JsonWriter

System.Text.Json.Utf8JsonWriter provides a high-performance, non-cached, forward-only way to write UTF-8 encoded JSON text from common .NET types like `String`, `Int32`, and `DateTime`. Like the reader, the writer is a foundational, low-level type, that can be used to build custom serializers. Writing a JSON payload using the new

`Utf8JsonWriter` is 30-80% faster than using the writer from **Json.NET** and doesn't allocate.

JsonDocument

System.Text.Json.JsonDocument is built on top of the `Utf8JsonReader`. The `JsonDocument` provides the ability to parse JSON data and build a read-only Document Object Model (DOM) that can be queried to support random access and enumeration. The JSON elements that compose the data can be accessed via the JsonElement type that is exposed by the `JsonDocument` as a property called `RootElement`. The `JsonElement` contains the JSON array and object enumerators along with APIs to convert JSON text to common .NET types. Parsing a typical JSON payload and accessing all its members using the `JsonDocument` is 2-3x faster than **Json.NET** with little allocations for data that is reasonably sized (that is, < 1 MB).

Here is a sample usage of the `JsonDocument` and `JsonElement` that can be used as a starting point:

```
public static void ReadJson(string jsonString)
{
    using var document = JsonDocument.Parse(jsonString);

    var root = document.RootElement;
    var version = root.GetProperty("version").GetString();
    var configurations = root.GetProperty("configurations");

    Console.WriteLine($"Launch Version: {version}");

    foreach (var config in configurations.EnumerateArray())
    {
        var name = config.GetProperty("name").GetString();
        Console.WriteLine($"Config: {name}");
    }
}
```

Here is a C# 8.0 example of reading through the **launch.json** file created by Visual Studio Code:

```
// Calling code
Console.WriteLine("Read with JsonDocument");
ReadJson(File.ReadAllText("launch.json"));
```

JsonSerializer

System.Text.Json.JsonSerializer is built on top of Utf8JsonReader and Utf8JsonWriter to provide a fast, low-memory serialization option when working with JSON documents and fragments.

Here is an example of serializing an object to JSON:

```
var instance = new
{
    FirstName = "John",
    LastName = "Smith",
    Age = 18
};

System.Console.WriteLine(System.Text.Json.JsonSerializer.Serialize(instance));
```

Here is an example of deserializing a JSON string to an object. You can use the JSON string produced by the previous example:

```
public class JsonPerson
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }

    public static JsonPerson Parse(string json) =>
        System.Text.Json.JsonSerializer.Deserialize<JsonPerson>(json);
}
```

Interop improvements

.NET Core 3.0 improves native API interop.

Type: NativeLibrary

System.Runtime.InteropServices.NativeLibrary provides an encapsulation for loading a native library (using the same load logic as .NET Core P/Invoke) and providing the relevant helper functions such as `getSymbol`. For a code example, see the DLLMap Demo.

Windows Native Interop

Windows offers a rich native API in the form of flat C APIs, COM, and WinRT. While .NET Core supports **P/Invoke**, .NET Core 3.0 adds the ability to **CoCreate COM APIs** and **Activate WinRT APIs**. For a code example, see the Excel Demo.

HTTP/2 support

The System.Net.Http.HttpClient type supports the HTTP/2 protocol. If HTTP/2 is enabled, the HTTP protocol version is negotiated via TLS/ALPN, and HTTP/2 is used if the server elects to use it.

The default protocol remains HTTP/1.1, but HTTP/2 can be enabled in two different ways. First, you can set the HTTP request message to use HTTP/2:

```
var client = new HttpClient() { BaseAddress = new Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version = new Version(2, 0) })
using (var response = await client.SendAsync(request))
    Console.WriteLine(response.Content);
```

Second, you can change HttpClient to use HTTP/2 by default:

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

Many times when you're developing an application, you want to use an unencrypted connection. If you know the target endpoint will be using HTTP/2, you can turn on unencrypted connections for HTTP/2. You can turn it on by

setting the `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT` environment variable to `1` or by enabling it in the app context:

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

TLS 1.3 & OpenSSL 1.1.1 on Linux

.NET Core now takes advantage of TLS 1.3 support in OpenSSL 1.1.1, when it's available in a given environment. With TLS 1.3:

- Connection times are improved with reduced round trips required between the client and server.
- Improved security because of the removal of various obsolete and insecure cryptographic algorithms.

When available, .NET Core 3.0 uses **OpenSSL 1.1.1**, **OpenSSL 1.1.0**, or **OpenSSL 1.0.2** on a Linux system. When **OpenSSL 1.1.1** is available, both System.Net.Security.SslStream and System.Net.Http.HttpClient types will use **TLS 1.3** (assuming both the client and server support **TLS 1.3**).

> **IMPORTANT**
> Windows and macOS do not yet support **TLS 1.3**. .NET Core 3.0 will support **TLS 1.3** on these operating systems when support becomes available.

The following C# 8.0 example demonstrates .NET Core 3.0 on Ubuntu 18.10 connecting to https://www.cloudflare.com:

```
using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost} with {sslStream.SslProtocol}");
        }
    }
}
```

Cryptography ciphers

.NET 3.0 adds support for **AES-GCM** and **AES-CCM** ciphers, implemented with System.Security.Cryptography.AesGcm and System.Security.Cryptography.AesCcm respectively. These algorithms are both Authenticated Encryption with Association Data (AEAD) algorithms.

The following code demonstrates using `AesGcm` cipher to encrypt and decrypt random data.

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {
            // key should be: pre-known, derived, or transported via another channel, such as RSA encryption
            byte[] key = new byte[16];
            RandomNumberGenerator.Fill(key);

            byte[] nonce = new byte[12];
            RandomNumberGenerator.Fill(nonce);

            // normally this would be your data
            byte[] dataToEncrypt = new byte[1234];
            byte[] associatedData = new byte[333];
            RandomNumberGenerator.Fill(dataToEncrypt);
            RandomNumberGenerator.Fill(associatedData);

            // these will be filled during the encryption
            byte[] tag = new byte[16];
            byte[] ciphertext = new byte[dataToEncrypt.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag, associatedData);
            }

            // tag, nonce, ciphertext, associatedData should be sent to the other part

            byte[] decryptedData = new byte[ciphertext.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData, associatedData);
            }

            // do something with the data
            // this should always print that data is the same
            Console.WriteLine($"AES-GCM: Decrypted data is {(dataToEncrypt.SequenceEqual(decryptedData) ? "the
same as" : "different than")} original data.");
        }
    }
}
```

Cryptographic Key Import/Export

.NET Core 3.0 supports the import and export of asymmetric public and private keys from standard formats. You don't need to use an X.509 certificate.

All key types, such as *RSA*, *DSA*, *ECDsa*, and *ECDiffieHellman*, support the following formats:

- **Public Key**
  - X.509 SubjectPublicKeyInfo
- **Private key**
  - PKCS#8 PrivateKeyInfo
  - PKCS#8 EncryptedPrivateKeyInfo

RSA keys also support:

- **Public Key**
  - PKCS#1 RSAPublicKey
- **Private key**
  - PKCS#1 RSAPrivateKey

The export methods produce DER-encoded binary data, and the import methods expect the same. If a key is stored in the text-friendly PEM format, the caller will need to base64-decode the content before calling an import method.

```
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length - bytesRead} extra byte(s) in file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

PKCS#8 files can be inspected with System.Security.Cryptography.Pkcs.Pkcs8PrivateKeyInfo and **PFX/PKCS#12** files can be inspected with System.Security.Cryptography.Pkcs.Pkcs12Info. **PFX/PKCS#12** files can be manipulated with System.Security.Cryptography.Pkcs.Pkcs12Builder.

SerialPort for Linux

.NET Core 3.0 provides basic support for System.IO.Ports.SerialPort on Linux.

Previously, .NET Core only supported using `SerialPort` on Windows.

For more information about the limited support for the serial port on Linux, see GitHub issue #33146.

Docker and cgroup memory Limits

Starting with Preview 3, running .NET Core 3.0 on Linux with Docker works better with cgroup memory limits. Running a Docker container with memory limits, such as with `docker run -m`, changes how .NET Core behaves.

- Default Garbage Collector (GC) heap size: maximum of 20 mb or 75% of the memory limit on the container.
- Explicit size can be set as an absolute number or percentage of cgroup limit.
- Minimum reserved segment size per GC heap is 16 mb. This size reduces the number of heaps that are created on machines.

Smaller Garbage Collection heap sizes

The Garbage Collector's default heap size has been reduced resulting in .NET Core using less memory. This change better aligns with the generation 0 allocation budget with modern processor cache sizes.

Garbage Collection Large Page support

Large Pages (also known as Huge Pages on Linux) is a feature where the operating system is able to establish memory regions larger than the native page size (often 4K) to improve performance of the application requesting these large pages.

The Garbage Collector can now be configured with the **GCLargePages** setting as an opt-in feature to choose to allocate large pages on Windows.

GPIO Support for Raspberry Pi

Two packages have been released to NuGet that you can use for GPIO programming:

- System.Device.Gpio
- Iot.Device.Bindings

The GPIO packages include APIs for *GPIO*, *SPI*, *I2C*, and *PWM* devices. The IoT bindings package includes device bindings. For more information, see the devices GitHub repo.

ARM64 Linux support

.NET Core 3.0 adds support for ARM64 for Linux. The primary use case for ARM64 is currently with IoT scenarios. For more information, see .NET Core ARM64 Status.

Docker images for .NET Core on ARM64 are available for Alpine, Debian, and Ubuntu.

> **NOTE**
>
> **ARM64** Windows support isn't yet available.

What's new in .NET Core 2.2

3/8/2019 • 4 minutes to read • Edit Online

.NET Core 2.2 includes enhancements in application deployment, event handling for runtime services, authentication to Azure SQL databases, JIT compiler performance, and code injection prior to the execution of the `Main` method.

New deployment mode

Starting with .NET Core 2.2, you can deploy framework-dependent executables, which are **.exe** files instead of **.dll** files. Functionally similar to framework-dependent deployments, framework-dependent executables (FDE) still rely on the presence of a shared system-wide version of .NET Core to run. Your app contains only your code and any third-party dependencies. Unlike framework-dependent deployments, FDEs are platform-specific.

This new deployment mode has the distinct advantage of building an executable instead of a library, which means you can run your app directly without invoking `dotnet` first.

Core

Handling events in runtime services

You may often want to monitor your application's use of runtime services, such as the GC, JIT, and ThreadPool, to understand how they impact your application. On Windows systems, this is commonly done by monitoring the ETW events of the current process. While this continues to work well, it's not always possible to use ETW if you're running in a low-privilege environment or on Linux or macOS.

Starting with .NET Core 2.2, CoreCLR events can now be consumed using the System.Diagnostics.Tracing.EventListener class. These events describe the behavior of such runtime services as GC, JIT, ThreadPool, and interop. These are the same events that are exposed as part of the CoreCLR ETW provider. This allows for applications to consume these events or use a transport mechanism to send them to a telemetry aggregation service. You can see how to subscribe to events in the following code sample:

```
internal sealed class SimpleEventListener : EventListener
{
    // Called whenever an EventSource is created.
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        // Watch for the .NET runtime EventSource and enable all of its events.
        if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
        {
            EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)(-1));
        }
    }

    // Called whenever an event is written.
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Write the contents of the event to the console.
        Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID = {eventData.EventId} Name =
{eventData.EventName}");
        for (int i = 0; i < eventData.Payload.Count; i++)
        {
            string payloadString = eventData.Payload[i]?.ToString() ?? string.Empty;
            Console.WriteLine($"\tName = \"{eventData.PayloadNames[i]}\" Value = \"{payloadString}\"");
        }
        Console.WriteLine("\n");
    }
}
```

In addition, .NET Core 2.2 adds the following two properties to the EventWrittenEventArgs class to provide additional information about ETW events:

- EventWrittenEventArgs.OSThreadId
- EventWrittenEventArgs.TimeStamp

Data

AAD authentication to Azure SQL databases with the SqlConnection.AccessToken property

Starting with .NET Core 2.2, an access token issued by Azure Active Directory can be used to authenticate to an Azure SQL database. To support access tokens, the AccessToken property has been added to the SqlConnection class. To take advantage of AAD authentication, download version 4.6 of the System.Data.SqlClient NuGet package. In order to use the feature, you can obtain the access token value using the Active Directory Authentication Library for .NET contained in the `Microsoft.IdentityModel.Clients.ActiveDirectory` NuGet package.

JIT compiler improvements

Tiered compilation remains an opt-in feature

In .NET Core 2.1, the JIT compiler implemented a new compiler technology, *tiered compilation*, as an opt-in feature. The goal of tiered compilation is improved performance. One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends executing unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second tier of compilation is performed in parallel for enhanced performance.

For information on the performance improvement that can result from tiered compilation, see Announcing .NET

Core 2.2 Preview 2.

In .NET Core 2.2 Preview 2, tiered compilation was enabled by default. However, we've decided that we are still not ready to enable tiered compilation by default. So in .NET Core 2.2, tiered compilation continues to be an opt-in feature. For information on opting in to tiered compilation, see Jit compiler improvements in What's new in .NET Core 2.1.

Runtime

Injecting code prior to executing the Main method

Starting with .NET Core 2.2, you can use a startup hook to inject code prior to running an application's Main method. Startup hooks make it possible for a host to customize the behavior of applications after they have been deployed without needing to recompile or change the application.

We expect hosting providers to define custom configuration and policy, including settings that potentially influence the load behavior of the main entry point, such as the System.Runtime.Loader.AssemblyLoadContext behavior. The hook can be used to set up tracing or telemetry injection, to set up callbacks for handling, or to define other environment-dependent behavior. The hook is separate from the entry point, so that user code doesn't need to be modified.

See Host startup hook for more information.

See also

- What's new in .NET Core
- What's new in ASP.NET Core 2.2
- New features in EF Core 2.2

What's new in .NET Core 2.1

7/19/2019 • 9 minutes to read • Edit Online

.NET Core 2.1 includes enhancements and new features in the following areas:

- Tooling
- Roll forward
- Deployment
- Windows Compatibility Pack
- JIT compilation improvements
- API changes

Tooling

The .NET Core 2.1 SDK (v 2.1.300), the tooling included with .NET Core 2.1, includes the following changes and enhancements:

Build performance improvements

A major focus of .NET Core 2.1 is improving build-time performance, particularly for incremental builds. These performance improvements apply to both command-line builds using `dotnet build` and to builds in Visual Studio. Some individual areas of improvement include:

- For package asset resolution, resolving only assets used by a build rather than all assets.
- Caching of assembly references.
- Use of long-running SDK build servers, which are processes that span across individual `dotnet build` invocations. They eliminate the need to JIT-compile large blocks of code every time `dotnet build` is run. Build server processes can be automatically terminated with the following command:

  ```
  dotnet buildserver shutdown
  ```

New CLI commands

A number of tools that were available only on a per project basis using `DotnetCliToolReference` are now available as part of the .NET Core SDK. These tools include:

- `dotnet watch` provides a file system watcher that waits for a file to change before executing a designated set of commands. For example, the following command automatically rebuilds the current project and generates verbose output whenever a file in it changes:

  ```
  dotnet watch -- --verbose build
  ```

  Note the `--` option that precedes the `--verbose` option. It delimits the options passed directly to the `dotnet watch` command from the arguments that are passed to the child `dotnet` process. Without it, the `--verbose` option applies to the `dotnet watch` command, not the `dotnet build` command.

  For more information, see Develop ASP.NET Core apps using dotnet watch

- `dotnet dev-certs` generates and manages certificates used during development in ASP.NET Core applications.

- `dotnet user-secrets` manages the secrets in a user secret store in ASP.NET Core applications.
- `dotnet sql-cache` creates a table and indexes in a Microsoft SQL Server database to be used for distributed caching.
- `dotnet ef` is a tool for managing databases, DbContext objects, and migrations in Entity Framework Core applications. For more information, see EF Core .NET Command-line Tools.

Global Tools

.NET Core 2.1 supports *Global Tools* -- that is, custom tools that are available globally from the command line. The extensibility model in previous versions of .NET Core made custom tools available on a per project basis only by using `DotnetCliToolReference`.

To install a Global Tool, you use the dotnet tool install command. For example:

```
dotnet tool install -g dotnetsay
```

Once installed, the tool can be run from the command line by specifying the tool name. For more information, see .NET Core Global Tools overview.

Tool management with the `dotnet tool` command

In .NET Core 2.1 SDK, all tools operations use the `dotnet tool` command. The following options are available:

- `dotnet tool install` to install a tool.
- `dotnet tool update` to uninstall and reinstall a tool, which effectively updates it.
- `dotnet tool list` to list currently installed tools.
- `dotnet tool uninstall` to uninstall currently installed tools.

Roll forward

All .NET Core applications starting with .NET Core 2.0 automatically roll forward to the latest *minor version* installed on a system.

Starting with .NET Core 2.0, if the version of .NET Core that an application was built with is not present at runtime, the application automatically runs against the latest installed *minor version* of .NET Core. In other words, if an application is built with .NET Core 2.0, and .NET Core 2.0 is not present on the host system but .NET Core 2.1 is, the application runs with .NET Core 2.1.

> **IMPORTANT**
> This roll-forward behavior doesn't apply to preview releases. By default, it also doesn't apply to major releases, but this can be changed with the settings below.

You can modify this behavior by changing the setting for the roll-forward on no candidate shared framework. The available settings are:

- `0` - disable minor version roll-forward behavior. With this setting, an application built for .NET Core 2.0.0 will roll forward to .NET Core 2.0.1, but not to .NET Core 2.2.0 or .NET Core 3.0.0.
- `1` - enable minor version roll-forward behavior. This is the default value for the setting. With this setting, an application built for .NET Core 2.0.0 will roll forward to either .NET Core 2.0.1 or .NET Core 2.2.0, depending on which one is installed, but it will not roll forward to .NET Core 3.0.0.
- `2` - enable minor and major version roll-forward behavior. If set, even different major versions are considered,

so an application built for .NET Core 2.0.0 will roll forward to .NET Core 3.0.0.

You can modify this setting in any of three ways:

- Set the `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` environment variable to the desired value.
- Add the following line with the desired value to the *.runtimeconfig.json* file:

```
"rollForwardOnNoCandidateFx" : 0
```

- When using .NET Core CLI tools, add the following option with the desired value to a .NET Core command such as `run`:

```
dotnet run --rollForwardOnNoCandidateFx=0
```

Patch version roll forward is independent of this setting and is done after any potential minor or major version roll forward is applied.

Deployment

Self-contained application servicing

`dotnet publish` now publishes self-contained applications with a serviced runtime version. When you publish a self-contained application with the .NET Core 2.1 SDK (v 2.1.300), your application includes the latest serviced runtime version known by that SDK. When you upgrade to the latest SDK, you'll publish with the latest .NET Core runtime version. This applies for .NET Core 1.0 runtimes and later.

Self-contained publishing relies on runtime versions on NuGet.org. You do not need to have the serviced runtime on your machine.

Using the .NET Core 2.0 SDK, self-contained applications are published with the .NET Core 2.0.0 runtime unless a different version is specified via the `RuntimeFrameworkVersion` property. With this new behavior, you'll no longer need to set this property to select a higher runtime version for a self-contained application. The easiest approach going forward is to always publish with .NET Core 2.1 SDK (v 2.1.300).

For more information, see Self-contained deployment runtime roll forward.

Windows Compatibility Pack

When you port existing code from the .NET Framework to .NET Core, you can use the Windows Compatibility Pack. It provides access to 20,000 more APIs than are available in .NET Core. These APIs include types in the System.Drawing namespace, the EventLog class, WMI, Performance Counters, Windows Services, and the Windows registry types and members.

JIT compiler improvements

.NET Core incorporates a new JIT compiler technology called *tiered compilation* (also known as *adaptive optimization*) that can significantly improve performance. Tiered compilation is an opt-in setting.

One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends running unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second

tier of compilation is performed in parallel for enhanced performance.

You can opt into tiered compilation in either of two ways.

- To use tiered compilation in all projects that use the .NET Core 2.1 SDK, set the following environment variable:

```
COMPlus_TieredCompilation="1"
```

- To use tiered compilation on a per-project basis, add the `<TieredCompilation>` property to the `<PropertyGroup>` section of the MSBuild project file, as the following example shows:

```
<PropertyGroup>
    <!-- other property definitions -->

    <TieredCompilation>true</TieredCompilation>
</PropertyGroup>
```

API changes

`Span<T>` and `Memory<T>`

.NET Core 2.1 includes some new types that make working with arrays and other types of memory much more efficient. The new types include:

- System.Span<T> and System.ReadOnlySpan<T>.
- System.Memory<T> and System.ReadOnlyMemory<T>.

Without these types, when passing such items as a portion of an array or a section of a memory buffer, you have to make a copy of some portion of the data before passing it to a method. These types provide a virtual view of that data that eliminates the need for the additional memory allocation and copy operations.

The following example uses a Span<T> and Memory<T> instance to provide a virtual view of 10 elements of an array.

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
        foreach (var value in part)
            Console.Write($"{value}  ");
    }
}
// The example displays the following output:
//     20  22  24  26  28  30  32  34  36  38
```

```
Module Program
    Sub Main()
        Dim numbers As Integer() = New Integer(99) {}

        For i As Integer = 0 To 99
            numbers(i) = i * 2
        Next

        Dim part = New Memory(Of Integer)(numbers, start:=10, length:=10)

        For Each value In part.Span
            Console.Write($"{value}  ")
        Next
    End Sub
End Module
' The example displays the following output:
'     20  22  24  26  28  30  32  34  36  38
```

Brotli compression

.NET Core 2.1 adds support for Brotli compression and decompression. Brotli is a general-purpose lossless compression algorithm that is defined in RFC 7932 and is supported by most web browsers and major web servers. You can use the stream-based System.IO.Compression.BrotliStream class or the high-performance span-based System.IO.Compression.BrotliEncoder and System.IO.Compression.BrotliDecoder classes. The following example illustrates compression with the BrotliStream class:

```
public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress, CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}
```

The BrotliStream behavior is the same as DeflateStream and GZipStream, which makes it easy to convert code that calls these APIs to BrotliStream.

New cryptography APIs and cryptography improvements

.NET Core 2.1 includes numerous enhancements to the cryptography APIs:

- System.Security.Cryptography.Pkcs.SignedCms is available in the System.Security.Cryptography.Pkcs package. The implementation is the same as the SignedCms class in the .NET Framework.
- New overloads of the X509Certificate.GetCertHash and X509Certificate.GetCertHashString methods accept a hash algorithm identifier to enable callers to get certificate thumbprint values using algorithms other than SHA-1.
- New Span<T>-based cryptography APIs are available for hashing, HMAC, cryptographic random number generation, asymmetric signature generation, asymmetric signature processing, and RSA encryption.
- The performance of System.Security.Cryptography.Rfc2898DeriveBytes has improved by about 15% by using a Span<T>-based implementation.
- The new System.Security.Cryptography.CryptographicOperations class includes two new methods:
    - FixedTimeEquals takes a fixed amount of time to return for any two inputs of the same length, which makes it suitable for use in cryptographic verification to avoid contributing to timing side-channel

information.

  - ZeroMemory is a memory-clearing routine that cannot be optimized.

- The static RandomNumberGenerator.Fill method fills a Span<T> with random values.
- The System.Security.Cryptography.Pkcs.EnvelopedCms is now supported on Linux and maxOS.
- Elliptic-Curve Diffie-Hellman (ECDH) is now available in the System.Security.Cryptography.ECDiffieHellman class family. The surface area is the same as in the .NET Framework.
- The instance returned by RSA.Create can encrypt or decrypt with OAEP using a SHA-2 digest, as well as generate or validate signatures using RSA-PSS.

Sockets improvements

.NET Core includes a new type, System.Net.Http.SocketsHttpHandler, and a rewritten System.Net.Http.HttpMessageHandler, that form the basis of higher-level networking APIs. System.Net.Http.SocketsHttpHandler, for example, is the basis of the HttpClient implementation. In previous versions of .NET Core, higher-level APIs were based on native networking implementations.

The sockets implementation introduced in .NET Core 2.1 has a number of advantages:

- A significant performance improvement when compared with the previous implementation.
- Elimination of platform dependencies, which simplifies deployment and servicing.
- Consistent behavior across all .NET Core platforms.

SocketsHttpHandler is the default implementation in .NET Core 2.1. However, you can configure your application to use the older HttpClientHandler class by calling the AppContext.SetSwitch method:

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", False)
```

You can also use an environment variable to opt out of using sockets implementations based on SocketsHttpHandler. To do this, set the `DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` to either `false` or 0.

On Windows, you can also choose to use System.Net.Http.WinHttpHandler, which relies on a native implementation, or the SocketsHttpHandler class by passing an instance of the class to the HttpClient constructor.

On Linux and macOS, you can only configure HttpClient on a per-process basis. On Linux, you need to deploy libcurl if you want to use the old HttpClient implementation. (It is installed with .NET Core 2.0.)

See also

- What's new in .NET Core
- New features in EF Core 2.1
- What's new in ASP.NET Core 2.1

What's new in .NET Core 2.0

7/30/2019 • 6 minutes to read • Edit Online

.NET Core 2.0 includes enhancements and new features in the following areas:

- Tooling
- Language support
- Platform improvements
- API changes
- Visual Studio integration
- Documentation improvements

Tooling

dotnet restore runs implicitly

In previous versions of .NET Core, you had to run the dotnet restore command to download dependencies immediately after you created a new project with the dotnet new command, as well as whenever you added a new dependency to your project.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

You can also disable the automatic invocation of `dotnet restore` by passing the `--no-restore` switch to the `new`, `run`, `build`, `publish`, `pack`, and `test` commands.

Retargeting to .NET Core 2.0

If the .NET Core 2.0 SDK is installed, projects that target .NET Core 1.x can be retargeted to .NET Core 2.0.

To retarget to .NET Core 2.0, edit your project file by changing the value of the `<TargetFramework>` element (or the `<TargetFrameworks>` element if you have more than one target in your project file) from 1.x to 2.0:

```
<PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
 </PropertyGroup>
```

You can also retarget .NET Standard libraries to .NET Standard 2.0 the same way:

```
<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
 </PropertyGroup>
```

For more information about migrating your project to .NET Core 2.0, see Migrating from ASP.NET Core 1.x to ASP.NET Core 2.0.

Language support

In addition to supporting C# and F#, .NET Core 2.0 also supports Visual Basic.

Visual Basic

With version 2.0, .NET Core now supports Visual Basic 2017. You can use Visual Basic to create the following project types:

- .NET Core console apps
- .NET Core class libraries
- .NET Standard class libraries
- .NET Core unit test projects
- .NET Core xUnit test projects

For example, to create a Visual Basic "Hello World" application, do the following steps from the command line:

1. Open a console window, create a directory for your project, and make it the current directory.
2. Enter the command `dotnet new console -lang vb`.

   The command creates a project file with a `.vbproj` file extension, along with a Visual Basic source code file named *Program.vb*. This file contains the source code to write the string "Hello World!" to the console window.
3. Enter the command `dotnet run`. The .NET Core CLI automatically compiles and executes the application, which displays the message "Hello World!" in the console window.

Support for C# 7.1

.NET Core 2.0 supports C# 7.1, which adds a number of new features, including:

- The `Main` method, the application entry point, can be marked with the async keyword.
- Inferred tuple names.
- Default expressions.

Platform improvements

.NET Core 2.0 includes a number of features that make it easier to install .NET Core and to use it on supported operating systems.

.NET Core for Linux is a single implementation

.NET Core 2.0 offers a single Linux implementation that works on multiple Linux distributions. .NET Core 1.x required that you download a distribution-specific Linux implementation.

You can also develop apps that target Linux as a single operating system. .NET Core 1.x required that you target each Linux distribution separately.

Support for the Apple cryptographic libraries

.NET Core 1.x on macOS required the OpenSSL toolkit's cryptographic library. .NET Core 2.0 uses the Apple cryptographic libraries and doesn't require OpenSSL, so you no longer need to install it.

API changes and library support

Support for .NET Standard 2.0

The .NET Standard defines a versioned set of APIs that must be available on .NET implementations that comply with that version of the standard. The .NET Standard is targeted at library developers. It aims to guarantee the functionality that is available to a library that targets a version of the .NET Standard on each .NET implementation.

.NET Core 1.x supports the .NET Standard version 1.6; .NET Core 2.0 supports the latest version, .NET Standard 2.0. For more information, see .NET Standard.

.NET Standard 2.0 includes over 20,000 more APIs than were available in the .NET Standard 1.6. Much of this expanded surface area results from incorporating APIs that are common to the .NET Framework and Xamarin into .NET Standard.

.NET Standard 2.0 class libraries can also reference .NET Framework class libraries, provided that they call APIs that are present in the .NET Standard 2.0. No recompilation of the .NET Framework libraries is required.

For a list of the APIs that have been added to the .NET Standard since its last version, the .NET Standard 1.6, see .NET Standard 2.0 vs. 1.6.

Expanded surface area

The total number of APIs available on .NET Core 2.0 has more than doubled in comparison with .NET Core 1.1.

And with the Windows Compatibility Pack porting from .NET Framework has also become much simpler.

Support for .NET Framework libraries

.NET Core code can reference existing .NET Framework libraries, including existing NuGet packages. Note that the libraries must use APIs that are found in .NET Standard.

Visual Studio integration

Visual Studio 2017 version 15.3 and in some cases Visual Studio for Mac offer a number of significant enhancements for .NET Core developers.

Retargeting .NET Core apps and .NET Standard libraries

If the .NET Core 2.0 SDK is installed, you can retarget .NET Core 1.x projects to .NET Core 2.0 and .NET Standard 1.x libraries to .NET Standard 2.0.

To retarget your project in Visual Studio, you open the **Application** tab of the project's properties dialog and change the **Target framework** value to **.NET Core 2.0** or **.NET Standard 2.0**. You can also change it by right-clicking on the project and selecting the **Edit *.csproj file** option. For more information, see the Tooling section earlier in this topic.

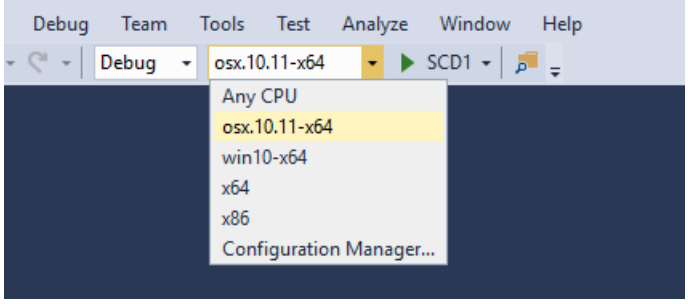
Live Unit Testing support for .NET Core

Whenever you modify your code, Live Unit Testing automatically runs any affected unit tests in the background and displays the results and code coverage live in the Visual Studio environment. .NET Core 2.0 now supports Live Unit Testing. Previously, Live Unit Testing was available only for .NET Framework applications.

For more information, see Live Unit Testing with Visual Studio 2017 and the Live Unit Testing FAQ.

Better support for multiple target frameworks

If you're building a project for multiple target frameworks, you can now select the target platform from the top-level menu. In the following figure, a project named SCD1 targets 64-bit macOS X 10.11 (`osx.10.11-x64`) and 64-bit Windows 10/Windows Server 2016 (`win10-x64`). You can select the target framework before selecting the project button, in this case to run a debug build.



Side-by-side support for .NET Core SDKs

You can now install the .NET Core SDK independently of Visual Studio. This makes it possible for a single version of Visual Studio to build projects that target different versions of .NET Core. Previously, Visual Studio and the .NET Core SDK were tightly coupled; a particular version of the SDK accompanied a particular version of Visual Studio.

Documentation improvements

.NET Application Architecture

.NET Application Architecture gives you access to a set of e-books that provide guidance, best practices, and sample applications when using .NET to build:

- Microservices and Docker containers
- Web applications with ASP.NET
- Mobile applications with Xamarin
- Applications that are deployed to the Cloud with Azure

See also

- What's new in ASP.NET Core 2.0

Evaluate breaking changes in .NET Core

7/10/2019 • 13 minutes to read • Edit Online

Throughout its history, .NET has attempted to maintain a high level of compatibility from version to version and across flavors of .NET. This continues to be true for .NET Core. Although .NET Core can be considered as a new technology that is independent of the .NET Framework, two major factors limit the ability of .NET Core to diverge from .NET Framework:

- A large number of developers either originally developed or continue to develop .NET Framework applications. They expect consistent behavior across .NET implementations.
- .NET Standard library projects allow developers to create libraries that target common APIs shared by .NET Core and .NET Framework. Developers expect that a library used in a .NET Core application should behave identically to the same library used in a .NET Framework application.

Along with compatibility across .NET implementations, developers expect a high level of compatibility across .NET Core versions. In particular, code written for an earlier version of .NET Core should run seamlessly on a later version of .NET Core. In fact, many developers expect that the new APIs found in newly released versions of .NET Core should also be compatible with the pre-release versions in which those APIs were introduced.

This article outlines the categories of compatibility changes (or breaking changes) and the way in which the .NET team evaluates changes in each of these categories. Understanding how the .NET team approaches possible breaking changes is particularly helpful for developers who are opening pull requests in the dotnet/corefx GitHub repository that modify the behavior of existing APIs.

> **NOTE**
> For a definition of compatibility categories, such as binary compatibility and backward compatibility, see Breaking change categories.

The following sections describes the categories of changes made to .NET Core APIs and their impact on application compatibility. The ✔ icon indicates that a particular kind of change is allowed, indicates that it is disallowed, and indicates a change that may or may not be allowed. Changes in this last category require judgment and an evaluation of how predictable, obvious, and consistent the previous behavior was.

> **NOTE**
> In addition to serving as a guide to how changes to .NET Core libraries are evaluated, library developers can also use these criteria to evaluate changes to their libraries that target multiple .NET implementations and versions.

Modifications to the public contract

Changes in this category *modify* the public surface area of a type. Most of the changes in this category are disallowed since they violate *backwards compatibility* (the ability of an application that was developed with a previous version of an API to execute without recompilation on a later version).

Types

- ✔ **Removing an interface implementation from a type when the interface is already implemented by a base type**

- ❌ **Adding a new interface implementation to a type**

  This is an acceptable change because it does not adversely affect existing clients. Any changes to the type must work within the boundaries of acceptable changes defined here for the new implementation to remain acceptable. Extreme caution is necessary when adding interfaces that directly affect the ability of a designer or serializer to generate code or data that cannot be consumed down-level. An example is the ISerializable interface.

- ❌ **Introducing a new base class**

  A type can be introduced into an hierarchy between two existing types if it doesn't introduce any new abstract members or change the semantics or behavior of existing types. For example, in .NET Framework 2.0, the DbConnection class became a new base class for SqlConnection, which had previously derived directly from Component.

- ✔️❌ **Moving a type from one assembly to another**

  Note that the *old* assembly must be marked with the TypeForwardedToAttribute that points to the new assembly.

- ✔️❌ **Changing a struct type to a `readonly struct` type**

  Note that changing a `readonly struct` type to a `struct` type is not allowed.

- ✔️❌ **Adding the sealed or abstract keyword to a type when there are no *accessible* (public or protected) constructors**

- ✔️❌ **Expanding the visibility of a type**

- ❌ **Changing the namespace or name of a type**

- ❌ **Renaming or removing a public type**

  This breaks all code that uses the renamed or removed type.

- ❌ **Changing the underlying type of an enumeration**

  This is a compile-time and behavioral breaking change as well as a binary breaking change that can make attribute arguments unparsable.

- ❌ **Sealing a type that was previously unsealed**

- ❌ **Adding an interface to the set of base types of an interface**

  If an interface implements an interface that it previously did not implement, all types that implemented the original version of the interface are broken.

- ❌ **Removing a class from the set of base classes or an interface from the set of implemented interfaces**

  There is one exception to the rule for interface removal: you can add the implementation of an interface that derives from the removed interface. For example, you can remove IDisposable if the type or interface now implements IComponent, which implements IDisposable.

- ❌ **Changing a `readonly struct` type to a struct type**

  Note that the change of a `struct` type to a `readonly struct` type is allowed.

- ❌ **Changing a struct type to a `ref struct` type, and vice versa**

- ❌ **Reducing the visibility of a type**

However, increasing the visibility of a type is allowed.

Members

- ✔ **Expanding the visibility of a member that is not virtual**
- ✔ **Adding an abstract member to a public type that has no *accessible* (public or protected) constructors, or the type is sealed**

  However, adding an abstract member to a type that has accessible (public or protected) constructors and is not `sealed` is not allowed.
- ✔ **Restricting the visibility of a protected member when the type has no accessible (public or protected) constructors, or the type is sealed**
- ✔ **Moving a member into a class higher in the hierarchy than the type from which it was removed**
- ✔ **Adding or removing an override**

  Note that introducing an override might cause previous consumers to skip over the override when calling base.
- ✔ **Adding a constructor to a class, along with a default (parameterless) constructor if the class previously had no constructors**

  However, adding a constructor to a class that previously had no constructors *without* adding the parameterless constructor is not allowed.
- ✔ **Changing a member from abstract to virtual**
- ✔ **Changing from a `ref readonly` to a `ref` return value (except for virtual methods or interfaces)**
- ✔ **Removing readonly from a field, unless the static type of the field is a mutable value type**
- ✔ **Calling a new event that wasn't previously defined**
- **Adding a new instance field to a type**

  This change impacts serialization.
- **Renaming or removing a public member or parameter**

  This breaks all code that uses the renamed or removed member, or parameter.

  Note that this includes removing or renaming a getter or setter from a property, as well as renaming or removing enumeration members.
- **Adding a member to an interface**
- **Changing the value of a public constant or enumeration member**
- **Changing the type of a property, field, parameter, or return value**
- **Adding, removing, or changing the order of parameters**
- **Adding or removing the in, out , or ref keyword from a parameter**
- **Renaming a parameter (including changing its case)**

  This is considered breaking for two reasons:

  - It breaks late-bound scenarios such as the late binding feature in Visual Basic and dynamic in C#.

- It breaks source compatibility when developers use named arguments.

- ✔️ **Changing from a `ref` return value to a `ref readonly` return value**
- ✔️✔️ **Changing from a `ref readonly` to a `ref` return value on a virtual method or interface**
- ✔️ **Adding or removing abstract from a member**
- ✔️ **Removing the virtual keyword from a member**

  While this often is not a breaking change because the C# compiler tends to emit callvirt Intermediate Language (IL) instructions to call non-virtual methods (`callvirt` performs a null check, while a normal call doesn't), this behavior is not invariable for several reasons:

  - C# is not the only language that .NET targets.
  - The C# compiler increasingly tries to optimize `callvirt` to a normal call whenever the target method is non-virtual and is probably not null (such as a method accessed through the ?. null propagation operator).

  Making a method virtual means that the consumer code would often end up calling it non-virtually.

- ✔️ **Adding the virtual keyword to a member**
- ✔️ **Making a virtual member abstract**

  A virtual member provides a method implementation that *can be* overridden by a derived class. An abstract member provides no implementation and *must be* overridden.

- ✔️ **Adding an abstract member to a public type that has accessible (public or protected) constructors and that is not sealed**
- ✔️ **Adding or removing the static keyword from a member**
- ✔️ **Adding an overload that precludes an existing overload and defines a different behavior**

  This breaks existing clients that were bound to the previous overload. For example, if a class has a single version of a method that accepts a UInt32, an existing consumer will successfully bind to that overload when passing a Int32 value. However, if you add an overload that accepts an Int32, when recompiling or using late-binding, the compiler now binds to the new overload. If different behavior results, this is a breaking change.

- ✔️ **Adding a constructor to a class that previously had no constructor without adding the parameterless constructor**
- ✔️✔️ **Adding readonly to a field**
- ✔️ **Reducing the visibility of a member**

  This includes reducing the visibility of a protected member when there are *accessible* (public or protected) constructors and the type is *not* sealed. If this is not the case, reducing the visibility of a protected member is allowed.

  Note that increasing the visibility of a member is allowed.

- ✔️ **Changing the type of a member**

  The return value of a method or the type of a property or field cannot be modified. For example, the signature of a method that returns an Object cannot be changed to return a String, or vice versa.

- ✔️ **Adding a field to a struct that previously had no state**

Definite assignment rules allow the use of uninitialized variables so long as the variable type is a stateless struct. If the struct is made stateful, code could end up with uninitialized data. This is both potentially a source breaking and a binary breaking change.

- **Firing an existing event when it was never fired before**

Behavioral changes

Assemblies

- **✔ Making an assembly portable when the same platforms are still supported**
- **Changing the name of an assembly**
- **Changing the public key of an assembly**

Properties, fields, parameters, and return values

- **✔ Changing the value of a property, field, return value, or out parameter to a more derived type**

  For example, a method that returns a type of Object can return a String instance. (However, the method signature cannot change.)

- **✔ Increasing the range of accepted values for a property or parameter if the member is not virtual**

  Note that while the range of values that can be passed to the method or are returned by the member can expand, the parameter or member type cannot. For example, while the values passed to a method can expand from 0-124 to 0-255, the parameter type cannot change from Byte to Int32.

- **Increasing the range of accepted values for a property or parameter if the member is virtual**

  This change breaks existing overridden members, which will not function correctly for the extended range of values.

- **Decreasing the range of accepted values for a property or parameter**
- **Increasing the range of returned values for a property, field, return value, or out parameter**
- **Changing the returned values for a property, field, method return value, or out parameter**
- **Changing the default value of a property, field, or parameter**
- **Changing the precision of a numeric return value**
- **A change in the parsing of input and throwing new exceptions (even if parsing behavior is not specified in the documentation**

Exceptions

- **✔ Throwing a more derived exception than an existing exception**

  Because the new exception is a subclass of an existing exception, previous exception handling code continues to handle the exception. For example, in .NET Framework 4, culture creation and retrieval methods began to throw a CultureNotFoundException instead of an ArgumentException if the culture could not be found. Because CultureNotFoundException derives from ArgumentException, this is an acceptable change.

- **✔ Throwing a more specific exception than NotSupportedException, NotImplementedException, NullReferenceException**
- **✔ Throwing an exception that is considered unrecoverable**

Unrecoverable exceptions should not be caught but instead should be handled by a high-level catch-all handler. Therefore, users are not expected to have code that catches these explicit exceptions. The unrecoverable exceptions are:

  - AccessViolationException
  - ExecutionEngineException
  - SEHException
  - StackOverflowException

- ✔ **Throwing a new exception in a new code path**

  The exception must apply only to a new code-path which is executed with new parameter values or state, and that can't be executed by existing code that targets the previous version.

- ✔ **Removing an exception to enable more robust behavior or new scenarios**

  For example, a `Divide` method that previously only handled positive values and threw an ArgumentOutOfRangeException otherwise can be changed to support both negative and positive values without throwing an exception.

- ✔ **Changing the text of an error message**

  Developers should not rely on the text of error messages, which also change based on the user's culture.

- **Throwing an exception in any other case not listed above**

- **Removing an exception in any other case not listed above**

Attributes

- ✔ **Changing the value of an attribute that is *not* observable**

- **Changing the value of an attribute that *is* observable**

- **Removing an attribute**

  In most cases, removing an attribute (such as NonSerializedAttribute) is a breaking change.

Platform support

- ✔ **Supporting an operation on a platform that was previously not supported**

- **Not supporting or now requiring a specific service pack for an operation that was previously supported on a platform**

Internal implementation changes

- **Changing the surface area of an internal type**

  Such changes are generally allowed, although they break private reflection. In some cases, where popular third-party libraries or a large number of developers depend on the internal APIs, such changes may not be allowed.

- **Changing the internal implementation of a member**

  These changes are generally allowed, although they break private reflection. In some cases, where customer code frequently depends on private reflection or where the change introduces unintended side effects, these changes may not be allowed.

- ✔ **Improving the performance of an operation**

The ability to modify the performance of an operation is essential, but such changes can break code that relies upon the current speed of an operation. This is particularly true of code that depends on the timing of asynchronous operations. Note that the performance change should have no effect on other behavior of the API in question; otherwise, the change will be breaking.

- ✔ **Indirectly (and often adversely) changing the performance of an operation**

  If the change in question is not categorized as breaking for some other reason, this is acceptable. Often, actions need to be taken that may include extra operations or that add new functionality. This will almost always affect performance but may be essential to make the API in question function as expected.

- **Changing a synchronous API to asynchronous (and vice versa)**

Code changes

- ✔ **Adding params to a parameter**
- **Changing a struct to a class and vice versa**
- **Adding the checked keyword to a code block**

  This change may cause code that previously executed to throw an OverflowException and is unacceptable.

- **Removing params from a parameter**
- **Changing the order in which events are fired**

  Developers can reasonably expect events to fire in the same order, and developer code frequently depends on the order in which events are fired.

- **Removing the raising of an event on a given action**
- **Changing the number of times given events are called**
- **Adding the FlagsAttribute to an enumeration type**

Breaking change categories

6/21/2019 • 2 minutes to read • Edit Online

Compatibility refers to the ability to compile or execute code on a version of a .NET implementation other than the one with which the code was originally developed. A particular change can affect compatibility in six different ways. The individual kinds of changes that are considered when evaluating compatibility fall into the first five categories.

Behavioral change

A behavioral change represents a change to the behavior of a member. The change may be externally visible (for example, a method may throw a different exception), or it may represent a changed implementation (for example, a change in the way a return value is calculated, the addition or removal of internal method calls, or even a significant performance improvement).

When behavioral changes are externally visible and modify a type's public contract, they are easy to evaluate since they affect binary compatibility. Implementation changes are much more difficult to evaluate; depending on the nature of the change and the frequency and patterns of use of the API, the impact of a change can range from severe to innocuous.

Binary compatibility

Binary compatibility refers to the ability of a consumer of an API to use the API on a newer version without recompilation. Such changes as adding methods or adding a new interface implementation to a type do not affect binary compatibility. However, removing or altering an assembly's public signatures so that consumers can no longer access the same interface exposed by the assembly does affect binary compatibility. A change of this kind is termed a *binary incompatible change*.

Source compatibility

Source compatibility refers to the ability of existing consumers of an API to recompile against a newer version without any source changes. A *source incompatible change* occurs when a consumer needs to modify source code for it to build successfully against a newer version of an API.

Design-time compatibility

Design-time compatibility refers to preserving the design-time experience across versions of Visual Studio and other design-time environments. While this can involve the behavior or the UI of designers, the most important aspect of design-time compatibility concerns project compatibility. A project or solution must be able to be opened and used on a newer version of the design-time environment.

Backwards compatibility

Backwards compatibility refers to the ability of an existing consumer of an API to run against a new version while behaving in the same way. Both behavioral changes and changes in binary compatibility affect backwards compatibility. If a consumer is not able to run or behaves differently when running against the newer version of the API, the API is *backwards incompatible*.

Changes that affect backwards compatibility are strongly discouraged since developers by default expect backwards compatibility in newer versions of an API.

Forward compatibility

Forward compatibility refers to the ability of an existing consumer of an API to run against an older version while exhibiting the same behavior. If a consumer is not able to run or behaves differently when run against an older version of the API, the API is *forward incompatible*.

Maintaining forward compatibility virtually precludes any changes or additions from version to version, since those changes prevent a consumer that targets a later version from running under an earlier version. Developers expect that a consumer that relies on a newer API may not function correctly against the older API.

Maintaining forward compatibility is not a goal of .NET Core.

See also

Evaluate breaking changes in .NET Core

Learn .NET Core and the .NET Core SDK tools by exploring these Tutorials

3/12/2019 • 2 minutes to read • Edit Online

The following tutorials are available for learning about .NET Core.

Building applications with Visual Studio 2017

- Building a C# Hello World application
- Debugging your C# Hello World application
- Publishing your C# Hello World application
- Building a C# class library
- Building a class library with Visual Basic
- Testing a class library
- Consuming a class library
- Azure Cosmos DB: Get started with the SQL API and .NET Core

Building applications with Visual Studio Code

- Get started with C# and Visual Studio Code
- Get started with .NET Core on macOS

Building applications with Visual Studio for Mac

- Get started with .NET Core on macOS using Visual Studio for Mac
- Building a complete .NET Core solution on macOS using Visual Studio for Mac

Building applications with the .NET Core CLI tools

- Get started with .NET Core on Windows/Linux/macOS using the .NET Core CLI tools
- Organizing and testing projects with the .NET Core CLI tools
- Get started with F#

Other

- Unit Testing in .NET Core using dotnet test
- Unit testing with MSTest and .NET Core
- Developing Libraries with Cross Platform Tools
- Hosting .NET Core from native code
- Create a custom template for dotnet new

For tutorials about developing ASP.NET Core web applications, see the ASP.NET Core documentation.

Tutorial: Create an item template

7/14/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part one of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series, you'll learn how to:

- Create a class for an item template
- Create the template config folder and file
- Install a template from a file path
- Test an item template
- Uninstall an item template

Prerequisites

- .NET Core 2.2 SDK or later versions.
- Read the reference article Custom templates for dotnet new.

  The reference article explains the basics about templates and how they're put together. Some of this information will be reiterated here.

- Open a terminal and navigate to the *working\templates* folder.

Create the required folders

This series uses a "working folder" where your template source is contained and a "testing folder" used to test your templates. The working folder and testing folder should be under the same parent folder.

First, create the parent folder, the name does not matter. Then, create a subfolder named*working*. Inside of the *working* folder, create a subfolder named *templates*.

Next, create a folder under the parent folder named *test*. The folder structure should look like the following:

```
parent_folder
├──test
└──working
   └──templates
```

Create an item template

An item template is a specific type of template that contains one or more files. These types of templates are useful when you want to generate something like a config, code, or solution file. In this example, you'll create a class that adds an extension method to the string type.

In your terminal, navigate to the *working\templates* folder and create a new subfolder named *extensions*. Enter the folder.

```
working
└───templates
    └───extensions
```

Create a new file named *CommonExtensions.cs* and open it with your favorite text editor. This class will provide an extension method named `Reverse` that reverses the contents of a string. Paste in the following code and save the file:

```
using System;

namespace System
{
    public static class StringExtensions
    {
        public static string Reverse(this string value)
        {
            var tempArray = value.ToCharArray();
            Array.Reverse(tempArray);
            return new string(tempArray);
        }
    }
}
```

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

Create the template config

Templates are recognized in .NET Core by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at *working\templates\extensions*.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named *.template.config*.

First, create a new subfolder named *.template.config*, enter it. Then, create a new file named *template.json*. Your folder structure should look like this:

```
working
└───templates
    └───extensions
        └───.template.config
                template.json
```

Open the *template.json* with your favorite text editor and paste in the following JSON code and save it:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Code" ],
  "identity": "ExampleTemplate.StringExtensions",
  "name": "Example templates: string extensions",
  "shortName": "stringext",
  "tags": {
    "language": "C#",
    "type": "item"
  }
}
```

This config file contains all the settings for your template. You can see the basic settings, such as `name` and `shortName`, but there's also a `tags/type` value that is set to `item`. This categorizes your template as an item template. There's no restriction on the type of template you create. The `item` and `project` values are common names that .NET Core recommends so that users can easily filter the type of template they're searching for.

The `classifications` item represents the **tags** column you see when you run `dotnet new` and get a list of templates. Users can also search based on classification tags. Don't confuse the `tags` property in the *.json file with the `classifications` tags list. They're two different things unfortunately named similarly. The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Now that you have a valid *.template.config/template.json* file, your template is ready to be installed. In your terminal, navigate to the *extensions* folder and run the following command to install the template located at the current folder:

- **On Windows**: `dotnet new -i .\`
- **On Linux or macOS**: `dotnet new -i ./`

This command outputs the list of templates installed, which should include yours.

```
C:\working\templates\extensions> dotnet new -i .\
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                         Short Name            Language          Tags
-------------------------------------------------------------------------------------------------------------------
-----------------
Example templates: string extensions              stringext             [C#]              Common/Code
Console Application                               console               [C#], F#, VB      Common/Console
Class library                                     classlib              [C#], F#, VB      Common/Library
WPF Application                                   wpf                   [C#], VB          Common/WPF
Windows Forms (WinForms) Application              winforms              [C#], VB          Common/WinForms
Worker Service                                    worker                [C#]              Common/Worker/Web
```

Test the item template

Now that you have an item template installed, test it. Navigate to the *test/* folder and create a new console application with `dotnet new console`. This generates a working project you can easily test with the `dotnet run` command.

```
C:\test> dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\test\test.csproj...
  Restore completed in 54.82 ms for C:\test\test.csproj.

Restore succeeded.
```

```
C:\test> dotnet run
Hello World!
```

Next, run `dotnet new stringext` to generate the *CommonExtensions.cs* from the template.

```
C:\test> dotnet new stringext
The template "Example templates: string extensions" was created successfully.
```

Change the code in *Program.cs* to reverse the `"Hello World"` string with the extension method provided by the template.

```
Console.WriteLine("Hello World!".Reverse());
```

Run the program again and you'll see that the result is reversed.

```
C:\test> dotnet run
!dlroW olleH
```

Congratulations! You created and deployed an item template with .NET Core. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the *test* folder too. This will get you back to a clean state ready for the next major section of this tutorial.

Uninstall the template

Because you installed the template by file path, you must uninstall it with the **absolute** file path. You can see a list of templates installed by running the `dotnet new -u` command. Your template should be listed last. Use the path listed to uninstall your template with the `dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY>` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  C:\working\templates\extensions
    Templates:
      Example templates: string extensions (stringext) C#
```

```
C:\working> dotnet new -u C:\working\templates\extensions
```

Next steps

In this tutorial, you created an item template. To learn how to create a project template, continue this tutorial series.

Create a project template

Tutorial: Create a project template

7/14/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part two of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series you'll learn how to:

- Create the resources of a project template
- Create the template config folder and file
- Install a template from a file path
- Test an item template
- Uninstall an item template

Prerequisites

- Complete part 1 of this tutorial series.
- Open a terminal and navigate to the *working\templates* folder.

Create a project template

Project templates produce ready-to-run projects that make it easy for users to start with a working set of code. .NET Core includes a few project templates such as a console application or a class library. In this example, you'll create a new console project that enables C# 8.0 and produces an `async main` entry point.

In your terminal, navigate to the *working\templates* folder and create a new subfolder named *consoleasync*. Enter the subfolder and run `dotnet new console` to generate the standard console application. You'll be editing the files produced by this template to create a new template.

```
working
└───templates
    └───consoleasync
            consoleasync.csproj
            Program.cs
```

Modify Program.cs

Open up the *program.cs* file. The console project doesn't use an asynchronous entry point, so let's add that. Change your code to the following and save the file:

```
using System;
using System.Threading.Tasks;

namespace consoleasync
{
    class Program
    {
        static async Task Main(string[] args)
        {
            await Console.Out.WriteAsync("Hello World with C# 8.0!");
        }
    }
}
```

Modify consoleasync.csproj

Let's update the C# language version the project uses to version 8.0. Edit the *consoleasync.csproj* file and add the `<LangVersion>` setting to a `<PropertyGroup>` node.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>

    <LangVersion>8.0</LangVersion>

  </PropertyGroup>

</Project>
```

Build the project

Before you complete a project template, you should test it to make sure it compiles and runs correctly. In your terminal, run the `dotnet run` command and you should see the following output:

```
C:\working\templates\consoleasync> dotnet run
Hello World with C# 8.0!
```

You can delete the *obj* and *bin* folders created by using `dotnet run`. Deleting these files ensures your template only includes the files related to your template and not any files that result of a build action.

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

Create the template config

Templates are recognized in .NET Core by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at *working\templates\consoleasync*.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named *.template.config*.

First, create a new subfolder named *.template.config*, enter it. Then, create a new file named *template.json*. Your folder structure should look like this:

```
working
└───templates
    └───consoleasync
        └───.template.config
                template.json
```

Open the *template.json* with your favorite text editor and paste in the following json code and save it:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Console", "C#8" ],
  "identity": "ExampleTemplate.AsyncProject",
  "name": "Example templates: async project",
  "shortName": "consoleasync",
  "tags": {
    "language": "C#",
    "type": "project"
  }
}
```

This config file contains all of the settings for your template. You can see the basic settings such as `name` and `shortName` but also there's a `tags/type` value that's set to `project`. This designates your template as a project template. There's no restriction on the type of template you create. The `item` and `project` values are common names that .NET Core recommends so that users can easily filter the type of template they're searching for.

The `classifications` item represents the **tags** column you see when you run `dotnet new` and get a list of templates. Users can also search based on classification tags. Don't confuse the `tags` property in the json file with the `classifications` tags list. They're two different things unfortunately named similarly. The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Now that you have a valid *.template.config/template.json* file, your template is ready to be installed. Before you install the template, make sure that you delete any extra files folders and files you don't want included in your template, like the *bin* or *obj* folders. In your terminal, navigate to the *consoleasync* folder and run `dotnet new -i .\` to install the template located at the current folder. If you're using a Linux or MacOS operating system, use a forward slash: `dotnet new -i ./`.

This command outputs the list of templates installed, which should include yours.

```
C:\working\templates\consoleasync> dotnet new -i .\
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                       Short Name            Language          Tags
----------------------------------------------------------------------------------------------------------
-----------------
Console Application                             console               [C#], F#, VB      Common/Console
Example templates: async project                consoleasync          [C#]              Common/Console/C#8
Class library                                   classlib              [C#], F#, VB      Common/Library
WPF Application                                 wpf                   [C#], VB          Common/WPF
Windows Forms (WinForms) Application            winforms              [C#], VB          Common/WinForms
Worker Service                                  worker                [C#]              Common/Worker/Web
```

Test the project template

Now that you have an item template installed, test it. Navigate to the *test* folder and create a new console application with `dotnet new console`. This generates a working project you can easily test with the `dotnet run` command.

```
C:\test> dotnet new consoleasync
The template "Example templates: async project" was created successfully.
```

```
C:\test> dotnet run
Hello World with C# 8.0!
```

Congratulations! You created and deployed a project template with .NET Core. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the *test* folder too. This will get you back to a clean state ready for the next major section of this tutorial.

Uninstall the template

Because you installed the template by using a file path, you must uninstall it with the **absolute** file path. You can see a list of templates installed by running the `dotnet new -u` command. Your template should be listed last. Use the path listed to uninstall your template with the `dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY>` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  C:\working\templates\consoleasync
    Templates:
      Example templates: async project (consoleasync) C#
```

```
C:\working> dotnet new -u C:\working\templates\consoleasync
```

Next steps

In this tutorial, you created a project template. To learn how to package both the item and project templates into an easy-to-use file, continue this tutorial series.

Create a template pack

Tutorial: Create a template pack

7/14/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part three of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series you'll learn how to:

- Create a _.csproj* project to build a template pack
- Configure the project file for packing
- Install a template from a NuGet package file
- Uninstall a template by package ID

Prerequisites

- Complete part 1 and part 2 of this tutorial series.

  This tutorial uses the two templates created in the first two parts of this tutorial. It's possible you can use a different template as long as you copy the template as a folder into the *working\templates* folder.

- Open a terminal and navigate to the *working\templates* folder.

Create a template pack project

A template pack is one or more templates packaged into a file. When you install or uninstall a pack, all templates contained in the pack are added or removed, respectively. The previous parts of this tutorial series only worked with individual templates. To share a non-packed template, you have to copy the template folder and install via that folder. Because a template pack can have more than one template in it, and is a single file, sharing is easier.

Template packs are represented by a NuGet package (*.nupkg*) file. And, like any NuGet package, you can upload the template pack to a NuGet feed. The `dotnet new -i` command supports installing template pack from a NuGet package feed. Additionally, you can install a template pack from a *.nupkg* file directly.

Normally you use a C# project file to compile code and produce a binary. However, the project can also be used to generate a template pack. By changing the settings of the *.csproj*, you can prevent it from compiling any code and instead include all the assets of your templates as resources. When this project is built, it produces a template pack NuGet package.

The pack you'll create will include the item template and package template previously created. Because we grouped the two templates into the *working\templates* folder, we can use the *working* folder for the *.csproj* file.

In your terminal, navigate to the *working* folder. Create a new project and set the name to `templatepack` and the output folder to the current folder.

```
dotnet new console -n templatepack -o .
```

The `-n` parameter sets the *.csproj* filename to *templatepack.csproj* and the `-o` parameters creates the files in the current directory. You should see a result similar to the following output.

```
C:\working> dotnet new console -n templatepack -o .
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on .\templatepack.csproj...
  Restore completed in 52.38 ms for C:\working\templatepack.csproj.

Restore succeeded.
```

Next, open the *templatepack.csproj* file in your favorite editor and replace the content with the following XML:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>

    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

The `<PropertyGroup>` settings in the XML above is broken into three groups. The first group deals with properties required for a NuGet package. The three `<Package` settings have to do with the NuGet package properties to identify your package on a NuGet feed. Specifically the `<PacakgeId>` value is used to uninstall the template pack with a single name instead of a directory path. It can also be used to install the template pack from a NuGet feed. The remaining settings such as `<Title>` and `<Tags>` have to do with metadata displayed on the NuGet feed. For more information about NuGet settings, see NuGet and MSBuild properties.

The `<TargetFramework>` setting must be set so that MSBuild will run properly when you run the pack command to compile and pack the project.

The last three settings have to do with configuring the project correctly to include the templates in the appropriate folder in the NuGet pack when it's created.

The `<ItemGroup>` contains two settings. First, the `<Content>` setting includes everything in the *templates* folder as content. It's also set to exclude any *bin* folder or *obj* folder to prevent any compiled code (if you tested and compiled your templates) from being included. Second, the `<Compile>` setting excludes all code files from compiling no matter where they're located. This setting prevents the project being used to create a template pack from trying to compile the code in the *templates* folder hierarchy.

Build and install

Save this file and then run the pack command

```
dotnet pack
```

This command will build your project and create a NuGet package in This should be the *working\bin\Debug* folder.

```
C:\working> dotnet pack
Microsoft (R) Build Engine version 16.2.0-preview-19278-01+d635043bd for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 123.86 ms for C:\working\templatepack.csproj.

  templatepack -> C:\working\bin\Debug\netstandard2.0\templatepack.dll
  Successfully created package 'C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg'.
```

Next, install the template pack file with the `dotnet new -i PATH_TO_NUPKG_FILE` command.

```
C:\working> dotnet new -i C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                       Short Name            Language          Tags
-----------------------------------------------------------------------------------------------------------
-----------------
Example templates: string extensions            stringext             [C#]              Common/Code
Console Application                             console               [C#], F#, VB      Common/Console
Example templates: async project                consoleasync          [C#]              Common/Console/C#8
Class library                                   classlib              [C#], F#, VB      Common/Library
```

If you uploaded the NuGet package to a NuGet feed, you can use the `dotnet new -i PACKAGEID` command where `PACKAGEID` is the same as the `<PackageId>` setting from the *.csproj* file. This package ID is the same as the NuGet package identifier.

Uninstall the template pack

No matter how you installed the template pack, either with the *.nupkg* file directly or by NuGet feed, removing a template pack is the same. Use the `<PackageId>` of the template you want to uninstall. You can get a list of templates that are installed by running the `dotnet new -u` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  AdatumCorporation.Utility.Templates
    Templates:
      Example templates: async project (consoleasync) C#
      Example templates: string extensions (stringext) C#
```

Run `dotnet new -u AdatumCorporation.Utility.Templates` to uninstall the template. The `dotnet new` command will output help information that should omit the templates you previously installed.

Congratulations! you've installed and uninstalled a template pack.

Next steps

To learn more about templates, most of which you've already learned, see the Custom templates for dotnet new article.

- dotnet/templating GitHub repo Wiki
- dotnet/dotnet-template-samples GitHub repo
- *template.json* schema at the JSON Schema Store

Get started with .NET Core on macOS

5/15/2019 • 6 minutes to read • Edit Online

This document provides the steps and workflow to create a .NET Core solution for macOS. Learn how to create projects, unit tests, use the debugging tools, and incorporate third-party libraries via NuGet.

> **NOTE**
> This article uses Visual Studio Code on macOS.

Prerequisites

Install the .NET Core SDK. The .NET Core SDK includes the latest release of the .NET Core framework and runtime.

Install Visual Studio Code. During the course of this article, you also install Visual Studio Code extensions that improve the .NET Core development experience.

Install the Visual Studio Code C# extension by opening Visual Studio Code and pressing `F1` to open the Visual Studio Code palette. Type **ext install** to see the list of extensions. Select the C# extension. Restart Visual Studio Code to activate the extension. For more information, see the Visual Studio Code C# Extension documentation.

Get started

In this tutorial, you create three projects: a library project, tests for that library project, and a console application that makes use of the library. You can view or download the source for this topic at the dotnet/samples repository on GitHub. For download instructions, see Samples and Tutorials.

Start Ctrl+` (the backquote or backtick character) or select **View > Integrated Terminal** from the menu to open an embedded terminal in Visual Studio Code. You can still open an external shell with the Explorer **Open in Command Prompt** command (**Open in Terminal** on Mac or Linux) if you prefer to work outside of Visual Studio Code.

Begin by creating a solution file, which serves as a container for one or more .NET Core projects. In the terminal, create a *golden* folder and open the folder. This folder is the root of your solution. Run the `dotnet new` command to create a new solution, *golden.sln*:

```
dotnet new sln
```

From the *golden* folder, execute the following command to create a library project, which produces two files,*library.csproj* and *Class1.cs*, in the *library* folder:

```
dotnet new classlib -o library
```

Execute the `dotnet sln` command to add the newly created *library.csproj* project to the solution:

```
dotnet sln add library/library.csproj
```

The *library.csproj* file contains the following information:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard1.4</TargetFramework>
  </PropertyGroup>

</Project>
```

Our library methods serialize and deserialize objects in JSON format. To support JSON serialization and deserialization, add a reference to the `Newtonsoft.Json` NuGet package. The `dotnet add` command adds new items to a project. To add a reference to a NuGet package, use the `dotnet add package` command and specify the name of the package:

```
dotnet add library package Newtonsoft.Json
```

This adds `Newtonsoft.Json` and its dependencies to the library project. Alternatively, manually edit the *library.csproj* file and add the following node:

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="10.0.1" />
</ItemGroup>
```

Execute `dotnet restore`, (see note) which restores dependencies and creates an *obj* folder inside *library* with three files in it, including a *project.assets.json* file:

```
dotnet restore
```

In the *library* folder, rename the file *Class1.cs* to *Thing.cs*. Replace the code with the following:

```
using static Newtonsoft.Json.JsonConvert;

namespace Library
{
    public class Thing
    {
        public int Get(int left, int right) =>
            DeserializeObject<int>($"{left + right}");
    }
}
```

The `Thing` class contains one public method, `Get`, which returns the sum of two numbers but does so by converting the sum into a string and then deserializing it into an integer. This makes use of a number of modern C# features, such as `using static` directives, expression-bodied members, and string interpolation.

Build the library with the `dotnet build` command. This produces a *library.dll* file under *golden/library/bin/Debug/netstandard1.4*:

```
dotnet build
```

Create the test project

Build a test project for the library. From the *golden* folder, create a new test project:

```
dotnet new xunit -o test-library
```

Add the test project to the solution:

```
dotnet sln add test-library/test-library.csproj
```

Add a project reference the library you created in the previous section so that the compiler can find and use the library project. Use the `dotnet add reference` command:

```
dotnet add test-library/test-library.csproj reference library/library.csproj
```

Alternatively, manually edit the *test-library.csproj* file and add the following node:

```
<ItemGroup>
  <ProjectReference Include="..\library\library.csproj" />
</ItemGroup>
```

Now that the dependencies have been properly configured, create the tests for your library. Open *UnitTest1.cs* and replace its contents with the following code:

```
using Library;
using Xunit;

namespace TestApp
{
    public class LibraryTests
    {
        [Fact]
        public void TestThing() {
            Assert.NotEqual(42, new Thing().Get(19, 23));
        }
    }
}
```

Note that you assert the value 42 is not equal to 19+23 (or 42) when you first create the unit test (`Assert.NotEqual`), which will fail. An important step in building unit tests is to create the test to fail once first to confirm its logic.

From the *golden* folder, execute the following commands:

```
dotnet restore
dotnet test test-library/test-library.csproj
```

These commands will recursively find all projects to restore dependencies, build them, and activate the xUnit test runner to run the tests. The single test fails, as you expect.

Edit the *UnitTest1.cs* file and change the assertion from `Assert.NotEqual` to `Assert.Equal` . Execute the following command from the *golden* folder to re-run the test, which passes this time:

```
dotnet test test-library/test-library.csproj
```

Create the console app

The console app you create over the following steps takes a dependency on the library project you created earlier and calls its library method when it runs. Using this pattern of development, you see how to create reusable libraries for multiple projects.

Create a new console application from the *golden* folder:

```
dotnet new console -o app
```

Add the console app project to the solution:

```
dotnet sln add app/app.csproj
```

Create the dependency on the library by running the `dotnet add reference` command:

```
dotnet add app/app.csproj reference library/library.csproj
```

Run `dotnet restore` (see note) to restore the dependencies of the three projects in the solution. Open *Program.cs* and replace the contents of the `Main` method with the following line:

```
WriteLine($"The answer is {new Thing().Get(19, 23)}");
```

Add two `using` directives to the top of the *Program.cs* file:

```
using static System.Console;
using Library;
```

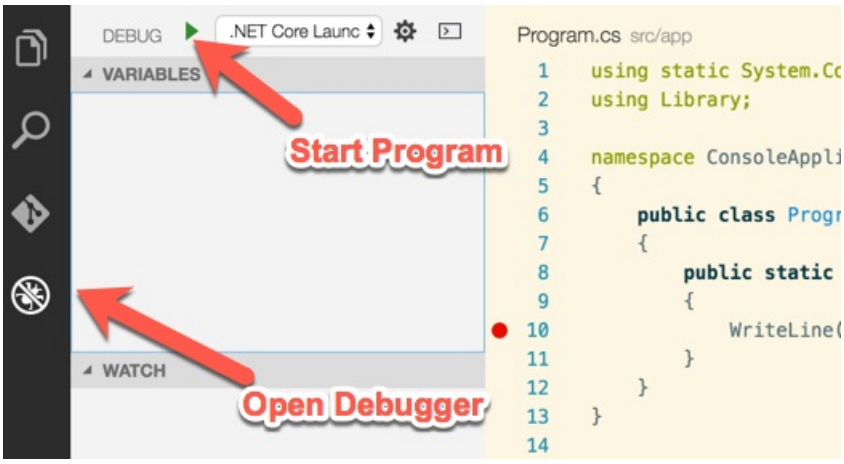
Execute the following `dotnet run` command to run the executable, where the `-p` option to `dotnet run` specifies the project for the main application. The app produces the string "The answer is 42".

```
dotnet run -p app/app.csproj
```

Debug the application

Set a breakpoint at the `WriteLine` statement in the `Main` method. Do this by either pressing the F9 key when the cursor is over the `WriteLine` line or by clicking the mouse in the left margin on the line where you want to set the breakpoint. A red circle will appear in the margin next to the line of code. When the breakpoint is reached, code execution will stop *before* the breakpoint line is executed.

Open the debugger tab by selecting the Debug icon in the Visual Studio Code toolbar, selecting **View > Debug** from the menu bar, or using the keyboard shortcut CTRL+SHIFT+D:



Press the Play button to start the application under the debugger. The app begins execution and runs to the breakpoint, where it stops. Step into the `Get` method and make sure that you have passed in the correct arguments. Confirm that the answer is 42.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Get started with .NET Core on macOS using Visual Studio for Mac

8/20/2019 • 2 minutes to read • Edit Online

Visual Studio for Mac provides a full-featured Integrated Development Environment (IDE) for developing .NET Core applications. This topic walks you through building a simple console application using Visual Studio for Mac and .NET Core.

> **NOTE**
>
> Your feedback is highly valued. There are two ways you can provide feedback to the development team on Visual Studio for Mac:
>
> - In Visual Studio for Mac, select **Help** > **Report a Problem** from the menu or **Report a Problem** from the Welcome screen, which will open a window for filing a bug report. You can track your feedback in the Developer Community portal.
> - To make a suggestion, select **Help** > **Provide a Suggestion** from the menu or **Provide a Suggestion** from the Welcome screen, which will take you to the Visual Studio for Mac Developer Community webpage.

Prerequisites

See the Prerequisites for .NET Core on Mac topic.

Check the .NET Core Support guide to ensure you're using a supported version of .NET Core.

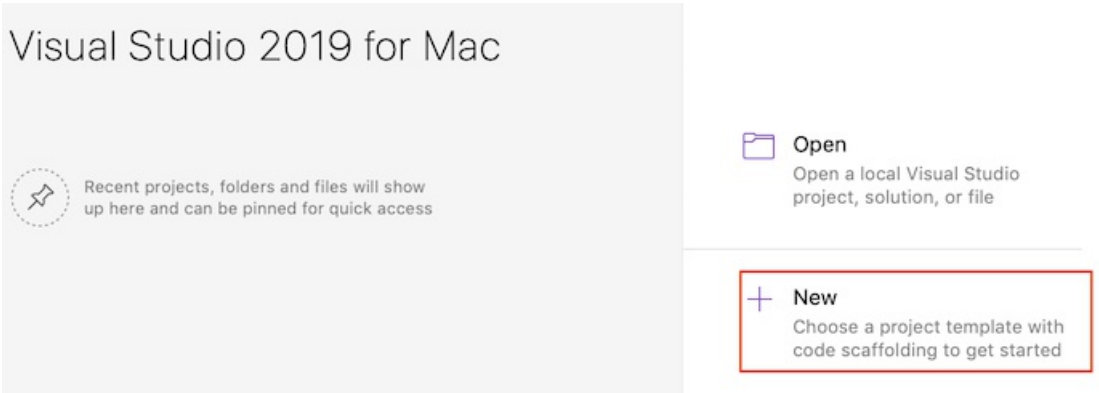
Get started

If you've already installed the prerequisites and Visual Studio for Mac, skip this section and proceed to Creating a project. Follow these steps to install the prerequisites and Visual Studio for Mac:

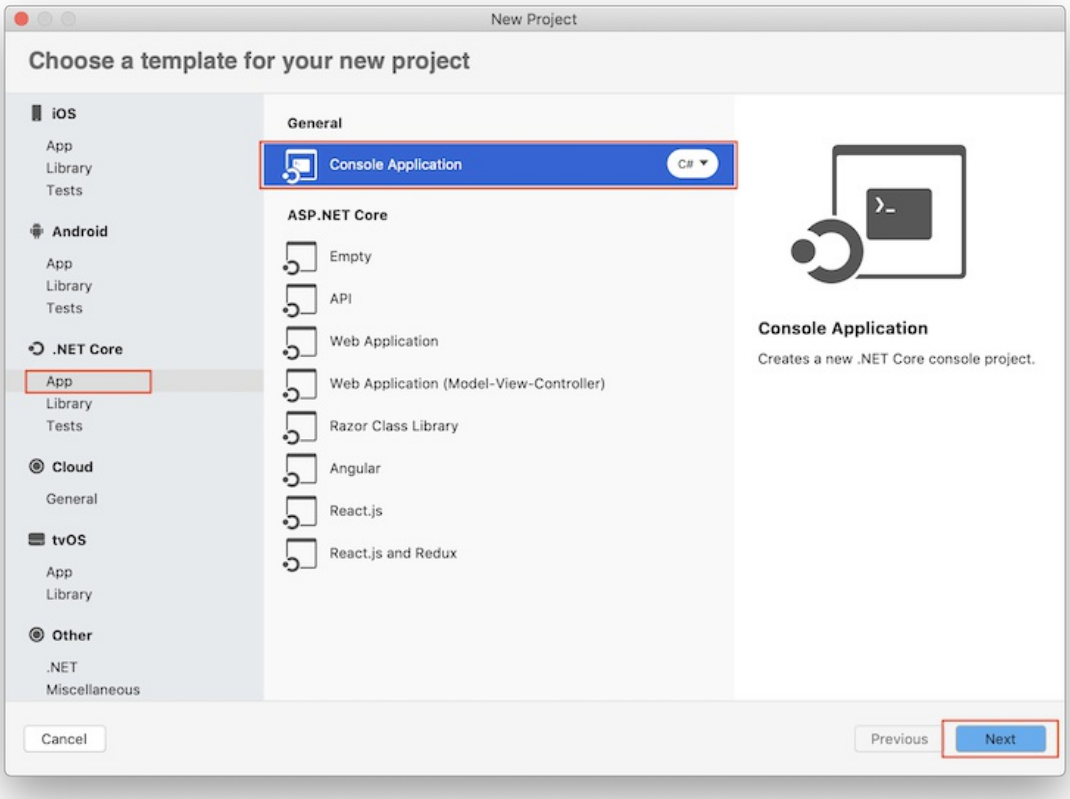
Download the Visual Studio for Mac installer. Run the installer. Read and accept the license agreement. During the install, select the option to install .NET Core. You're provided the opportunity to install Xamarin, a cross-platform mobile app development technology. Installing Xamarin and its related components is optional for .NET Core development. For a walk-through of the Visual Studio for Mac install process, see Visual Studio for Mac documentation. When the install is complete, start the Visual Studio for Mac IDE.

Creating a project

1. Select **New** on the Start Window.

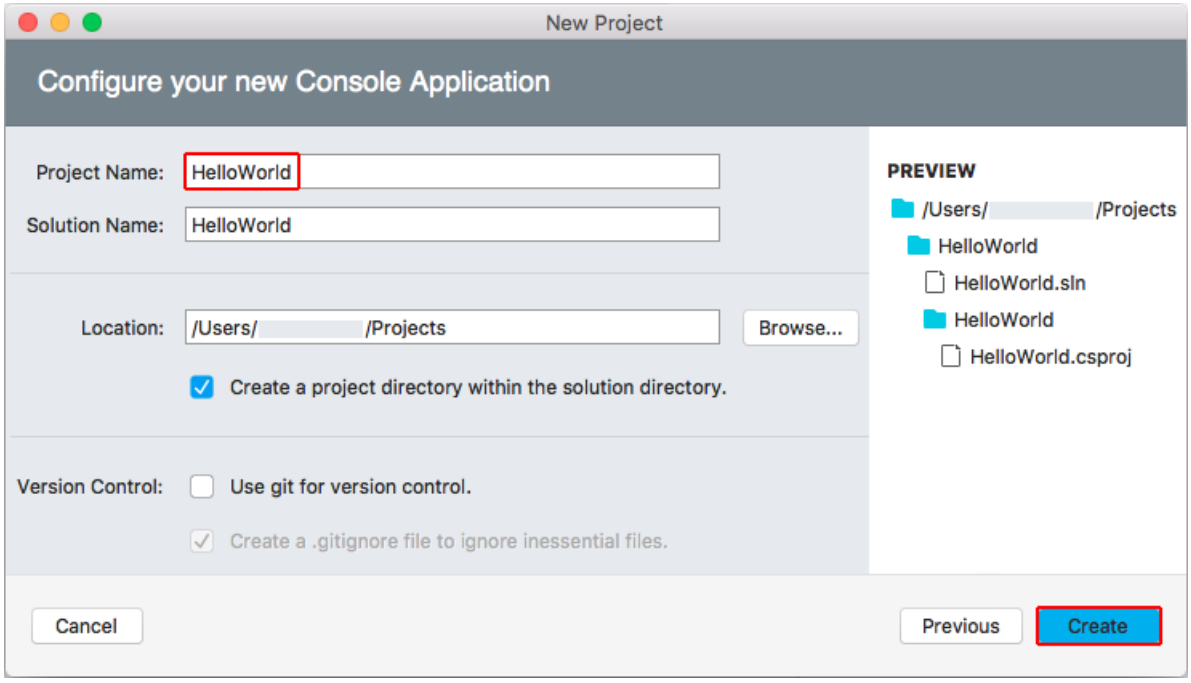


2. In the **New Project** dialog, select **App** under the **.NET Core** node. Select the **Console Application** template followed by **Next**.

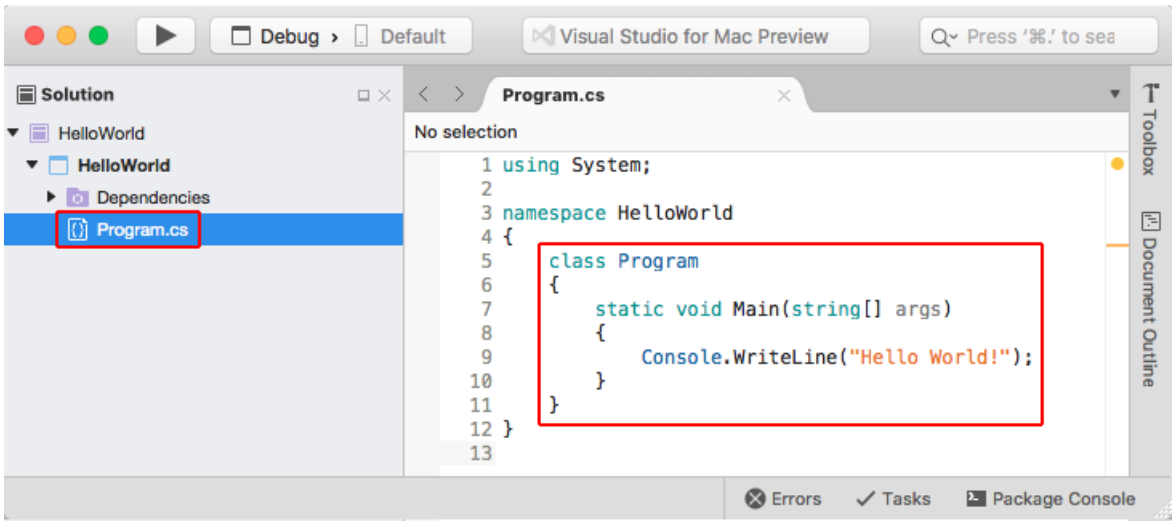


3. If you have more than one version of .NET Core installed, select the target framework for your project.

4. Type "HelloWorld" for the **Project Name**. Select **Create**.

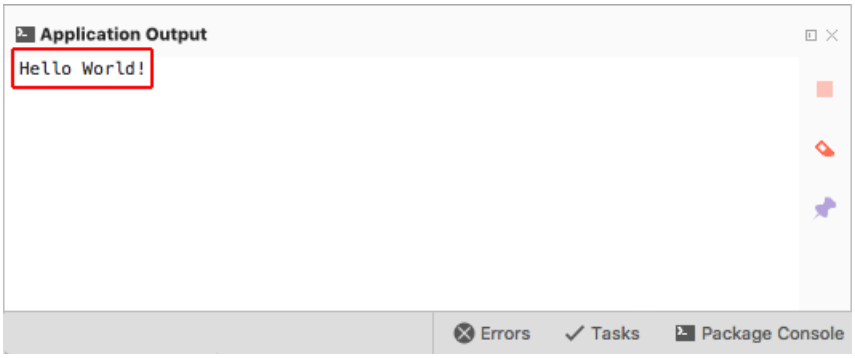


5. Wait while the project's dependencies are restored. The project has a single C# file, *Program.cs*, containing a `Program` class with a `Main` method. The `Console.WriteLine` statement will output "Hello World!" to the console when the app is run.



Run the application

Run the app in Debug mode using ⌘ ↵ (command + enter) or in Release mode using ⌥ ⌘ ↵ (option + command + enter).



Next step

The Building a complete .NET Core solution on macOS using Visual Studio for Mac topic shows you how to build a complete .NET Core solution that includes a reusable library and unit testing.

Building a complete .NET Core solution on macOS using Visual Studio for Mac

8/20/2019 • 9 minutes to read • Edit Online

Visual Studio for Mac provides a full-featured Integrated Development Environment (IDE) for developing .NET Core applications. This topic walks you through building a .NET Core solution that includes a reusable library and unit testing.

This tutorial shows you how to create an application that accepts a search word and a string of text from the user, counts the number of times the search word appears in the string using a method in a class library, and returns the result to the user. The solution also includes unit testing for the class library as an introduction to unit testing concepts. If you prefer to proceed through the tutorial with a complete sample, download the sample solution. For download instructions, see Samples and Tutorials.

> **NOTE**
>
> Your feedback is highly valued. There are two ways you can provide feedback to the development team on Visual Studio for Mac:
>
> - In Visual Studio for Mac, select **Help** > **Report a Problem** from the menu or **Report a Problem** from the Welcome screen, which opens a window for filing a bug report. You can track your feedback in the Developer Community portal.
> - To make a suggestion, select **Help** > **Provide a Suggestion** from the menu or **Provide a Suggestion** from the Welcome screen, which takes you to the Visual Studio for Mac Developer Community webpage.

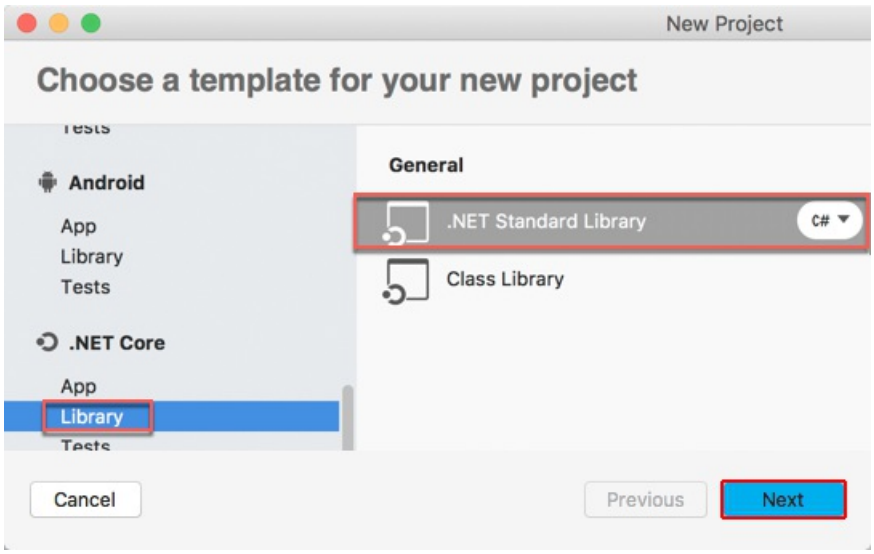
Prerequisites

- OpenSSL (if running .NET Core 1.1): See the Prerequisites for .NET Core on Mac topic.
- .NET Core SDK 1.1 or later
- Visual Studio 2017 for Mac

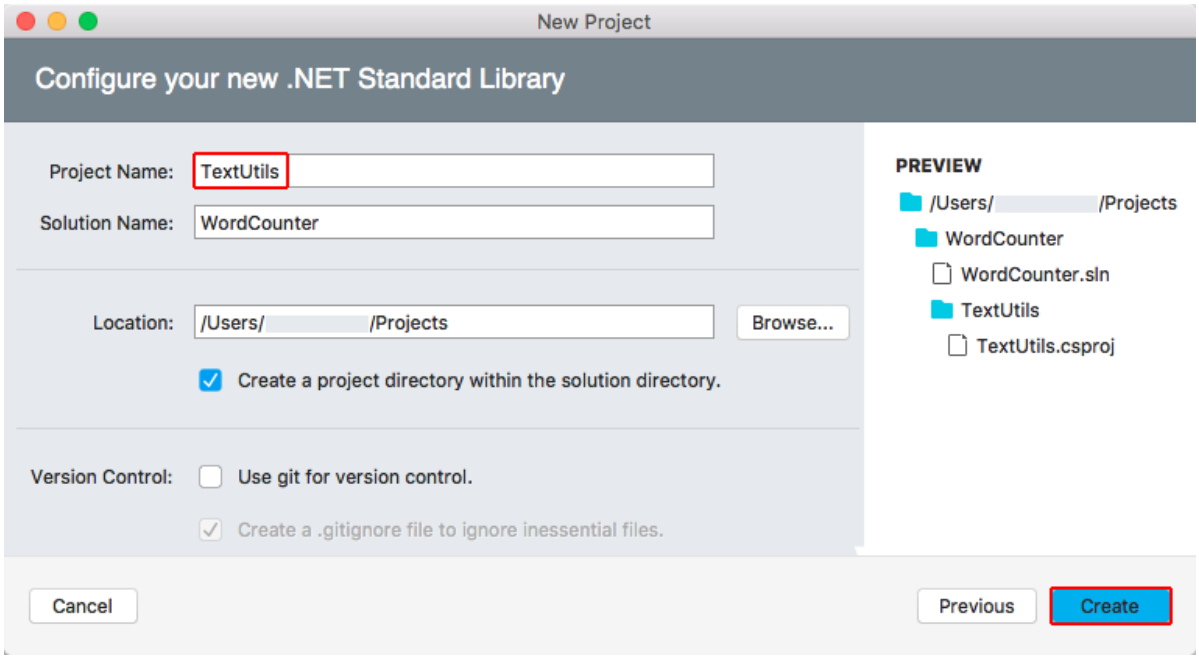
For more information on prerequisites, see the Prerequisites for .NET Core on Mac. For the full system requirements of Visual Studio 2017 for Mac, see Visual Studio 2017 for Mac Product Family System Requirements.

Building a library

1. On the Welcome screen, select **New Project**. In the **New Project** dialog under the **.NET Core** node, select the **.NET Standard Library** template. This creates a .NET Standard library that targets .NET Core as well as any other .NET implementation that supports version 2.0 of the .NET Standard. Select **Next**.



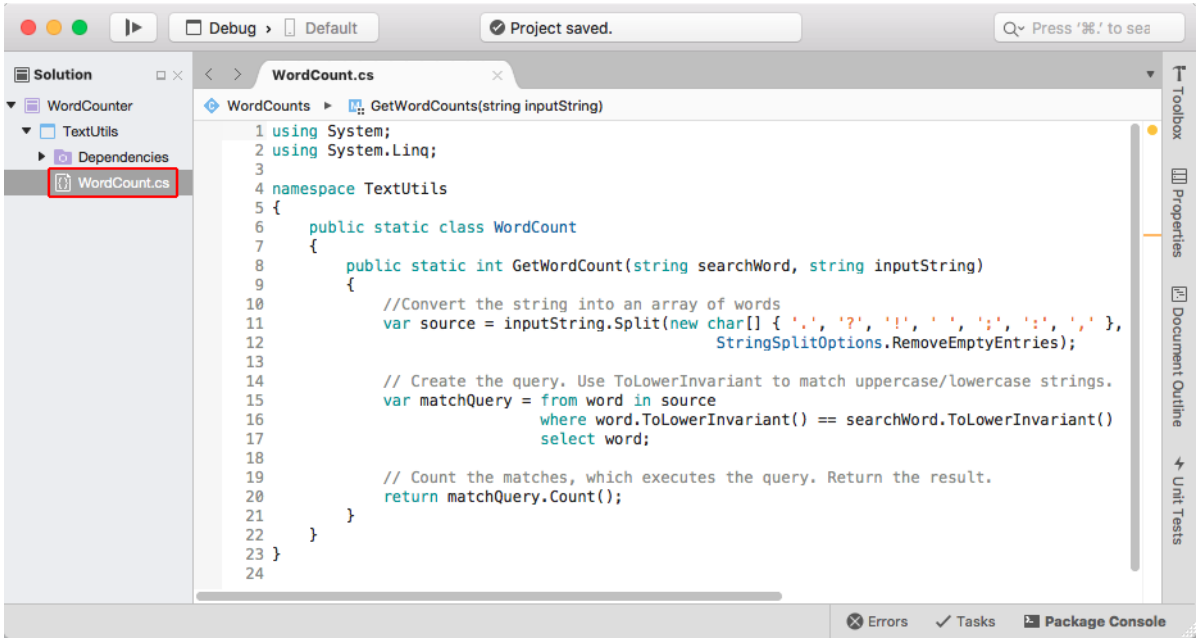
2. Name the project "TextUtils" (a short name for "Text Utilities") and the solution "WordCounter". Leave **Create a project directory within the solution directory** checked. Select **Create**.



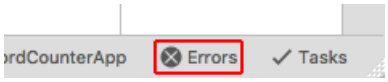
3. In the **Solution** sidebar, expand the `TextUtils` node to reveal the class file provided by the template, *Class1.cs*. Right-click the file, select **Rename** from the context menu, and rename the file to *WordCount.cs*. Open the file and replace the contents with the following code:

```
using System;
using System.Linq;

namespace TextUtils
{
    public static class WordCount
    {
        public static int GetWordCount(string searchWord, string inputString)
        {
            // Null check these variables and determine if they have values.
            if (string.IsNullOrEmpty(searchWord) || string.IsNullOrEmpty(inputString))
            {
                return 0;
            }

            // Convert the string into an array of words.
            var source = inputString.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                           StringSplitOptions.RemoveEmptyEntries);

            // Create the query. Use ToLowerInvariant to match uppercase/lowercase strings.
            var matchQuery = from word in source
                             where word.ToLowerInvariant() == searchWord.ToLowerInvariant()
                             select word;

            // Count the matches, which executes the query. Return the result.
            return matchQuery.Count();
        }
    }
}
```

4. Save the file by using any of three different methods: use the keyboard shortcut ⌘+s, select **File** > **Save** from the menu, or right-click on the file's tab and select **Save** from the contextual menu. The following image shows the IDE window:



5. Select **Errors** in the margin at the bottom of the IDE window to open the **Errors** panel. Select the **Build Output** button.



6. Select **Build** > **Build All** from the menu.

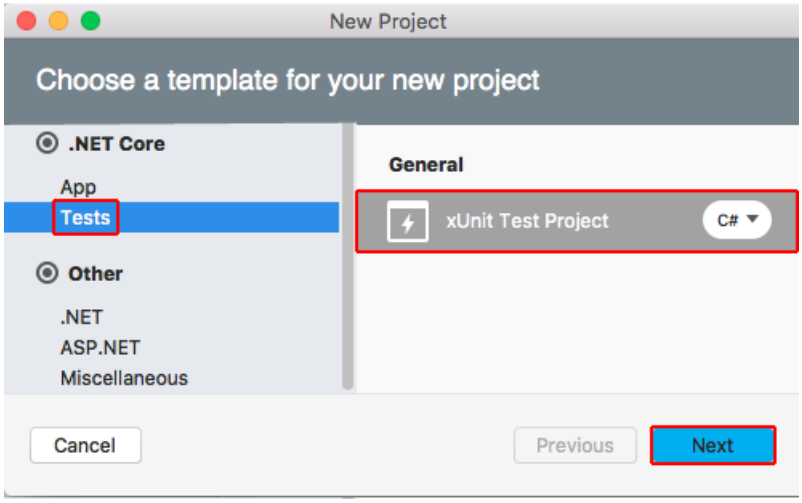
   The solution builds. The build output panel shows that the build is successful.

```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:05.13

---------------------- Done ----------------------

Build successful.
```

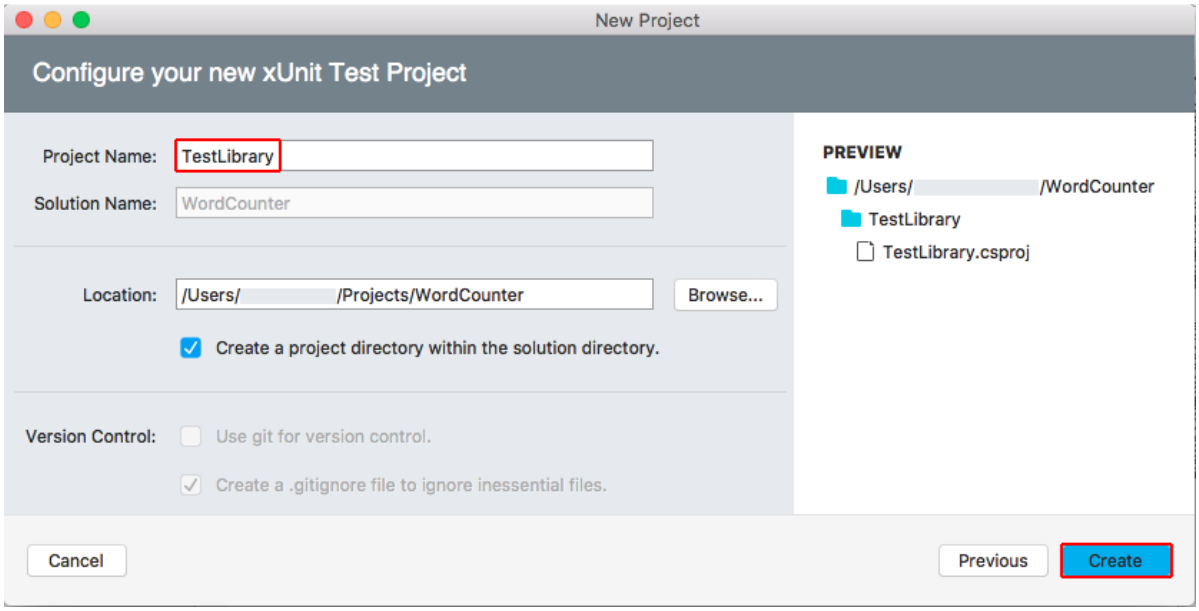
Creating a test project

Unit tests provide automated software testing during your development and publishing. The testing framework that you use in this tutorial is xUnit (version 2.2.0 or later), which is installed automatically when the xUnit test project is added to the solution in the following steps:

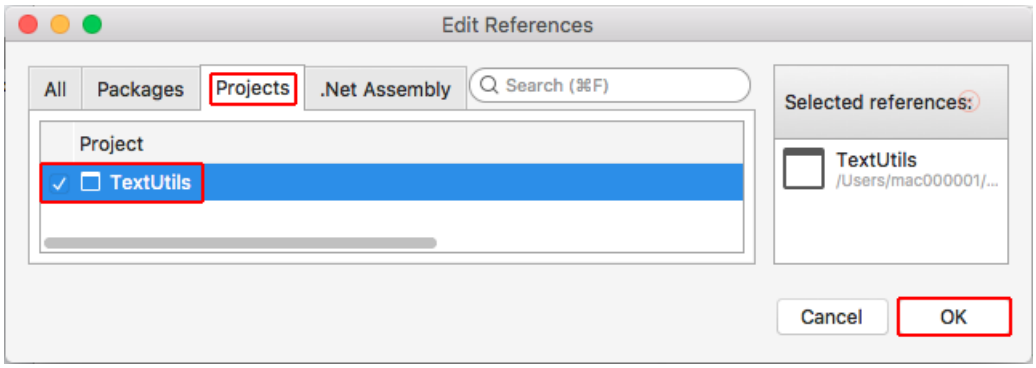
1. In the **Solution** sidebar, right-click the `WordCounter` solution and select **Add** > **Add New Project**.
2. In the **New Project** dialog, select **Tests** from the **.NET Core** node. Select the **xUnit Test Project** followed by **Next**.



3. Name the new project "TestLibrary" and select **Create**.



4. In order for the test library to work with the `WordCount` class, add a reference to the `TextUtils` project. In the **Solution** sidebar, right-click **Dependencies** under **TestLibrary**. Select **Edit References** from the context menu.
5. In the **Edit References** dialog, select the **TextUtils** project on the **Projects** tab. Select **OK**.



6. In the **TestLibrary** project, rename the *UnitTest1.cs* file to *TextUtilsTests.cs*.

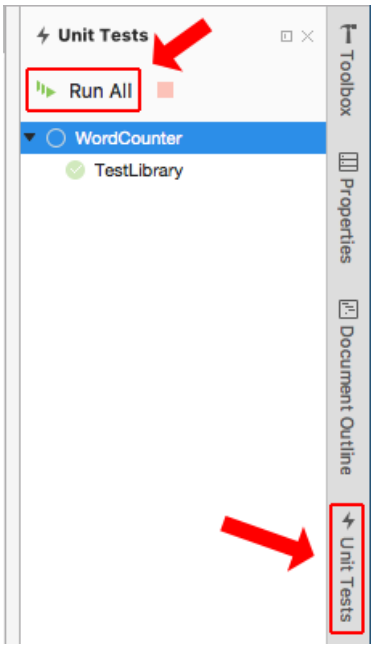
7. Open the file and replace the code with the following:

```
using Xunit;
using TextUtils;
using System.Diagnostics;

namespace TestLibrary
{
    public class TextUtils_GetWordCountShould
    {
        [Fact]
        public void IgnoreCasing()
        {
            var wordCount = WordCount.GetWordCount("Jack", "Jack jack");

            Assert.NotEqual(2, wordCount);
        }
    }
}
```

The following image shows the IDE with the unit test code in place. Pay attention to the `Assert.NotEqual` statement.

It's important to make a new test fail once to confirm its testing logic is correct. The method passes in the name "Jack" (uppercase) and a string with "Jack" and "jack" (uppercase and lowercase). If the `GetWordCount` method is working properly, it returns a count of two instances of the search word. In order to fail this test on purpose, you first implement the test asserting that two instances of the search word "Jack" aren't returned by the `GetWordCount` method. Continue to the next step to fail the test on purpose.

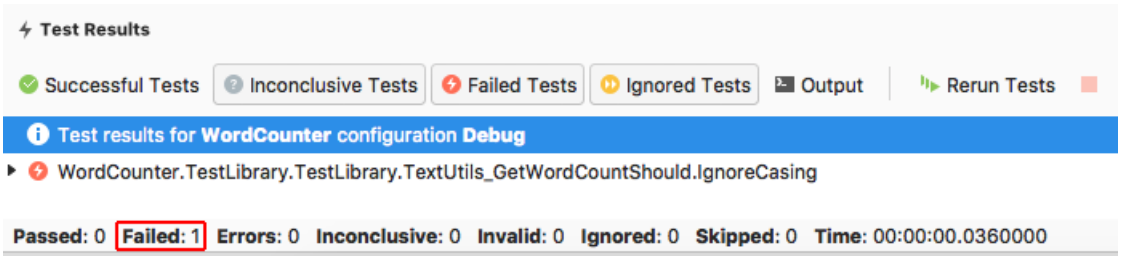
8. Open the **Unit Tests** panel on the right side of the screen.



9. Click the **Dock** icon to keep the panel open.

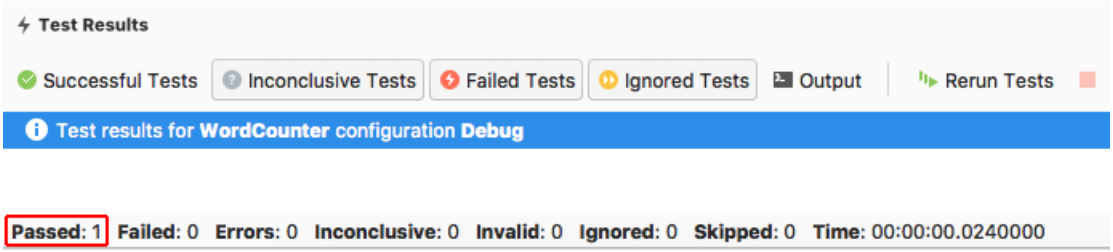
10. Click the **Run All** button.

    The test fails, which is the correct result. The test method asserts that two instances of the `inputString`, "Jack," aren't returned from the string "Jack jack" provided to the `GetWordCount` method. Since word casing was factored out in the `GetWordCount` method, two instances are returned. The assertion that 2 *is not equal to* 2 fails. This is the correct outcome, and the logic of our test is good.



11. Modify the `IgnoreCasing` test method by changing `Assert.NotEqual` to `Assert.Equal`. Save the file by using the keyboard shortcut ⌘+s, **File** > **Save** from the menu, or right-clicking on the file's tab and selecting **Save** from the context menu.

    You expect that the `searchWord` "Jack" returns two instances with `inputString` "Jack jack" passed into `GetWordCount`. Run the test again by clicking the **Run Tests** button in the **Unit Tests** panel or the **Rerun Tests** button in the **Test Results** panel at the bottom of the screen. The test passes. There are two instances of "Jack" in the string "Jack jack" (ignoring casing), and the test assertion is `true`.



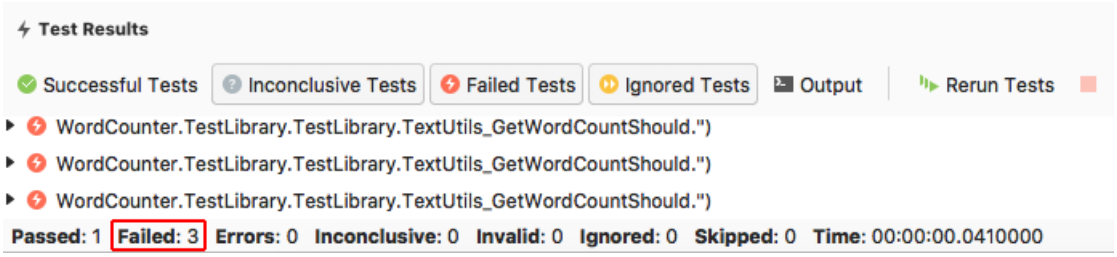
12. Testing individual return values with a `Fact` is only the beginning of what you can do with unit testing. Another powerful technique allows you to test several values at once using a `Theory`. Add the following

method to your `TextUtils_GetWordCountShould` class. You have two methods in the class after you add this method:

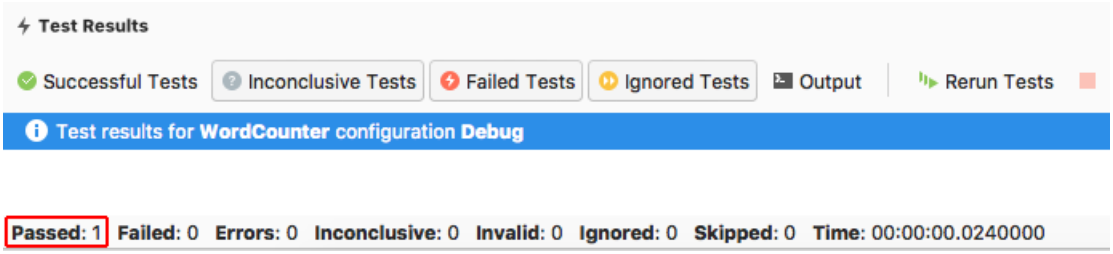
```
[Theory]
[InlineData(0, "Ting", "Does not appear in the string.")]
[InlineData(1, "Ting", "Ting appears once.")]
[InlineData(2, "Ting", "Ting appears twice with Ting.")]
public void CountInstancesCorrectly(int count,
                                    string searchWord,
                                    string inputString)
{
    Assert.NotEqual(count, WordCount.GetWordCount(searchWord,
                                                  inputString));
}
```

The `CountInstancesCorrectly` checks that the `GetWordCount` method counts correctly. The `InlineData` provides a count, a search word, and an input string to check. The test method runs once for each line of data. Note once again that you're asserting a failure first by using `Assert.NotEqual`, even when you know that the counts in the data are correct and that the values match the counts returned by the `GetWordCount` method. Performing the step of failing the test on purpose might seem like a waste of time at first, but checking the logic of the test by failing it first is an important check on the logic of your tests. When you come across a test method that passes when you expect it to fail, you've found a bug in the logic of the test. It's worth the effort to take this step every time you create a test method.

13. Save the file and run the tests again. The casing test passes but the three count tests fail. This is exactly what you expect to happen.



14. Modify the `CountInstancesCorrectly` test method by changing `Assert.NotEqual` to `Assert.Equal`. Save the file. Run the tests again. All tests pass.



Adding a console app

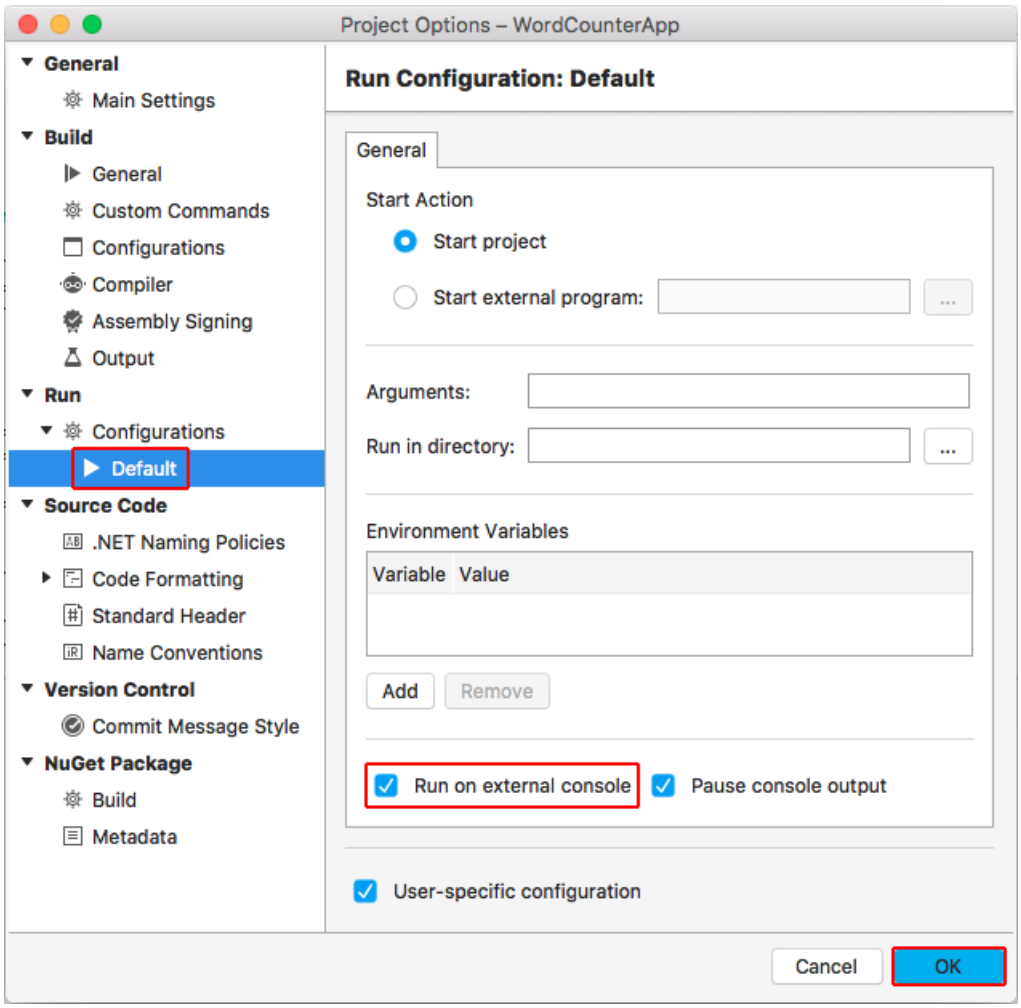
1. In the **Solution** sidebar, right-click the `WordCounter` solution. Add a new **Console Application** project by selecting the template from the **.NET Core** > **App** templates. Select **Next**. Name the project **WordCounterApp**. Select **Create** to create the project in the solution.

2. In the **Solutions** sidebar, right-click the **Dependencies** node of the new **WordCounterApp** project. In the **Edit References** dialog, check **TextUtils** and select **OK**.

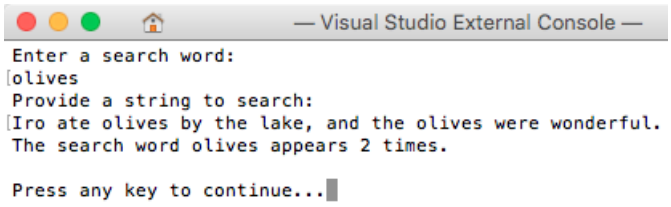
3. Open the *Program.cs* file. Replace the code with the following:

```
using System;
using TextUtils;

namespace WordCounterApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a search word:");
            var searchWord = Console.ReadLine();
            Console.WriteLine("Provide a string to search:");
            var inputString = Console.ReadLine();

            var wordCount = WordCount.GetWordCount(searchWord, inputString);

            var pluralChar = "s";
            if (wordCount == 1)
            {
                pluralChar = string.Empty;
            }

            Console.WriteLine($"The search word {searchWord} appears " +
                              $"{wordCount} time{pluralChar}.");
        }
    }
}
```

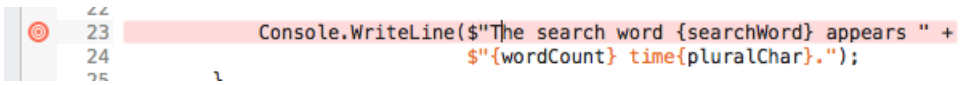
4. To run the app in a console window instead of the IDE, right-click the `WordCounterApp` project, select **Options**, and open the **Default** node under **Configurations**. Check the box for **Run on external console**. Leave the **Pause console output** option checked. This setting causes the app to spawn in a console window so that you can type input for the `Console.ReadLine` statements. If you leave the app to run in the IDE, you can only see the output of `Console.WriteLine` statements. `Console.ReadLine` statements do not work in the IDE's **Application Output** panel.



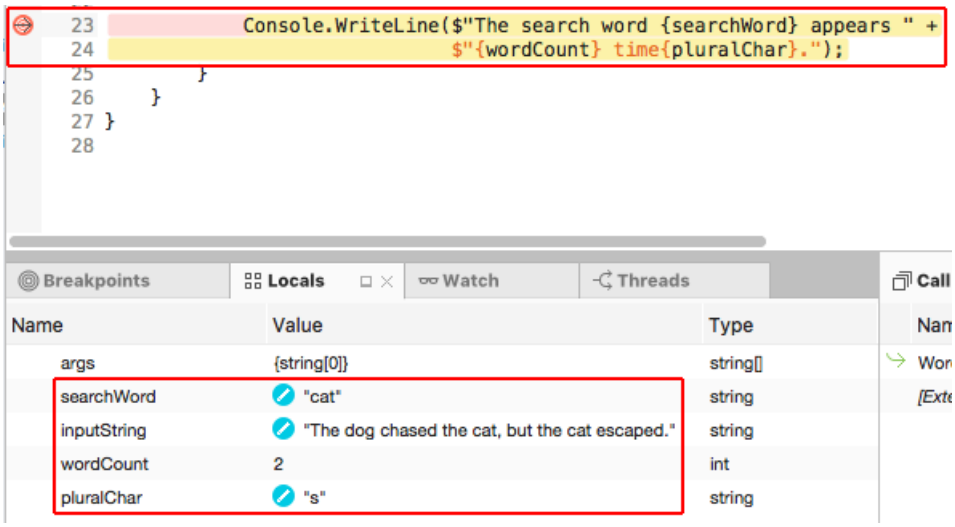
5. Because the current version of Visual Studio for Mac cannot run the tests when the solution is run, you run the console app directly. Right-click on the `WordCounterApp` project and select **Run item** from the context menu. If you attempt to run the app with the Play button, the test runner and app fail to run. For more information on the status of the work on this issue, see xunit/xamarinstudio.xunit (#60). When you run the app, provide values for the search word and input string at the prompts in the console window. The app indicates the number of times the search word appears in the string.



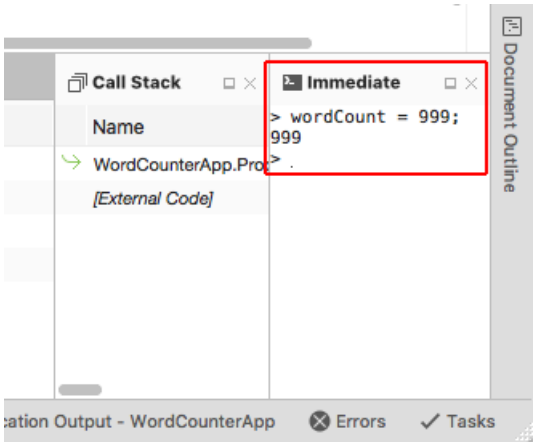
6. The last feature to explore is debugging with Visual Studio for Mac. Set a breakpoint on the `Console.WriteLine` statement: Select in the left margin of line 23, and you see a red circle appear next to the line of code. Alternatively, select anywhere on the line of code and select **Run** > **Toggle Breakpoint** from the menu.



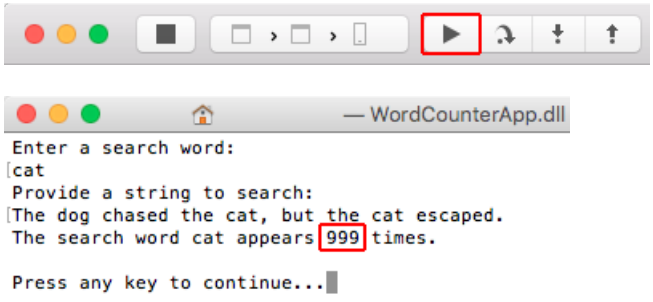
7. Right-click the `WordCounterApp` project. Select **Start Debugging item** from the context menu. When the app runs, enter the search word "cat" and "The dog chased the cat, but the cat escaped." for the string to search. When the `Console.WriteLine` statement is reached, program execution halts before the statement is executed. In the **Locals** tab, you can see the `searchWord`, `inputString`, `wordCount`, and `pluralChar` values.



8. In the **Immediate** pane, type "wordCount = 999;" and press Enter. This assigns a nonsense value of 999 to the `wordCount` variable showing that you can replace variable values while debugging.



9. In the toolbar, click the *continue* arrow. Look at the output in the console window. It reports the incorrect value of 999 that you set when you were debugging the app.



See also

- Visual Studio 2017 for Mac Release Notes

Get started with .NET Core on Windows/Linux/macOS using the command line

8/8/2019 • 5 minutes to read • Edit Online

This topic will show you how to start developing cross-platforms apps in your machine using the .NET Core CLI tools.

If you're unfamiliar with the .NET Core CLI toolset, read the .NET Core SDK overview.

Prerequisites

- .NET Core SDK 2.1.
- A text editor or code editor of your choice.

Hello, Console App!

You can view or download the sample code from the dotnet/samples GitHub repository. For download instructions, see Samples and Tutorials.

Open a command prompt and create a folder named *Hello*. Navigate to the folder you created and type the following:

```
dotnet new console
dotnet run
```

Let's do a quick walkthrough:

1. `dotnet new console`

   `dotnet new` creates an up-to-date `Hello.csproj` project file with the dependencies necessary to build a console app. It also creates a `Program.cs`, a basic file containing the entry point for the application.

   `Hello.csproj`:

   ```
   <Project Sdk="Microsoft.NET.Sdk">

     <PropertyGroup>
       <OutputType>Exe</OutputType>
       <TargetFramework>netcoreapp2.2</TargetFramework>
     </PropertyGroup>

   </Project>
   ```

   The project file specifies everything that's needed to restore dependencies and build the program.

   - The `OutputType` tag specifies that we're building an executable, in other words a console application.
   - The `TargetFramework` tag specifies what .NET implementation we're targeting. In an advanced scenario, you can specify multiple target frameworks and build to all those in a single operation. In this tutorial, we'll stick to building only for .NET Core 2.1.

   `Program.cs`:

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The program starts by `using System`, which means "bring everything in the `System` namespace into scope for this file". The `System` namespace includes basic constructs such as `string`, or numeric types.

We then define a namespace called `Hello`. You can change this to anything you want. A class named `Program` is defined within that namespace, with a `Main` method that takes an array of strings as its argument. This array contains the list of arguments passed in when the compiled program is called. As it is, this array is not used: all the program is doing is to write "Hello World!" to the console. Later, we'll make changes to the code that will make use of this argument.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

`dotnet new` calls `dotnet restore` implicitly. `dotnet restore` calls into NuGet (.NET package manager) to restore the tree of dependencies. NuGet analyzes the *Hello.csproj* file, downloads the dependencies defined in the file (or grabs them from a cache on your machine), and writes the *obj/project.assets.json* file, which is necessary to compile and run the sample.

> **IMPORTANT**
> If you're using a .NET Core 1.x version of the SDK, you'll have to call `dotnet restore` yourself after calling `dotnet new`.

2. `dotnet run`

   `dotnet run` calls `dotnet build` to ensure that the build targets have been built, and then calls `dotnet <assembly.dll>` to run the target application.

```
$ dotnet run
Hello World!
```

Alternatively, you can also execute `dotnet build` to compile the code without running the build console applications. This results in a compiled application as a DLL file that can be run with `dotnet bin\Debug\netcoreapp2.1\Hello.dll` on Windows (use `/` for non-Windows systems). You may also specify arguments to the application as you'll see later on the topic.

```
$ dotnet bin\Debug\netcoreapp2.1\Hello.dll
Hello World!
```

As an advanced scenario, it's possible to build the application as a self-contained set of platform-specific files that can be deployed and run to a machine that doesn't necessarily have .NET Core installed. See .NET Core Application Deployment for details.

Augmenting the program

Let's change the program a bit. Fibonacci numbers are fun, so let's add that in addition to use the argument to greet the person running the app.

1. Replace the contents of your *Program.cs* file with the following code:

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length > 0)
            {
                Console.WriteLine($"Hello {args[0]}!");
            }
            else
            {
                Console.WriteLine("Hello!");
            }

            Console.WriteLine("Fibonacci Numbers 1-15:");

            for (int i = 0; i < 15; i++)
            {
                Console.WriteLine($"{i + 1}: {FibonacciNumber(i)}");
            }
        }

        static int FibonacciNumber(int n)
        {
            int a = 0;
            int b = 1;
            int tmp;

            for (int i = 0; i < n; i++)
            {
                tmp = a;
                a = b;
                b += tmp;
            }

            return a;
        }

    }
}
```

2. Execute `dotnet build` to compile the changes.

3. Run the program passing a parameter to the app:

```
$ dotnet run -- John
Hello John!
Fibonacci Numbers 1-15:
1: 0
2: 1
3: 1
4: 2
5: 3
6: 5
7: 8
8: 13
9: 21
10: 34
11: 55
12: 89
13: 144
14: 233
15: 377
```

And that's it! You can augment `Program.cs` any way you like.

Working with multiple files

Single files are fine for simple one-off programs, but if you're building a more complex app, you're probably going to have multiple source files on your project. Let's build off of the previous Fibonacci example by caching some Fibonacci values and add some recursive features.

1. Add a new file inside the *Hello* directory named *FibonacciGenerator.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace Hello
{
    public class FibonacciGenerator
    {
        private Dictionary<int, int> _cache = new Dictionary<int, int>();

        private int Fib(int n) => n < 2 ? n : FibValue(n - 1) + FibValue(n - 2);

        private int FibValue(int n)
        {
            if (!_cache.ContainsKey(n))
            {
                _cache.Add(n, Fib(n));
            }

            return _cache[n];
        }

        public IEnumerable<int> Generate(int n)
        {
            for (int i = 0; i < n; i++)
            {
                yield return FibValue(i);
            }
        }
    }
}
```

2. Change the `Main` method in your *Program.cs* file to instantiate the new class and call its method as in the following example:

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            var generator = new FibonacciGenerator();
            foreach (var digit in generator.Generate(15))
            {
                Console.WriteLine(digit);
            }
        }
    }
}
```

3. Execute `dotnet build` to compile the changes.

4. Run your app by executing `dotnet run`. The following shows the program output:

```
$ dotnet run
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Publish your app

Once you're ready to distribute your app, use the `dotnet publish` command to generate the *publish* folder at *bin\debug\netcoreapp2.1\publish* (use `/` for non-Windows systems). You can distribute the contents of the *publish* folder to other platforms as long as they've already installed the dotnet runtime.

You can run your published app with the dotnet command:

```
$ dotnet bin\Debug\netcoreapp2.1\publish\Hello.dll
Hello World!
```

Conclusion

And that's it! Now, you can start using the basic concepts learned here to create your own programs.

See also

- Organizing and testing projects with the .NET Core CLI tools
- Publish .NET Core apps with the CLI
- Learn more about app deployment

Organizing and testing projects with the .NET Core command line

5/23/2019 • 6 minutes to read • Edit Online

This tutorial follows Get started with .NET Core on Windows/Linux/macOS using the command line, taking you beyond the creation of a simple console app to develop advanced and well-organized applications. After showing you how to use folders to organize your code, this tutorial shows you how to extend a console application with the xUnit testing framework.

Using folders to organize code

If you want to introduce new types into a console app, you can do so by adding files containing the types to the app. For example if you add files containing `AccountInformation` and `MonthlyReportRecords` types to your project, the project file structure is flat and easy to navigate:

```
/MyProject
|__AccountInformation.cs
|__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

However, this only works well when the size of your project is relatively small. Can you imagine what will happen if you add 20 types to the project? The project definitely wouldn't be easy to navigate and maintain with that many files littering the project's root directory.

To organize the project, create a new folder and name it *Models* to hold the type files. Place the type files into the *Models* folder:

```
/MyProject
|__/Models
   |__AccountInformation.cs
   |__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

Projects that logically group files into folders are easy to navigate and maintain. In the next section, you create a more complex sample with folders and unit testing.

Organizing and testing using the NewTypes Pets Sample

Building the sample

For the following steps, you can either follow along using the NewTypes Pets Sample or create your own files and folders. The types are logically organized into a folder structure that permits the addition of more types later, and tests are also logically placed in folders permitting the addition of more tests later.

The sample contains two types, `Dog` and `Cat`, and has them implement a common interface, `IPet`. For the `NewTypes` project, your goal is to organize the pet-related types into a *Pets* folder. If another set of types is added later, *WildAnimals* for example, they're placed in the *NewTypes* folder alongside the *Pets* folder. The *WildAnimals* folder may contain types for animals that aren't pets, such as `Squirrel` and `Rabbit` types. In this way as types are added, the project remains well organized.

Create the following folder structure with file content indicated:

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
```

IPet.cs:

```
using System;

namespace Pets
{
    public interface IPet
    {
        string TalkToOwner();
    }
}
```

Dog.cs:

```
using System;

namespace Pets
{
    public class Dog : IPet
    {
        public string TalkToOwner() => "Woof!";
    }
}
```

Cat.cs:

```
using System;

namespace Pets
{
    public class Cat : IPet
    {
        public string TalkToOwner() => "Meow!";
    }
}
```

Program.cs:

```
using System;
using Pets;
using System.Collections.Generic;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<IPet> pets = new List<IPet>
            {
                new Dog(),
                new Cat()
            };

            foreach (var pet in pets)
            {
                Console.WriteLine(pet.TalkToOwner());
            }
        }
    }
}
```

NewTypes.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

</Project>
```

Execute the following command:

```
dotnet run
```

Obtain the following output:

```
Woof!
Meow!
```

Optional exercise: You can add a new pet type, such as a `Bird`, by extending this project. Make the bird's `TalkToOwner` method give a `Tweet!` to the owner. Run the app again. The output will include `Tweet!`

Testing the sample

The `NewTypes` project is in place, and you've organized it by keeping the pets-related types in a folder. Next, create your test project and start writing tests with the xUnit test framework. Unit testing allows you to automatically check the behavior of your pet types to confirm that they're operating properly.

Navigate back to the *src* folder and create a *test* folder with a *NewTypesTests* folder within it. At a command prompt from the *NewTypesTests* folder, execute `dotnet new xunit`. This produces two files: *NewTypesTests.csproj* and *UnitTest1.cs*.

The test project cannot currently test the types in `NewTypes` and requires a project reference to the `NewTypes` project. To add a project reference, use the `dotnet add reference` command:

```
dotnet add reference ../../src/NewTypes/NewTypes.csproj
```

Or, you also have the option of manually adding the project reference by adding an `<ItemGroup>` node to the *NewTypesTests.csproj* file:

```
<ItemGroup>
  <ProjectReference Include="../../src/NewTypes/NewTypes.csproj" />
</ItemGroup>
```

NewTypesTests.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.9.0" />
    <PackageReference Include="xunit" Version="2.4.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="../../src/NewTypes/NewTypes.csproj"/>
  </ItemGroup>

</Project>
```

The *NewTypesTests.csproj* file contains the following:

- Package reference to `Microsoft.NET.Test.Sdk`, the .NET testing infrastructure
- Package reference to `xunit`, the xUnit testing framework
- Package reference to `xunit.runner.visualstudio`, the test runner
- Project reference to `NewTypes`, the code to test

Change the name of *UnitTest1.cs* to *PetTests.cs* and replace the code in the file with the following:

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }
}
```

Optional exercise: If you added a `Bird` type earlier that yields a `Tweet!` to the owner, add a test method to the *PetTests.cs* file, `BirdTalkToOwnerReturnsTweet`, to check that the `TalkToOwner` method works correctly for the `Bird` type.

> **NOTE**
> Although you expect that the `expected` and `actual` values are equal, an initial assertion with the `Assert.NotEqual` check specifies that these values are *not equal*. Always initially create a test to fail in order to check the logic of the test. After you confirm that the test fails, adjust the assertion to allow the test to pass.

The following shows the complete project structure:

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
|__/test
   |__NewTypesTests
      |__PetTests.cs
      |__NewTypesTests.csproj
```

Start in the *test/NewTypesTests* directory. Restore the test project with the `dotnet restore` command. Run the tests with the `dotnet test` command. This command starts the test runner specified in the project file.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

As expected, testing fails, and the console displays the following output:

```
Test run for c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:00.77]     PetTests.DogTalkToOwnerReturnsWoof [FAIL]
[xUnit.net 00:00:00.78]     PetTests.CatTalkToOwnerReturnsMeow [FAIL]
Failed   PetTests.DogTalkToOwnerReturnsWoof
Error Message:
 Assert.NotEqual() Failure
Expected: Not "Woof!"
Actual:   "Woof!"
Stack Trace:
   at PetTests.DogTalkToOwnerReturnsWoof() in c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 13
Failed   PetTests.CatTalkToOwnerReturnsMeow
Error Message:
 Assert.NotEqual() Failure
Expected: Not "Meow!"
Actual:   "Meow!"
Stack Trace:
   at PetTests.CatTalkToOwnerReturnsMeow() in c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 22

Total tests: 2. Passed: 0. Failed: 2. Skipped: 0.
Test Run Failed.
Test execution time: 1.7000 Seconds
```

Change the assertions of your tests from `Assert.NotEqual` to `Assert.Equal`:

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.Equal(expected, actual);
    }
}
```

Re-run the tests with the `dotnet test` command and obtain the following output:

```
Test run for c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.6029 Seconds
```

Testing passes. The pet types' methods return the correct values when talking to the owner.

You've learned techniques for organizing and testing projects using xUnit. Go forward with these techniques applying them in your own projects. *Happy coding!*

Developing Libraries with Cross Platform Tools

3/1/2019 • 11 minutes to read • Edit Online

This article covers how to write libraries for .NET using cross-platform CLI tools. The CLI provides an efficient and low-level experience that works across any supported OS. You can still build libraries with Visual Studio, and if that is your preferred experience refer to the Visual Studio guide.

Prerequisites

You need the .NET Core SDK and CLI installed on your machine.

For the sections of this document dealing with .NET Framework versions, you need the .NET Framework installed on a Windows machine.

Additionally, if you wish to support older .NET Framework targets, you need to install targeting/developer packs for older framework versions from the .NET download archives page. Refer to this table:

.NET FRAMEWORK VERSION	WHAT TO DOWNLOAD
4.6.1	.NET Framework 4.6.1 Targeting Pack
4.6	.NET Framework 4.6 Targeting Pack
4.5.2	.NET Framework 4.5.2 Developer Pack
4.5.1	.NET Framework 4.5.1 Developer Pack
4.5	Windows Software Development Kit for Windows 8
4.0	Windows SDK for Windows 7 and .NET Framework 4
2.0, 3.0, and 3.5	.NET Framework 3.5 SP1 Runtime (or Windows 8+ version)

How to target the .NET Standard

If you're not quite familiar with the .NET Standard, refer to the .NET Standard to learn more.

In that article, there is a table which maps .NET Standard versions to various implementations:

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1

1 The versions listed for .NET Framework apply to .NET Core 2.0 SDK and later versions of the tooling. Older versions used a different mapping for .NET Standard 1.5 and higher. You can download tooling for .NET Core tools for Visual Studio 2015 if you cannot upgrade to Visual Studio 2017.

2 The versions listed here represent the rules that NuGet uses to determine whether a given .NET Standard library is applicable. While NuGet considers .NET Framework 4.6.1 as supporting .NET Standard 1.5 through 2.0, there are several issues with consuming .NET Standard libraries that were built for those versions from .NET Framework 4.6.1 projects. For .NET Framework projects that need to use such libraries, we recommend that you upgrade the project to target .NET Framework 4.7.2 or higher.

- The columns represent .NET Standard versions. Each header cell is a link to a document that shows which APIs got added in that version of .NET Standard.
- The rows represent the different .NET implementations.
- The version number in each cell indicates the *minimum* version of the implementation you'll need in order to target that .NET Standard version.
- For an interactive table, see .NET Standard versions.

Here's what this table means for the purposes of creating a library:

The version of the .NET Standard you pick will be a tradeoff between access to the newest APIs and the ability to target more .NET implementations and .NET Standard versions. You control the range of targetable platforms and versions by picking a version of `netstandardX.X` (Where `X.X` is a version number) and adding it to your project file (`.csproj` or `.fsproj`).

You have three primary options when targeting the .NET Standard, depending on your needs.

1. You can use the default version of the .NET Standard supplied by templates - `netstandard1.4` - which gives you access to most APIs on .NET Standard while still being compatible with UWP, .NET Framework 4.6.1, and the forthcoming .NET Standard 2.0.

   ```
   <Project Sdk="Microsoft.NET.Sdk">
     <PropertyGroup>
       <TargetFramework>netstandard1.4</TargetFramework>
     </PropertyGroup>
   </Project>
   ```

2. You can use a lower or higher version of the .NET Standard by modifying the value in the `TargetFramework` node of your project file.

   .NET Standard versions are backward compatible. That means that `netstandard1.0` libraries run on

`netstandard1.1` platforms and higher. However, there is no forward compatibility - lower .NET Standard platforms cannot reference higher ones. This means that `netstandard1.0` libraries cannot reference libraries targeting `netstandard1.1` or higher. Select the Standard version that has the right mix of APIs and platform support for your needs. We recommend `netstandard1.4` for now.

3. If you want to target the .NET Framework versions 4.0 or below, or you wish to use an API available in the .NET Framework but not in the .NET Standard (for example, `System.Drawing`), read the following sections and learn how to multitarget.

How to target the .NET Framework

> **NOTE**
> These instructions assume you have the .NET Framework installed on your machine. Refer to the Prerequisites to get dependencies installed.

Keep in mind that some of the .NET Framework versions used here are no longer in support. Refer to the .NET Framework Support Lifecycle Policy FAQ about unsupported versions.

If you want to reach the maximum number of developers and projects, use the .NET Framework 4.0 as your baseline target. To target the .NET Framework, you will need to begin by using the correct Target Framework Moniker (TFM) that corresponds to the .NET Framework version you wish to support.

```
.NET Framework 2.0   --> net20
.NET Framework 3.0   --> net30
.NET Framework 3.5   --> net35
.NET Framework 4.0   --> net40
.NET Framework 4.5   --> net45
.NET Framework 4.5.1 --> net451
.NET Framework 4.5.2 --> net452
.NET Framework 4.6   --> net46
.NET Framework 4.6.1 --> net461
.NET Framework 4.6.2 --> net462
.NET Framework 4.7   --> net47
```

You then insert this TFM into the `TargetFramework` section of your project file. For example, here's how you would write a library which targets the .NET Framework 4.0:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net40</TargetFramework>
  </PropertyGroup>
</Project>
```

And that's it! Although this compiled only for the .NET Framework 4, you can use the library on newer versions of the .NET Framework.

How to Multitarget

> **NOTE**
> The following instructions assume you have the .NET Framework installed on your machine. Refer to the Prerequisites section to learn which dependencies you need to install and where to download them from.

You may need to target older versions of the .NET Framework when your project supports both the .NET Framework and .NET Core. In this scenario, if you want to use newer APIs and language constructs for the newer targets, use `#if` directives in your code. You also might need to add different packages and dependencies for each platform you're targeting to include the different APIs needed for each case.

For example, let's say you have a library that performs networking operations over HTTP. For .NET Standard and the .NET Framework versions 4.5 or higher, you can use the `HttpClient` class from the `System.Net.Http` namespace. However, earlier versions of the .NET Framework don't have the `HttpClient` class, so you could use the `WebClient` class from the `System.Net` namespace for those instead.

Your project file could look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.0 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net40'">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.5 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net45'">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>
</Project>
```

You'll notice three major changes here:

1. The `TargetFramework` node has been replaced by `TargetFrameworks`, and three TFMs are expressed inside.
2. There is an `<ItemGroup>` node for the `net40` target pulling in one .NET Framework reference.
3. There is an `<ItemGroup>` node for the `net45` target pulling in two .NET Framework references.

The build system is aware of the following preprocessor symbols used in `#if` directives:

TARGET FRAMEWORKS	SYMBOLS
.NET Framework	`NETFRAMEWORK`, `NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`, `NET48`
.NET Standard	`NETSTANDARD`, `NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`

Here is an example making use of conditional compilation per-target:

```
using System;
using System.Text.RegularExpressions;
#if NET40
// This only compiles for the .NET Framework 4 targets
using System.Net;
#else
 // This compiles for all other targets
using System.Net.Http;
using System.Threading.Tasks;
#endif

namespace MultitargetLib
{
    public class Library
    {
#if NET40
        private readonly WebClient _client = new WebClient();
        private readonly object _locker = new object();
#else
        private readonly HttpClient _client = new HttpClient();
#endif

#if NET40
        // .NET Framework 4.0 does not have async/await
        public string GetDotNetCount()
        {
            string url = "https://www.dotnetfoundation.org/";

            var uri = new Uri(url);

            string result = "";

            // Lock here to provide thread-safety.
            lock(_locker)
            {
                result = _client.DownloadString(uri);
            }

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"Dotnet Foundation mentions .NET {dotNetCount} times!";
        }
#else
        // .NET 4.5+ can use async/await!
        public async Task<string> GetDotNetCountAsync()
        {
            string url = "https://www.dotnetfoundation.org/";

            // HttpClient is thread-safe, so no need to explicitly lock here
            var result = await _client.GetStringAsync(url);

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"dotnetfoundation.org mentions .NET {dotNetCount} times in its HTML!";
        }
#endif
    }
}
```

If you build this project with `dotnet build`, you'll notice three directories under the `bin/` folder:

```
net40/
net45/
netstandard1.4/
```

Each of these contain the `.dll` files for each target.

How to test libraries on .NET Core

It's important to be able to test across platforms. You can use either xUnit or MSTest out of the box. Both are perfectly suitable for unit testing your library on .NET Core. How you set up your solution with test projects will depend on the structure of your solution. The following example assumes that the test and source directories live in the same top-level directory.

> **NOTE**
> This uses some .NET Core CLI commands. See dotnet new and dotnet sln for more information.

1. Set up your solution. You can do so with the following commands:

   ```
   mkdir SolutionWithSrcAndTest
   cd SolutionWithSrcAndTest
   dotnet new sln
   dotnet new classlib -o MyProject
   dotnet new xunit -o MyProject.Test
   dotnet sln add MyProject/MyProject.csproj
   dotnet sln add MyProject.Test/MyProject.Test.csproj
   ```

   This will create projects and link them together in a solution. Your directory for `SolutionWithSrcAndTest` should look like this:

   ```
   /SolutionWithSrcAndTest
   |__SolutionWithSrcAndTest.sln
   |__MyProject/
   |__MyProject.Test/
   ```

2. Navigate to the test project's directory and add a reference to `MyProject.Test` from `MyProject`.

   ```
   cd MyProject.Test
   dotnet add reference ../MyProject/MyProject.csproj
   ```

3. Restore packages and build projects:

   ```
   dotnet restore
   dotnet build
   ```

   > **NOTE**
   > Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

4. Verify that xUnit runs by executing the `dotnet test` command. If you chose to use MSTest, then the MSTest console runner should run instead.

And that's it! You can now test your library across all platforms using command line tools. To continue testing now that you have everything set up, testing your library is very simple:

1. Make changes to your library.
2. Run tests from the command line, in your test directory, with `dotnet test` command.

Your code will be automatically rebuilt when you invoke `dotnet test` command.

How to use multiple projects

A common need for larger libraries is to place functionality in different projects.

Imagine you wished to build a library which could be consumed in idiomatic C# and F#. That would mean that consumers of your library consume them in ways which are natural to C# or F#. For example, in C# you might consume the library like this:

```
using AwesomeLibrary.CSharp;

public Task DoThings(Data data)
{
    var convertResult = await AwesomeLibrary.ConvertAsync(data);
    var result = AwesomeLibrary.Process(convertResult);
    // do something with result
}
```

In F#, it might look like this:

```
open AwesomeLibrary.FSharp

let doWork data = async {
    let! result = AwesomeLibrary.AsyncConvert data // Uses an F# async function rather than C# async method
    // do something with result
}
```

Consumption scenarios like this mean that the APIs being accessed have to have a different structure for C# and F#. A common approach to accomplishing this is to factor all of the logic of a library into a core project, with C# and F# projects defining the API layers that call into that core project. The rest of the section will use the following names:

- **AwesomeLibrary.Core** - A core project which contains all logic for the library
- **AwesomeLibrary.CSharp** - A project with public APIs intended for consumption in C#
- **AwesomeLibrary.FSharp** - A project with public APIs intended for consumption in F#

You can run the following commands in your terminal to produce the same structure as this guide:

```
mkdir AwesomeLibrary && cd AwesomeLibrary
dotnet new sln
mkdir AwesomeLibrary.Core && cd AwesomeLibrary.Core && dotnet new classlib
cd ..
mkdir AwesomeLibrary.CSharp && cd AwesomeLibrary.CSharp && dotnet new classlib
cd ..
mkdir AwesomeLibrary.FSharp && cd AwesomeLibrary.FSharp && dotnet new classlib -lang F#
cd ..
dotnet sln add AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
dotnet sln add AwesomeLibrary.CSharp/AwesomeLibrary.CSharp.csproj
dotnet sln add AwesomeLibrary.FSharp/AwesomeLibrary.FSharp.fsproj
```

This will add the three projects above and a solution file which links them together. Creating the solution file and linking projects will allow you to restore and build projects from a top-level.

Project-to-project referencing

The best way to reference a project is to use the .NET Core CLI to add a project reference. From the **AwesomeLibrary.CSharp** and **AwesomeLibrary.FSharp** project directories, you can run the following command:

```
dotnet add reference ../AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
```

The project files for both **AwesomeLibrary.CSharp** and **AwesomeLibrary.FSharp** will now reference **AwesomeLibrary.Core** as a `ProjectReference` target. You can verify this by inspecting the project files and seeing the following in them:

```
<ItemGroup>
  <ProjectReference Include="..\AwesomeLibrary.Core\AwesomeLibrary.Core.csproj" />
</ItemGroup>
```

You can add this section to each project file manually if you prefer not to use the .NET Core CLI.

Structuring a solution

Another important aspect of multi-project solutions is establishing a good overall project structure. You can organize code however you like, and as long as you link each project to your solution file with `dotnet sln add`, you will be able to run `dotnet restore` and `dotnet build` at the solution level.

Get started with ASP.NET Core

3/12/2019 • 2 minutes to read • Edit Online

For tutorials about developing ASP.NET Core web applications, see ASP.NET Core Tutorials.

Write a custom .NET Core host to control the .NET runtime from your native code

8/20/2019 • 21 minutes to read • Edit Online

Like all managed code, .NET Core applications are executed by a host. The host is responsible for starting the runtime (including components like the JIT and garbage collector) and invoking managed entry points.

Hosting the .NET Core runtime is an advanced scenario and, in most cases, .NET Core developers don't need to worry about hosting because .NET Core build processes provide a default host to run .NET Core applications. In some specialized circumstances, though, it can be useful to explicitly host the .NET Core runtime, either as a means of invoking managed code in a native process or in order to gain more control over how the runtime works.

This article gives an overview of the steps necessary to start the .NET Core runtime from native code and execute managed code in it.

Prerequisites

Because hosts are native applications, this tutorial will cover constructing a C++ application to host .NET Core. You will need a C++ development environment (such as that provided by Visual Studio).

You will also want a simple .NET Core application to test the host with, so you should install the .NET Core SDK and build a small .NET Core test app (such as a 'Hello World' app). The 'Hello World' app created by the new .NET Core console project template is sufficient.

Hosting APIs

There are three different APIs that can be used to host .NET Core. This document (and its associated samples) cover all options.

- The preferred method of hosting the .NET Core runtime in .NET Core 3.0 and above is with the `nethost` and `hostfxr` libraries' APIs. These entry points handle the complexity of finding and setting up the runtime for initialization and allow both launching a managed application and calling into a static managed method.
- The preferred method of hosting the .NET Core runtime prior to .NET Core 3.0 is with the CoreClrHost.h API. This API exposes functions for easily starting and stopping the runtime and invoking managed code (either by launching a managed exe or by calling static managed methods).
- .NET Core can also be hosted with the `ICLRRuntimeHost4` interface in mscoree.h. This API has been around longer than CoreClrHost.h, so you may have seen older hosts using it. It still works and allows a bit more control over the hosting process than CoreClrHost. For most scenarios, though, CoreClrHost.h is preferred now because of its simpler APIs.

Sample Hosts

Sample hosts demonstrating the steps outlined in the tutorials below are available in the dotnet/samples GitHub repository. Comments in the samples clearly associate the numbered steps from these tutorials with where they're performed in the sample. For download instructions, see Samples and Tutorials.

Keep in mind that the sample hosts are meant to be used for learning purposes, so they are light on error checking and are designed to emphasize readability over efficiency.

Create a host using NetHost.h and HostFxr.h

The following steps detail how to use the `nethost` and `hostfxr` libraries to start the .NET Core runtime in a native application and call into a managed static method. The sample uses the `nethost` header and library installed with the .NET SDK and copies of the `coreclr_delegates.h` and `hostfxr.h` files from the dotnet/core-setup repository.

Step 1 - Load HostFxr and get exported hosting functions

The `nethost` library provides the `get_hostfxr_path` function for locating the `hostfxr` library. The `hostfxr` library exposes functions for hosting the .NET Core runtime. The full list of functions can be found in `hostfxr.h` and the native hosting design document. The sample and this tutorial use the following:

- `hostfxr_initialize_for_runtime_config`: Initializes a host context and prepares for initialization of the .NET Core runtime using the specified runtime configuration.
- `hostfxr_get_runtime_delegate`: Gets a delegate for runtime functionality.
- `hostfxr_close`: Closes a host context.

The `hostfxr` library is found using `get_hostfxr_path`. It is then loaded and its exports are retrieved.

```
// Using the nethost library, discover the location of hostfxr and get exports
bool load_hostfxr()
{
    // Pre-allocate a large buffer for the path to hostfxr
    char_t buffer[MAX_PATH];
    size_t buffer_size = sizeof(buffer) / sizeof(char_t);
    int rc = get_hostfxr_path(buffer, &buffer_size, nullptr);
    if (rc != 0)
        return false;

    // Load hostfxr and get desired exports
    void *lib = load_library(buffer);
    init_fptr = (hostfxr_initialize_for_runtime_config_fn)get_export(lib,
"hostfxr_initialize_for_runtime_config");
    get_delegate_fptr = (hostfxr_get_runtime_delegate_fn)get_export(lib, "hostfxr_get_runtime_delegate");
    close_fptr = (hostfxr_close_fn)get_export(lib, "hostfxr_close");

    return (init_fptr && get_delegate_fptr && close_fptr);
}
```

Step 2 - Initialize and start the .NET Core runtime

The `hostfxr_initialize_for_runtime_config` and `hostfxr_get_runtime_delegate` functions initialize and start the .NET Core runtime using the runtime configuration for the managed component that will be loaded. The `hostfxr_get_runtime_delegate` function is used to get a runtime delegate that allows loading a managed assembly and getting a function pointer to a static method in that assembly.

```
// Load and initialize .NET Core and get desired function pointer for scenario
load_assembly_and_get_function_pointer_fn get_dotnet_load_assembly(const char_t *config_path)
{
    // Load .NET Core
    void *load_assembly_and_get_function_pointer = nullptr;
    hostfxr_handle cxt = nullptr;
    int rc = init_fptr(config_path, nullptr, &cxt);
    if (rc != 0 || cxt == nullptr)
    {
        std::cerr << "Init failed: " << std::hex << std::showbase << rc << std::endl;
        close_fptr(cxt);
        return nullptr;
    }

    // Get the load assembly function pointer
    rc = get_delegate_fptr(
        cxt,
        hdt_load_assembly_and_get_function_pointer,
        &load_assembly_and_get_function_pointer);
    if (rc != 0 || load_assembly_and_get_function_pointer == nullptr)
        std::cerr << "Get delegate failed: " << std::hex << std::showbase << rc << std::endl;

    close_fptr(cxt);
    return (load_assembly_and_get_function_pointer_fn)load_assembly_and_get_function_pointer;
}
```

Step 3 - Load managed assembly and get function pointer to a managed method

The runtime delegate is called to load the managed assembly and get a function pointer to a managed method. The delegate requires the assembly path, type name, and method name as inputs and returns a function pointer that can be used to invoke the managed method.

```
// Function pointer to managed delegate
component_entry_point_fn hello = nullptr;
int rc = load_assembly_and_get_function_pointer(
    dotnetlib_path.c_str(),
    dotnet_type,
    dotnet_type_method,
    nullptr /*delegate_type_name*/,
    nullptr,
    (void**)&hello);
```

By passing `nullptr` as the delegate type name when calling the runtime delegate, the sample uses a default signature for the managed method:

```
public delegate int ComponentEntryPoint(IntPtr args, int sizeBytes);
```

A different signature can be used by specifying the delegate type name when calling the runtime delegate.

Step 4 - Run managed code!

The native host can now call the managed method and pass it the desired parameters.

```
lib_args args
{
    STR("from host!"),
    i
};

hello(&args, sizeof(args));
```

Create a host using CoreClrHost.h

The following steps detail how to use the CoreClrHost.h API to start the .NET Core runtime in a native application and call into a managed static method. The code snippets in this document use some Windows-specific APIs, but the full sample host shows both Windows and Linux code paths.

The Unix CoreRun host shows a more complex, real-world example of hosting using coreclrhost.h.

Step 1 - Find and load CoreCLR

The .NET Core runtime APIs are in *coreclr.dll* (on Windows), in *libcoreclr.so* (on Linux), or in *libcoreclr.dylib* (on macOS). The first step to hosting .NET Core is to load the CoreCLR library. Some hosts probe different paths or use input parameters to find the library while others know to load it from a certain path (next to the host, for example, or from a machine-wide location).

Once found, the library is loaded with `LoadLibraryEx` (on Windows) or `dlopen` (on Linux/Mac).

```
HMODULE coreClr = LoadLibraryExA(coreClrPath.c_str(), NULL, 0);
```

Step 2 - Get .NET Core hosting functions

CoreClrHost has several important methods useful for hosting .NET Core:

- `coreclr_initialize`: Starts the .NET Core runtime and sets up the default (and only) AppDomain.
- `coreclr_execute_assembly`: Executes a managed assembly.
- `coreclr_create_delegate`: Creates a function pointer to a managed method.
- `coreclr_shutdown`: Shuts down the .NET Core runtime.
- `coreclr_shutdown_2`: Like `coreclr_shutdown`, but also retrieves the managed code's exit code.

After loading the CoreCLR library, the next step is to get references to these functions using `GetProcAddress` (on Windows) or `dlsym` (on Linux/Mac).

```
coreclr_initialize_ptr initializeCoreClr = (coreclr_initialize_ptr)GetProcAddress(coreClr,
"coreclr_initialize");
coreclr_create_delegate_ptr createManagedDelegate = (coreclr_create_delegate_ptr)GetProcAddress(coreClr,
"coreclr_create_delegate");
coreclr_shutdown_ptr shutdownCoreClr = (coreclr_shutdown_ptr)GetProcAddress(coreClr, "coreclr_shutdown");
```

Step 3 - Prepare runtime properties

Before starting the runtime, it is necessary to prepare some properties to specify behavior (especially concerning the assembly loader).

Common properties include:

- `TRUSTED_PLATFORM_ASSEMBLIES` This is a list of assembly paths (delimited by ';' on Windows and ':' on Linux) which the runtime will be able to resolve by default. Some hosts have hard-coded manifests listing assemblies they can load. Others will put any library in certain locations (next to *coreclr.dll*, for example) on this list.
- `APP_PATHS` This is a list of paths to probe in for an assembly if it can't be found in the trusted platform assemblies (TPA) list. Because the host has more control over which assemblies are loaded using the TPA list, it is a best practice for hosts to determine which assemblies they expect to load and list them explicitly. If probing at runtime is needed, however, this property can enable that scenario.
- `APP_NI_PATHS` This list is similar to APP_PATHS except that it's meant to be paths that will be probed for native images.
- `NATIVE_DLL_SEARCH_DIRECTORIES` This property is a list of paths the loader should probe when looking for native libraries called via p/invoke.

- `PLATFORM_RESOURCE_ROOTS` This list includes paths to probe in for resource satellite assemblies (in culture-specific sub-directories).

In this sample host, the TPA list is constructed by simply listing all libraries in the current directory:

```
void BuildTpaList(const char* directory, const char* extension, std::string& tpaList)
{
    // This will add all files with a .dll extension to the TPA list.
    // This will include unmanaged assemblies (coreclr.dll, for example) that don't
    // belong on the TPA list. In a real host, only managed assemblies that the host
    // expects to load should be included. Having extra unmanaged assemblies doesn't
    // cause anything to fail, though, so this function just enumerates all dll's in
    // order to keep this sample concise.
    std::string searchPath(directory);
    searchPath.append(FS_SEPARATOR);
    searchPath.append("*");
    searchPath.append(extension);

    WIN32_FIND_DATAA findData;
    HANDLE fileHandle = FindFirstFileA(searchPath.c_str(), &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Append the assembly to the list
            tpaList.append(directory);
            tpaList.append(FS_SEPARATOR);
            tpaList.append(findData.cFileName);
            tpaList.append(PATH_DELIMITER);

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
path,
            // and have no native images, we can ignore that complication.
        }
        while (FindNextFileA(fileHandle, &findData));
        FindClose(fileHandle);
    }
}
```

Because the sample is simple, it only needs the `TRUSTED_PLATFORM_ASSEMBLIES` property:

```
// Define CoreCLR properties
// Other properties related to assembly loading are common here,
// but for this simple sample, TRUSTED_PLATFORM_ASSEMBLIES is all
// that is needed. Check hosting documentation for other common properties.
const char* propertyKeys[] = {
    "TRUSTED_PLATFORM_ASSEMBLIES"      // Trusted assemblies
};

const char* propertyValues[] = {
    tpaList.c_str()
};
```

Step 4 - Start the runtime

Unlike the mscoree.h hosting API (described below), CoreCLRHost.h APIs start the runtime and create the default

AppDomain all with a single call. The `coreclr_initialize` function takes a base path, name, and the properties described earlier and returns back a handle to the host via the `hostHandle` parameter.

```
void* hostHandle;
unsigned int domainId;

// This function both starts the .NET Core runtime and creates
// the default (and only) AppDomain
int hr = initializeCoreClr(
                runtimePath,        // App base path
                "SampleHost",       // AppDomain friendly name
                sizeof(propertyKeys) / sizeof(char*),   // Property count
                propertyKeys,       // Property names
                propertyValues,     // Property values
                &hostHandle,        // Host handle
                &domainId);         // AppDomain ID
```

Step 5 - Run managed code!

With the runtime started, the host can call managed code. This can be done in a couple of different ways. The sample code linked to this tutorial uses the `coreclr_create_delegate` function to create a delegate to a static managed method. This API takes the assembly name, namespace-qualified type name, and method name as inputs and returns a delegate that can be used to invoke the method.

```
doWork_ptr managedDelegate;

// The assembly name passed in the third parameter is a managed assembly name
// as described at https://docs.microsoft.com/dotnet/framework/app-domains/assembly-names
hr = createManagedDelegate(
        hostHandle,
        domainId,
        "ManagedLibrary, Version=1.0.0.0",
        "ManagedLibrary.ManagedWorker",
        "DoWork",
        (void**)&managedDelegate);
```

In this sample, the host can now call `managedDelegate` to run the `ManagedWorker.DoWork` method.

Alternatively, the `coreclr_execute_assembly` function can be used to launch a managed executable. This API takes an assembly path and array of arguments as input parameters. It loads the assembly at that path and invokes its main method.

```
int hr = executeAssembly(
        hostHandle,
        domainId,
        argumentCount,
        arguments,
        "HelloWorld.exe",
        (unsigned int*)&exitCode);
```

Step 6 - Shutdown and clean up

Finally, when the host is done running managed code, the .NET Core runtime is shut down with `coreclr_shutdown` or `coreclr_shutdown_2`.

```
hr = shutdownCoreClr(hostHandle, domainId);
```

CoreCLR does not support reinitialization or unloading. Do not call `coreclr_initialize` again or unload the CoreCLR library.

Create a host using Mscoree.h

As mentioned previously, CoreClrHost.h is now the preferred method of hosting the .NET Core runtime. The `ICLRRuntimeHost4` interface can still be used, though, if the CoreClrHost.h interfaces aren't sufficient (if non-standard startup flags are needed, for example, or if an AppDomainManager is needed on the default domain). These instructions will guide you through hosting .NET Core using mscoree.h.

The CoreRun host shows a more complex, real-world example of hosting using mscoree.h.

A note about mscoree.h

The `ICLRRuntimeHost4` .NET Core hosting interface is defined in MSCOREE.IDL. A header version of this file (mscoree.h), which your host will need to reference, is produced via MIDL when the .NET Core runtime is built. If you do not want to build the .NET Core runtime, mscoree.h is also available as a pre-built header in the dotnet/coreclr repository. Instructions on building the .NET Core runtime can be found in its GitHub repository.

Step 1 - Identify the managed entry point

After referencing necessary headers (mscoree.h and stdio.h, for example), one of the first things a .NET Core host must do is locate the managed entry point it will be using. In our sample host, this is done by just taking the first command line argument to our host as the path to a managed binary whose `main` method will be executed.

```
// The managed application to run should be the first command-line parameter.
// Subsequent command line parameters will be passed to the managed app later in this host.
wchar_t targetApp[MAX_PATH];
GetFullPathNameW(argv[1], MAX_PATH, targetApp, NULL);
```

Step 2 - Find and load CoreCLR

The .NET Core runtime APIs are in *CoreCLR.dll* (on Windows). To get our hosting interface (`ICLRRuntimeHost4`), it's necessary to find and load *CoreCLR.dll*. It is up to the host to define a convention for how it will locate *CoreCLR.dll*. Some hosts expect the file to be present in a well-known machine-wide location (such as *%programfiles%\dotnet\shared\Microsoft.NETCore.App\2.1.6*). Others expect that *CoreCLR.dll* will be loaded from a location next to either the host itself or the app to be hosted. Still others might consult an environment variable to find the library.

On Linux or Mac, the core runtime library is *libcoreclr.so* or *libcoreclr.dylib*, respectively.

Our sample host probes a few common locations for *CoreCLR.dll*. Once found, it must be loaded via `LoadLibrary` (or `dlopen` on Linux/Mac).

```
HMODULE ret = LoadLibraryExW(coreDllPath, NULL, 0);
```

Step 3 - Get an ICLRRuntimeHost4 Instance

The `ICLRRuntimeHost4` hosting interface is retrieved by calling `GetProcAddress` (or `dlsym` on Linux/Mac) on `GetCLRRuntimeHost`, and then invoking that function.

```
ICLRRuntimeHost4* runtimeHost;

FnGetCLRRuntimeHost pfnGetCLRRuntimeHost =
    (FnGetCLRRuntimeHost)::GetProcAddress(coreCLRModule, "GetCLRRuntimeHost");

if (!pfnGetCLRRuntimeHost)
{
    printf("ERROR - GetCLRRuntimeHost not found");
    return -1;
}

// Get the hosting interface
HRESULT hr = pfnGetCLRRuntimeHost(IID_ICLRRuntimeHost4, (IUnknown**)&runtimeHost);
```

Step 4 - Set startup flags and start the runtime

With an `ICLRRuntimeHost4` in-hand, we can now specify runtime-wide startup flags and start the runtime. Startup flags determine which garbage collector (GC) to use (concurrent or server), whether we will use a single AppDomain or multiple AppDomains, and what loader optimization policy to use (for domain-neutral loading of assemblies).

```
hr = runtimeHost->SetStartupFlags(
    // These startup flags control runtime-wide behaviors.
    // A complete list of STARTUP_FLAGS can be found in mscoree.h,
    // but some of the more common ones are listed below.
    static_cast<STARTUP_FLAGS>(
        // STARTUP_FLAGS::STARTUP_SERVER_GC |        // Use server GC
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN |  // Maximize domain-neutral loading
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST | // Domain-neutral loading for
strongly-named assemblies
        STARTUP_FLAGS::STARTUP_CONCURRENT_GC |      // Use concurrent GC
        STARTUP_FLAGS::STARTUP_SINGLE_APPDOMAIN |     // All code executes in the default AppDomain
                                                      // (required to use the runtimeHost-
>ExecuteAssembly helper function)
        STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN // Prevents domain-neutral loading
    )
);
```

The runtime is started with a call to the `Start` function.

```
hr = runtimeHost->Start();
```

Step 5 - Preparing AppDomain settings

Once the runtime is started, we will want to set up an AppDomain. There are a number of options that must be specified when creating a .NET AppDomain, however, so it's necessary to prepare those first.

AppDomain flags specify AppDomain behaviors related to security and interop. Older Silverlight hosts used these settings to sandbox user code, but most modern .NET Core hosts run user code as full trust and enable interop.

```
int appDomainFlags =
    // APPDOMAIN_FORCE_TRIVIAL_WAIT_OPERATIONS |  // Do not pump messages during wait
    // APPDOMAIN_SECURITY_SANDBOXED |     // Causes assemblies not from the TPA list to be loaded as partially
trusted
    APPDOMAIN_ENABLE_PLATFORM_SPECIFIC_APPS |   // Enable platform-specific assemblies to run
    APPDOMAIN_ENABLE_PINVOKE_AND_CLASSIC_COMINTEROP | // Allow PInvoking from non-TPA assemblies
    APPDOMAIN_DISABLE_TRANSPARENCY_ENFORCEMENT;   // Entirely disables transparency checks
```

After deciding which AppDomain flags to use, AppDomain properties must be defined. The properties are key/value pairs of strings. Many of the properties relate to how the AppDomain will load assemblies.

Common AppDomain properties include:

- `TRUSTED_PLATFORM_ASSEMBLIES` This is a list of assembly paths (delimited by `;` on Windows and `:` on Linux/Mac) which the AppDomain should prioritize loading and give full trust to (even in partially-trusted domains). This list is meant to contain 'Framework' assemblies and other trusted modules, similar to the GAC in .NET Framework scenarios. Some hosts will put any library next to *coreclr.dll* on this list, others have hard-coded manifests listing trusted assemblies for their purposes.
- `APP_PATHS` This is a list of paths to probe in for an assembly if it can't be found in the trusted platform assemblies (TPA) list. Because the host has more control over which assemblies are loaded using the TPA list, it is a best practice for hosts to determine which assemblies they expect to load and list them explicitly. If probing at runtime is needed, however, this property can enable that scenario.
- `APP_NI_PATHS` This list is very similar to APP_PATHS except that it's meant to be paths that will be probed for native images.
- `NATIVE_DLL_SEARCH_DIRECTORIES` This property is a list of paths the loader should probe when looking for native DLLs called via p/invoke.
- `PLATFORM_RESOURCE_ROOTS` This list includes paths to probe in for resource satellite assemblies (in culture-specific sub-directories).

In our simple sample host, these properties are set up as follows:

```
// TRUSTED_PLATFORM_ASSEMBLIES
// "Trusted Platform Assemblies" are prioritized by the loader and always loaded with full trust.
// A common pattern is to include any assemblies next to CoreCLR.dll as platform assemblies.
// More sophisticated hosts may also include their own Framework extensions (such as AppDomain managers)
// in this list.
size_t tpaSize = 100 * MAX_PATH; // Starting size for our TPA (Trusted Platform Assemblies) list
wchar_t* trustedPlatformAssemblies = new wchar_t[tpaSize];
trustedPlatformAssemblies[0] = L'\0';

// Extensions to probe for when finding TPA list files
const wchar_t *tpaExtensions[] = {
    L"*.dll",
    L"*.exe",
    L"*.winmd"
};

// Probe next to CoreCLR.dll for any files matching the extensions from tpaExtensions and
// add them to the TPA list. In a real host, this would likely be extracted into a separate function
// and perhaps also run on other directories of interest.
for (int i = 0; i < _countof(tpaExtensions); i++)
{
    // Construct the file name search pattern
    wchar_t searchPath[MAX_PATH];
    wcscpy_s(searchPath, MAX_PATH, coreRoot);
    wcscat_s(searchPath, MAX_PATH, L"\\");
    wcscat_s(searchPath, MAX_PATH, tpaExtensions[i]);

    // Find files matching the search pattern
    WIN32_FIND_DATAW findData;
    HANDLE fileHandle = FindFirstFileW(searchPath, &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Construct the full path of the trusted assembly
            wchar_t pathToAdd[MAX_PATH];
            wcscpy_s(pathToAdd, MAX_PATH, coreRoot);
            wcscat_s(pathToAdd, MAX_PATH, L"\\");
            wcscat_s(pathToAdd, MAX_PATH, findData.cFileName);

            // Check to see if TPA list needs expanded
```

```
            if (wcsnlen(pathToAdd, MAX_PATH) + (3) + wcsnlen(trustedPlatformAssemblies, tpaSize) >= tpaSize)
            {
                // Expand, if needed
                tpaSize *= 2;
                wchar_t* newTPAList = new wchar_t[tpaSize];
                wcscpy_s(newTPAList, tpaSize, trustedPlatformAssemblies);
                trustedPlatformAssemblies = newTPAList;
            }

            // Add the assembly to the list and delimited with a semi-colon
            wcscat_s(trustedPlatformAssemblies, tpaSize, pathToAdd);
            wcscat_s(trustedPlatformAssemblies, tpaSize, L";");

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
path,
            // we can ignore that complication.
        }
        while (FindNextFileW(fileHandle, &findData));
        FindClose(fileHandle);
    }
}


// APP_PATHS
// App paths are directories to probe in for assemblies which are not one of the well-known Framework
assemblies
// included in the TPA list.
//
// For this simple sample, we just include the directory the target application is in.
// More complex hosts may want to also check the current working directory or other
// locations known to contain application assets.
wchar_t appPaths[MAX_PATH * 50];

// Just use the targetApp provided by the user and remove the file name
wcscpy_s(appPaths, targetAppPath);


// APP_NI_PATHS
// App (NI) paths are the paths that will be probed for native images not found on the TPA list.
// It will typically be similar to the app paths.
// For this sample, we probe next to the app and in a hypothetical directory of the same name with 'NI'
suffixed to the end.
wchar_t appNiPaths[MAX_PATH * 50];
wcscpy_s(appNiPaths, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L";");
wcscat_s(appNiPaths, MAX_PATH * 50, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L"NI");


// NATIVE_DLL_SEARCH_DIRECTORIES
// Native dll search directories are paths that the runtime will probe for native DLLs called via PInvoke
wchar_t nativeDllSearchDirectories[MAX_PATH * 50];
wcscpy_s(nativeDllSearchDirectories, appPaths);
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, L";");
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, coreRoot);


// PLATFORM_RESOURCE_ROOTS
// Platform resource roots are paths to probe in for resource assemblies (in culture-specific sub-directories)
wchar_t platformResourceRoots[MAX_PATH * 50];
wcscpy_s(platformResourceRoots, appPaths);
```

Step 6 - Create the AppDomain

Once all AppDomain flags and properties are prepared, `ICLRRuntimeHost4::CreateAppDomainWithManager` can be used to set up the AppDomain. This function optionally takes a fully qualified assembly name and type name to use as the domain's AppDomain manager. An AppDomain manager can allow a host to control some aspects of AppDomain behavior and may provide entry points for launching managed code if the host doesn't intend to invoke user code directly.

```
DWORD domainId;

// Setup key/value pairs for AppDomain  properties
const wchar_t* propertyKeys[] = {
    L"TRUSTED_PLATFORM_ASSEMBLIES",
    L"APP_PATHS",
    L"APP_NI_PATHS",
    L"NATIVE_DLL_SEARCH_DIRECTORIES",
    L"PLATFORM_RESOURCE_ROOTS"
};

// Property values which were constructed in step 5
const wchar_t* propertyValues[] = {
    trustedPlatformAssemblies,
    appPaths,
    appNiPaths,
    nativeDllSearchDirectories,
    platformResourceRoots
};

// Create the AppDomain
hr = runtimeHost->CreateAppDomainWithManager(
    L"Sample Host AppDomain",  // Friendly AD name
    appDomainFlags,
    NULL,       // Optional AppDomain manager assembly name
    NULL,       // Optional AppDomain manager type (including namespace)
    sizeof(propertyKeys) / sizeof(wchar_t*),
    propertyKeys,
    propertyValues,
    &domainId);
```

Step 7 - Run managed code!

With an AppDomain up and running, the host can now start executing managed code. The easiest way to do this is to use `ICLRRuntimeHost4::ExecuteAssembly` to invoke a managed assembly's entry point method. Note that this function only works in single-domain scenarios.

```
DWORD exitCode = -1;
hr = runtimeHost->ExecuteAssembly(domainId, targetApp, argc - 1, (LPCWSTR*)(argc > 1 ? &argv[1] : NULL),
&exitCode);
```

Another option, if `ExecuteAssembly` doesn't meet your host's needs, is to use `CreateDelegate` to create a function pointer to a static managed method. This requires the host to know the signature of the method it is calling into (in order to create the function pointer type) but allows hosts the flexibility to invoke code other than an assembly's entry point. The assembly name provided in the second parameter is the full managed assembly name of the library to load.

```
void *pfnDelegate = NULL;
hr = runtimeHost->CreateDelegate(
    domainId,
    L"HW, Version=1.0.0.0, Culture=neutral", // Target managed assembly
    L"ConsoleApplication.Program",           // Target managed type
    L"Main",                                 // Target entry point (static method)
    (INT_PTR*)&pfnDelegate);

((MainMethodFp*)pfnDelegate)(NULL);
```

Step 8 - Clean up

Finally, the host should clean up after itself by unloading AppDomains, stopping the runtime, and releasing the `ICLRRuntimeHost4` reference.

```
runtimeHost->UnloadAppDomain(domainId, true /* Wait until unload complete */);
runtimeHost->Stop();
runtimeHost->Release();
```

CoreCLR does not support unloading. Do not unload the CoreCLR library.

Conclusion

Once your host is built, it can be tested by running it from the command line and passing any arguments the host expects (like the managed app to run for the mscoree example host). When specifying the .NET Core app for the host to run, be sure to use the .dll that is produced by `dotnet build`. Executables (.exe files) produced by `dotnet publish` for self-contained applications are actually the default .NET Core host (so that the app can be launched directly from the command line in mainline scenarios); user code is compiled into a dll of the same name.

If things don't work initially, double-check that *coreclr.dll* is available in the location expected by the host, that all necessary Framework libraries are in the TPA list, and that CoreCLR's bitness (32- or 64-bit) matches how the host was built.

Hosting the .NET Core runtime is an advanced scenario that many developers won't require, but for those who need to launch managed code from a native process, or who need more control over the .NET Core runtime's behavior, it can be very useful.

Exposing .NET Core Components to COM

7/30/2019 • 2 minutes to read • Edit Online

In .NET Core, the process for exposing your .NET objects to COM has been significantly streamlined in comparison to .NET Framework. The following process will walk you through how to expose a class to COM. This tutorial shows you how to:

- Expose a class to COM from .NET Core.
- Generate a COM server as part of building your .NET Core library.
- Automatically generate a side-by-side server manifest for Registry-Free COM.

Prerequisites

- Install the .NET Core 3.0 Preview 7 SDK or a newer version.

Create the library

The first step is to create the library.

1. Create a new folder, and in that folder run `dotnet new classlib`.
2. Open `Class1.cs`.
3. Add `using System.Runtime.InteropServices;` to the top of the file.
4. Create an interface named `IServer`. For example:

```
using System;
using System.Runtime.InteropServices;

[ComVisible(true)]
[Guid(ContractGuids.ServerInterface)]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IServer
{
    /// <summary>
    /// Compute the value of the constant Pi.
    /// </summary>
    double ComputePi();
}
```

5. Add the `[Guid("<IID>")]` attribute to the interface, with the interface GUID for the COM interface you're implementing. For example, `[Guid("fe103d6e-e71b-414c-80bf-982f18f6c1c7")]`. Note that this GUID needs to be unique since it is the only identifier of this interface for COM. In Visual Studio, you can generate a GUID by going to Tools > Create GUID to open the Create GUID tool.
6. Add the `[InterfaceType]` attribute to the interface and specify what base COM interfaces your interface should implement.
7. Create a class named `Server` that implements `IServer`.
8. Add the `[Guid("<CLSID>")]` attribute to the class, with the class identifier GUID for the COM class you're implementing. For example, `[Guid("9f35b6f5-2c05-4e7f-93aa-ee087f6e7ab6")]`. As with the interface GUID, this GUID must be unique since it is the only identifier of this interface to COM.
9. Add the `[ComVisible(true)]` attribute to both the interface and the class.

> **IMPORTANT**
> Unlike in .NET Framework, .NET Core requires you to specify the CLSID of any class you want to be activatable via COM.

Generate the COM host

1. Open the `.csproj` project file and add `<EnableComHosting>true</EnableComHosting>` inside a `<PropertyGroup></PropertyGroup>` tag.
2. Build the project.

The resulting output will have a `ProjectName.dll`, `ProjectName.deps.json`, `ProjectName.runtimeconfig.json` and `ProjectName.comhost.dll` file.

Register the COM host for COM

Open an elevated command prompt and run `regsvr32 ProjectName.comhost.dll`. That will register all of your exposed .NET objects with COM.

Enabling RegFree COM

1. Open the `.csproj` project file and add `<EnableRegFreeCom>true</EnableRegFreeCom>` inside a `<PropertyGroup></PropertyGroup>` tag.
2. Build the project.

The resulting output will now also have a `ProjectName.X.manifest` file. This file is the side-by-side manifest for use with Registry-Free COM.

Sample

There is a fully functional COM server sample in the dotnet/samples repository on GitHub.

Additional Notes

Unlike in .NET Framework, there is no support in .NET Core for generating a COM Type Library (TLB) from a .NET Core assembly. You will either have to manually write an IDL file or a C++ header for the native declarations of your interfaces.

Additionally, loading both .NET Framework and .NET Core into the same process is unsupported, and as a result loading a .NET Core COM server into a .NET Framework COM client process or vice versa is not supported.

Packages, metapackages and frameworks

4/8/2019 • 7 minutes to read • Edit Online

.NET Core is a platform made of NuGet packages. Some product experiences benefit from fine-grained definition of packages while others from coarse-grained. To accommodate this duality, the product is distributed as a fine-grained set of packages and in coarser chunks with a package type informally called a metapackage.

Each of the .NET Core packages support being run on multiple .NET implementations, represented as frameworks. Some of those frameworks are traditional frameworks, like `net46`, representing the .NET Framework. Another set is new frameworks that can be thought of as "package-based frameworks", which establish a new model for defining frameworks. These package-based frameworks are entirely formed and defined as packages, forming a strong relationship between packages and frameworks.

Packages

.NET Core is split into a set of packages, which provide primitives, higher-level data types, app composition types and common utilities. Each of these packages represent a single assembly of the same name. For example, System.Runtime contains System.Runtime.dll.

There are advantages to defining packages in a fine-grained manner:

- Fine-grained packages can ship on their own schedule with relatively limited testing of other packages.
- Fine-grained packages can provide differing OS and CPU support.
- Fine-grained packages can have dependencies specific to only one library.
- Apps are smaller because unreferenced packages don't become part of the app distribution.

Some of these benefits are only used in certain circumstances. For example, NET Core packages will typically ship on the same schedule with the same platform support. In the case of servicing, fixes can be distributed and installed as small single package updates. Due to the narrow scope of change, the validation and time to make a fix available is limited to what is needed for a single library.

The following is a list of the key NuGet packages for .NET Core:

- System.Runtime - The most fundamental .NET Core package, including Object, String, Array, Action, and IList<T>.
- System.Collections - A set of (primarily) generic collections, including List<T> and Dictionary<TKey,TValue>.
- System.Net.Http - A set of types for HTTP network communication, including HttpClient and HttpResponseMessage.
- System.IO.FileSystem - A set of types for reading and writing to local or networked disk-based storage, including File and Directory.
- System.Linq - A set of types for querying objects, including `Enumerable` and ILookup<TKey,TElement>.
- System.Reflection - A set of types for loading, inspecting, and activating types, including Assembly, TypeInfo and MethodInfo.

Typically, rather than including each package, it's easier and more robust to include a metapackage. However, when you need a single package, you can include it as in the following example, which references the System.Runtime package.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="System.Runtime" Version="4.3.0" />
  </ItemGroup>
</Project>
```

Metapackages

Metapackages are a NuGet package convention for describing a set of packages that are meaningful together. They represent this set of packages by making them dependencies. They can optionally establish a framework for this set of packages by specifying a framework.

Previous versions of the .NET Core tools (both project.json and csproj-based tools) by default specified both a framework and a metapackage. Currently, however, the metapackage is implicitly referenced by the target framework, so that each metapackage is tied to a target framework. For example, the `netstandard1.6` framework references the NetStandard.Library version 1.6.0 metapackage. Similarly, the `netcoreapp2.1` framework references the Microsoft.NETCore.App Version 2.1.0 metapackage. For more information, see Implicit metapackage package reference in the .NET Core SDK.

Targeting a framework and implicitly referencing a metapackage means that you in effect are adding a reference to each of its dependent packages as a single gesture. That makes all of the libraries in those packages available for IntelliSense (or similar experience) and for publishing your app.

There are advantages to using metapackages:

- Provides a convenient user experience to reference a large set of fine-grained packages.
- Defines a set of packages (including specific versions) that are tested and work well together.

The .NET Standard metapackage is:

- NETStandard.Library - Describes the libraries that are part of the ".NET Standard". Applies to all .NET implementations (for example, .NET Framework, .NET Core and Mono) that support .NET Standard. Establishes the 'netstandard' framework.

The key .NET Core metapackages are:

- Microsoft.NETCore.App - Describes the libraries that are part of the .NET Core distribution. Establishes the `.NETCoreApp` framework. Depends on the smaller `NETStandard.Library`.
- Microsoft.AspNetCore.App - Includes all the supported packages from ASP.NET Core and Entity Framework Core except those that contain third-party dependencies. See Microsoft.AspNetCore.App metapackage for ASP.NET Core for more information.
- Microsoft.AspNetCore.All - Includes all the supported packages from ASP.NET Core, Entity Framework Core, and internal and third-party dependencies used by ASP.NET Core and Entity Framework Core. See Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.x for more information.
- Microsoft.NETCore.Portable.Compatibility - A set of compatibility facades that enable mscorlib-based Portable Class Libraries (PCLs) to run on .NET Core.

Frameworks

.NET Core packages each support a set of runtime frameworks. Frameworks describe an available API set (and potentially other characteristics) that you can rely on when you target a given framework. They are versioned as new APIs are added.

For example, System.IO.FileSystem supports the following frameworks:

- .NETFramework,Version=4.6
- .NETStandard,Version=1.3
- 6 Xamarin platforms (for example, xamarinios10)

It is useful to contrast the first two of these frameworks, since they are examples of the two different ways that frameworks are defined.

The `.NETFramework,Version=4.6` framework represents the available APIs in the .NET Framework 4.6. You can produce libraries compiled with the .NET Framework 4.6 reference assemblies and then distribute those libraries in NuGet packages in a net46 lib folder. It will be used for apps that target the .NET Framework 4.6 or that are compatible with it. This is how all frameworks have traditionally worked.

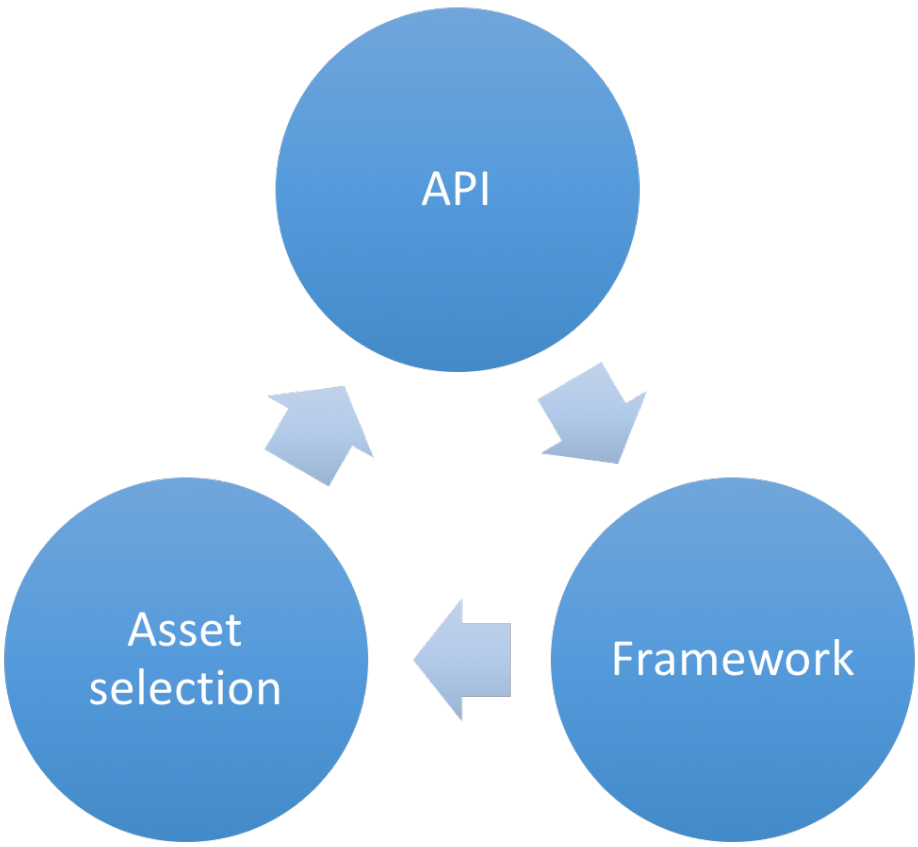
The `.NETStandard,Version=1.3` framework is a package-based framework. It relies on packages that target the framework to define and expose APIs in terms of the framework.

Package-based frameworks

There is a two-way relationship between frameworks and packages. The first part is defining the APIs available for a given framework, for example `netstandard1.3`. Packages that target `netstandard1.3` (or compatible frameworks, like `netstandard1.0`) define the APIs available for `netstandard1.3`. That may sound like a circular definition, but it isn't. By virtue of being "package-based", the API definition for the framework comes from packages. The framework itself doesn't define any APIs.

The second part of the relationship is asset selection. Packages can contain assets for multiple frameworks. Given a reference to a set of packages and/or metapackages, the framework is needed to determine which asset should be selected, for example `net46` or `netstandard1.3`. It is important to select the correct asset. For example, a `net46` asset is not likely to be compatible with .NET Framework 4.0 or .NET Core 1.0.

You can see this relationship in the following image. The *API* targets and defines the *framework*. The *framework* is used for *asset selection*. The *asset* gives you the API.



The two primary package-based frameworks used with .NET Core are:

- `netstandard`
- `netcoreapp`

.NET Standard

The .NET Standard (Target Framework Moniker: `netstandard`) framework represents the APIs defined by and built on top of the .NET Standard. Libraries that are intended to run on multiple runtimes should target this framework. They will be supported on any .NET Standard compliant runtime, such as .NET Core, .NET Framework and Mono/Xamarin. Each of these runtimes supports a set of .NET Standard versions, depending on which APIs they implement.

The `netstandard` framework implicitly references the `NETStandard.Library` metapackage. For example, the following MSBuild project file indicates that the project targets `netstandard1.6`, which references the `NETStandard.Library` version 1.6 metapackage.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
</Project>
```

However, the framework and metapackage references in the project file do not need to match, and you can use the `<NetStandardImplicitPackageVersion>` element in your project file to specify a framework version that is lower than the metapackage version. For example, the following project file is valid.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.3</TargetFramework>
    <NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
  </PropertyGroup>
</Project>
```

It may seem strange to target `netstandard1.3` but use the 1.6.0 version of `NETStandard.Library`. It is a valid use-case, since the metapackage maintains support for older `netstandard` versions. It could be the case you've standardized on the 1.6.0 version of the metapackage and use it for all your libraries, which target a variety of `netstandard` versions. With this approach, you only need to restore `NETStandard.Library` 1.6.0 and not earlier versions.

The reverse would not be valid: targeting `netstandard1.6` with the 1.3.0 version of `NETStandard.Library`. You cannot target a higher framework with a lower metapackage, since the lower version metapackage will not expose any assets for that higher framework. The versioning scheme for metapackages asserts that metapackages match the highest version of the framework they describe. By virtue of the versioning scheme, the first version of `NETStandard.Library` is v1.6.0 given that it contains `netstandard1.6` assets. v1.3.0 is used in the example above, for symmetry with the example above, but does not actually exist.

.NET Core application

The .NET Core (Target Framework Moniker: `netcoreapp`) framework represents the packages and associated APIs that come with the .NET Core distribution and the console application model that it provides. .NET Core apps must use this framework, due to targeting the console application model, as should libraries that intended to run only on .NET Core. Using this framework restricts apps and libraries to running only on .NET Core.

The `Microsoft.NETCore.App` metapackage targets the `netcoreapp` framework. It provides access to ~60 libraries, ~40 provided by the `NETStandard.Library` package and ~20 more in addition. You can reference additional libraries that target `netcoreapp` or compatible frameworks, such as `netstandard`, to get access to additional APIs.

Most of the additional libraries provided by `Microsoft.NETCore.App` also target `netstandard` given that their dependencies are satisfied by other `netstandard` libraries. That means that `netstandard` libraries can also reference those packages as dependencies.

High-level overview of changes in the .NET Core tools

3/8/2019 • 4 minutes to read • Edit Online

This document describes the changes associated with moving from *project.json* to MSBuild and the *csproj* project system with information on the changes to the layering of the .NET Core tooling and the implementation of the CLI commands. These changes occurred with the release of .NET Core SDK 1.0 and Visual Studio 2017 on March 7, 2017 (see the announcement) but were initially implemented with the release of the .NET Core SDK Preview 3.

Moving away from project.json

The biggest change in the tooling for .NET Core is certainly the move away from project.json to csproj as the project system. The latest versions of the command-line tools don't support *project.json* files. That means that it cannot be used to build, run or publish project.json based applications and libraries. In order to use this version of the tools, you will need to migrate your existing projects or start new ones.

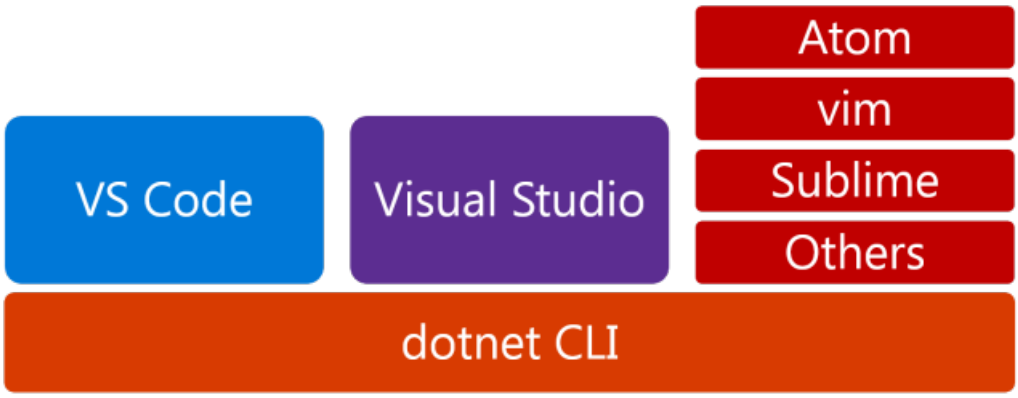
As part of this move, the custom build engine that was developed to build project.json projects was replaced with a mature and fully capable build engine called MSBuild. MSBuild is a well-known engine in the .NET community, since it has been a key technology since the platform's first release. Of course, because it needs to build .NET Core applications, MSBuild has been ported to .NET Core and can be used on any platform that .NET Core runs on. One of the main promises of .NET Core is that of a cross-platform development stack, and we have made sure that this move does not break that promise.

> **NOTE**
> If you are new to MSBuild and would like to learn more about it, you can start by reading the MSBuild Concepts article.

The tooling layers

With the move away from the existing project system as well as with building engine switches, the question that naturally follows is do any of these changes change the overall "layering" of the whole .NET Core tooling ecosystem? Are there new bits and components?

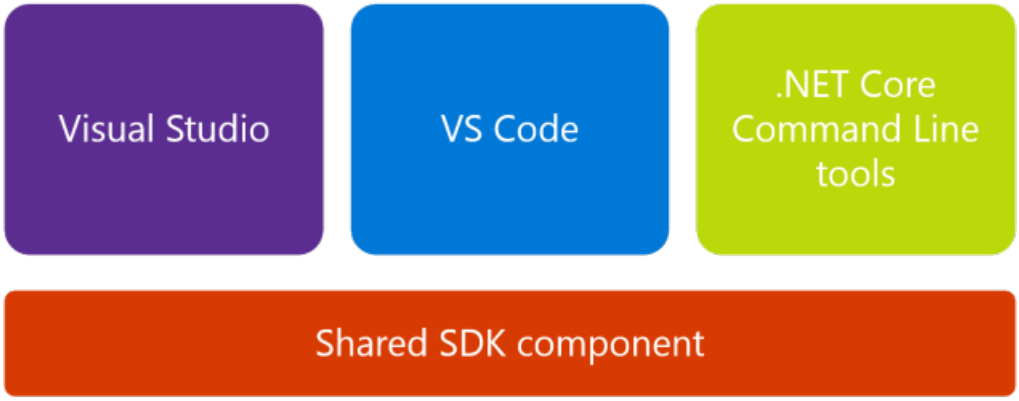
Let's start with a quick refresher on Preview 2 layering as shown in the following picture:



The layering of the tools is quite simple. At the bottom we have the .NET Core Command Line tools as a foundation. All other, higher-level tools such as Visual Studio or Visual Studio Code, depend and rely on the CLI to

build projects, restore dependencies and so on. This meant that, for example, if Visual Studio wanted to perform a restore operation, it would call into `dotnet restore` (see note) command in the CLI.

With the move to the new project system, the previous diagram changes:



The main difference is that the CLI is not the foundational layer anymore; this role is now filled by the "shared SDK component". This shared SDK component is a set of targets and associated tasks that are responsible for compiling your code, publishing it, packing NuGet packages etc. The SDK itself is open-source and is available on GitHub on the SDK repo.

> **NOTE**
> A "target" is a MSBuild term that indicates a named operation that MSBuild can invoke. It is usually coupled with one or more tasks that execute some logic that the target is supposed to do. MSBuild supports many ready-made targets such as `Copy` or `Execute`; it also allows users to write their own tasks using managed code and define targets to execute those tasks. For more information, see MSBuild tasks.

All the toolsets now consume the shared SDK component and its targets, CLI included. For example, the next version of Visual Studio will not call into `dotnet restore` (see note) command to restore dependencies for .NET Core projects, it will use the "Restore" target directly. Since these are MSBuild targets, you can also use raw MSBuild to execute them using the dotnet msbuild command.

CLI commands

The shared SDK component means that the majority of existing CLI commands have been re-implemented as MSBuild tasks and targets. What does this mean for the CLI commands and your usage of the toolset?

From an usage perspective, it doesn't change the way you use the CLI. The CLI still has the core commands that exist in Preview 2 release:

- `new`
- `restore`
- `run`
- `build`
- `publish`
- `test`
- `pack`

These commands still do what you expect them to do (new up a project, build it, publish it, pack it and so on). Majority of the options are not changed, and are still there, and you can consult either the commands' help screens (using `dotnet <command> --help`) or documentation on this site to get familiar with any changes.

From an execution perspective, the CLI commands will take their parameters and construct a call to "raw" MSBuild that will set the needed properties and run the desired target. To better illustrate this, consider the following

command:

```
dotnet publish -o pub -c Release
```

This command is publishing an application into a `pub` folder using the "Release" configuration. Internally, this command gets translated into the following MSBuild invocation:

```
dotnet msbuild -t:Publish -p:OutputPath=pub -p:Configuration=Release
```

The notable exception to this rule are `new` and `run` commands, as they have not been implemented as MSBuild targets.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Managing dependencies with .NET Core SDK 1.0

4/16/2019 • 2 minutes to read • Edit Online

With the move of .NET Core projects from project.json to csproj and MSBuild, a significant investment also happened that resulted in unification of the project file and assets that allow tracking of dependencies. For .NET Core projects this is similar to what project.json did. There is no separate JSON or XML file that tracks NuGet dependencies. With this change, we've also introduced another type of *reference* into the csproj syntax called the `<PackageReference>`.

This document describes the new reference type. It also shows how to add a package dependency using this new reference type to your project.

The new <PackageReference> element

The `<PackageReference>` has the following basic structure:

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" />
```

If you are familiar with MSBuild, it will look familiar to the other reference types that already exist. The key is the `Include` statement which specifies the package id that you wish to add to the project. The `<Version>` child element specifies the version to get. The versions are specified as per NuGet version rules.

> **NOTE**
> If you are not familiar with the overall `csproj` syntax, see the MSBuild project reference documentation for more information.

Adding a dependency that is available only in a specific target is done using conditions like in the following example:

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" Condition="'$(TargetFramework)' ==
'netcoreapp2.1'" />
```

The above means that the dependency will only be valid if the build is happening for that given target. The `$(TargetFramework)` in the condition is a MSBuild property that is being set in the project. For most common .NET Core applications, you will not need to do this.

Adding a dependency to your project

Adding a dependency to your project is straightforward. Here is an example of how to add Json.NET version `9.0.1` to your project. Of course, it is applicable to any other NuGet dependency.

When you open your project file, you will see two or more `<ItemGroup>` nodes. You will notice that one of the nodes already has `<PackageReference>` elements in it. You can add your new dependency to this node, or create a new one; it is completely up to you as the result will be the same.

In this example we will use the default template that is dropped by `dotnet new console`. This is a simple console application. When we open up the project, we first find the `<ItemGroup>` with already existing `<PackageReference>` in it. We then add the following to it:

```
<PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
```

After this, we save the project and run the `dotnet restore` command to install the dependency.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

The full project looks like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
  </ItemGroup>
</Project>
```

Removing a dependency from the project

Removing a dependency from the project file involves simply removing the `<PackageReference>` from the project file.

Additions to the csproj format for .NET Core

6/26/2019 • 14 minutes to read • Edit Online

This document outlines the changes that were added to the project files as part of the move from *project.json* to *csproj* and MSBuild. For more information about general project file syntax and reference, see the MSBuild project file documentation.

Implicit package references

Metapackages are implicitly referenced based on the target framework(s) specified in the `<TargetFramework>` or `<TargetFrameworks>` property of your project file. `<TargetFrameworks>` is ignored if `<TargetFramework>` is specified, independent of order. For more information, see Packages, metapackages and frameworks.

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>
```

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.1;net462</TargetFrameworks>
</PropertyGroup>
```

Recommendations

Since `Microsoft.NETCore.App` or `NETStandard.Library` metapackages are implicitly referenced, the following are our recommended best practices:

- When targeting .NET Core or .NET Standard, never have an explicit reference to the `Microsoft.NETCore.App` or `NETStandard.Library` metapackages via a `<PackageReference>` item in your project file.
- If you need a specific version of the runtime when targeting .NET Core, you should use the `<RuntimeFrameworkVersion>` property in your project (for example, `1.0.4`) instead of referencing the metapackage.
  - This might happen if you are using self-contained deployments and you need a specific patch version of 1.0.0 LTS runtime, for example.
- If you need a specific version of the `NETStandard.Library` metapackage when targeting .NET Standard, you can use the `<NetStandardImplicitPackageVersion>` property and set the version you need.
- Don't explicitly add or update references to either the `Microsoft.NETCore.App` or `NETStandard.Library` metapackage in .NET Framework projects. If any version of `NETStandard.Library` is needed when using a .NET Standard-based NuGet package, NuGet automatically installs that version.

Implicit version for some package references

Most usages of `<PackageReference>` require setting the `Version` attribute to specify the NuGet package version to be used. When using .NET Core 2.1 or 2.2 and referencing Microsoft.AspNetCore.App or Microsoft.AspNetCore.All, however, the attribute is unnecessary. The .NET Core SDK can automatically select the version of these packages that should be used.

Recommendation

When referencing the `Microsoft.AspNetCore.App` or `Microsoft.AspNetCore.All` packages, do not specify their version. If a version is specified, the SDK may produce warning NETSDK1071. To fix this warning, remove the package version like in the following example:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

> Known issue: the .NET Core 2.1 SDK only supported this syntax when the project also uses Microsoft.NET.Sdk.Web. This is resolved in the .NET Core 2.2 SDK.

These references to ASP.NET Core metapackages have a slightly different behavior from most normal NuGet packages. Framework-dependent deployments of applications that use these metapackages automatically take advantage of the ASP.NET Core shared framework. When you use the metapackages, **no** assets from the referenced ASP.NET Core NuGet packages are deployed with the application—the ASP.NET Core shared framework contains these assets. The assets in the shared framework are optimized for the target platform to improve application startup time. For more information about shared framework, see .NET Core distribution packaging.

If a version *is* specified, it's treated as the *minimum* version of ASP.NET Core shared framework for framework-dependent deployments and as an *exact* version for self-contained deployments. This can have the following consequences:

- If the version of ASP.NET Core installed on the server is less than the version specified on the PackageReference, the .NET Core process fails to launch. Updates to the metapackage are often available on NuGet.org before updates have been made available in hosting environments such as Azure. Updating the version on the PackageReference to ASP.NET Core could cause a deployed application to fail.
- If the application is deployed as a self-contained deployment, the application may not contain the latest security updates to .NET Core. When a version isn't specified, the SDK can automatically include the newest version of ASP.NET Core in the self-contained deployment.

Default compilation includes in .NET Core projects

With the move to the *csproj* format in the latest SDK versions, we've moved the default includes and excludes for compile items and embedded resources to the SDK properties files. This means that you no longer need to specify these items in your project file.

The main reason for doing this is to reduce the clutter in your project file. The defaults that are present in the SDK should cover most common use cases, so there is no need to repeat them in every project that you create. This leads to smaller project files that are much easier to understand as well as edit by hand, if needed.

The following table shows which element and which globs are both included and excluded in the SDK:

ELEMENT	INCLUDE GLOB	EXCLUDE GLOB	REMOVE GLOB
Compile	**/*.cs (or other language extensions)	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	**/*.cs; **/*.resx

> **NOTE**
>
> **Exclude glob** always excludes the `./bin` and `./obj` folders, which are represented by the `$(BaseOutputPath)` and `$(BaseIntermediateOutputPath)` MSBuild properties, respectively. As a whole, all excludes are represented by `$(DefaultItemExcludes)`.

If you have globs in your project and you try to build it using the newest SDK, you'll get the following error:

> Duplicate Compile items were included. The .NET SDK includes Compile items from your project directory by default. You can either remove these items from your project file, or set the 'EnableDefaultCompileItems' property to 'false' if you want to explicitly include them in your project file.

In order to get around this error, you can either remove the explicit `Compile` items that match the ones listed on the previous table, or you can set the `<EnableDefaultCompileItems>` property to `false`, like this:

```
<PropertyGroup>
    <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
</PropertyGroup>
```

Setting this property to `false` will disable implicit inclusion, reverting to the behavior of previous SDKs where you had to specify the default globs in your project.

This change does not modify the main mechanics of other includes. However, if you wish to specify, for example, some files to get published with your app, you can still use the known mechanisms in *csproj* for that (for example, the `<Content>` element).

`<EnableDefaultCompileItems>` only disables `Compile` globs but doesn't affect other globs, like the implicit `None` glob, which also applies to *.cs items. Because of that, **Solution Explorer** will continue show *.cs items as part of the project, included as `None` items. In a similar way, you can set `<EnableDefaultNoneItems>` to false to disable the implicit `None` glob, like this:

```
<PropertyGroup>
    <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
</PropertyGroup>
```

To disable **all implicit globs**, you can set the `<EnableDefaultItems>` property to `false` as in the following example:

```
<PropertyGroup>
    <EnableDefaultItems>false</EnableDefaultItems>
</PropertyGroup>
```

How to see the whole project as MSBuild sees it

While those csproj changes greatly simplify project files, you might want to see the fully expanded project as MSBuild sees it once the SDK and its targets are included. Preprocess the project with the `/pp` switch of the `dotnet msbuild` command, which shows which files are imported, their sources, and their contributions to the build without actually building the project:

```
dotnet msbuild -pp:fullproject.xml
```

If the project has multiple target frameworks, the results of the command should be focused on only one of them by specifying it as an MSBuild property:

```
dotnet msbuild -p:TargetFramework=netcoreapp2.0 -pp:fullproject.xml
```

Additions

Sdk attribute

The root `<Project>` element of the *.csproj* file has a new attribute called `Sdk`. `Sdk` specifies which SDK will be used by the project. The SDK, as the layering document describes, is a set of MSBuild tasks and targets that can build .NET Core code. We ship three main SDKs with the .NET Core tools and an additional two SDKs when using .NET Core 3.0

Preview:

1. The .NET Core SDK with the ID of `Microsoft.NET.Sdk`
2. The .NET Core web SDK with the ID of `Microsoft.NET.Sdk.Web`
3. The .NET Core Razor Class Library SDK with the ID of `Microsoft.NET.Sdk.Razor`
4. The .NET Core Worker Service with the ID of `Microsoft.NET.Sdk.Worker` (.NET Core 3.0 Preview)
5. The .NET Core WinForms and WPF with the ID of `Microsoft.NET.Sdk.WindowsDesktop` (.NET Core 3.0 Preview)

You need to have the `Sdk` attribute set to one of those IDs on the `<Project>` element in order to use the .NET Core tools and build your code.

PackageReference

A `<PackageReference>` item element specifies a NuGet dependency in the project. The `Include` attribute specifies the package ID.

```
<PackageReference Include="<package-id>" Version="" PrivateAssets="" IncludeAssets="" ExcludeAssets="" />
```

Version

The required `Version` attribute specifies the version of the package to restore. The attribute respects the rules of the NuGet versioning scheme. The default behavior is an exact version match. For example, specifying `Version="1.2.3"` is equivalent to NuGet notation `[1.2.3]` for the exact 1.2.3 version of the package.

IncludeAssets, ExcludeAssets and PrivateAssets

`IncludeAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should be consumed. By default, all package assets are included.

`ExcludeAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should not be consumed.

`PrivateAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should be consumed but not flow to the next project. The `Analyzers`, `Build` and `ContentFiles` assets are private by default when this attribute is not present.

> **NOTE**
> `PrivateAssets` is equivalent to the *project.json/xproj* `SuppressParent` element.

These attributes can contain one or more of the following items, separated by the semicolon `;` character if more than one is listed:

- `Compile` – the contents of the lib folder are available to compile against.
- `Runtime` – the contents of the runtime folder are distributed.
- `ContentFiles` – the contents of the *contentfiles* folder are used.
- `Build` – the props/targets in the build folder are used.
- `Native` – the contents from native assets are copied to the output folder for runtime.
- `Analyzers` – the analyzers are used.

Alternatively, the attribute can contain:

- `None` – none of the assets are used.
- `All` – all assets are used.

DotNetCliToolReference

A `<DotNetCliToolReference>` item element specifies the CLI tool that the user wants to restore in the context of the project. It's a replacement for the `tools` node in *project.json*.

```
<DotNetCliToolReference Include="<package-id>" Version="" />
```

Version

`Version` specifies the version of the package to restore. The attribute respects the rules of the NuGet versioning scheme. The default behavior is an exact version match. For example, specifying `Version="1.2.3"` is equivalent to NuGet notation `[1.2.3]` for the exact 1.2.3 version of the package.

RuntimeIdentifiers

The `<RuntimeIdentifiers>` property element lets you specify a semicolon-delimited list of Runtime Identifiers (RIDs) for the project. RIDs enable publishing self-contained deployments.

```
<RuntimeIdentifiers>win10-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
```

RuntimeIdentifier

The `<RuntimeIdentifier>` property element allows you to specify only one Runtime Identifier (RID) for the project. The RID enables publishing a self-contained deployment.

```
<RuntimeIdentifier>ubuntu.16.04-x64</RuntimeIdentifier>
```

Use `<RuntimeIdentifiers>` (plural) instead if you need to publish for multiple runtimes. `<RuntimeIdentifier>` can provide faster builds when only a single runtime is required.

PackageTargetFallback

The `<PackageTargetFallback>` property element allows you to specify a set of compatible targets to be used when restoring packages. It's designed to allow packages that use the dotnet TxM (Target x Moniker) to operate with packages that don't declare a dotnet TxM. If your project uses the dotnet TxM, then all the packages it depends on must also have a dotnet TxM, unless you add the `<PackageTargetFallback>` to your project in order to allow non-dotnet platforms to be compatible with dotnet.

The following example provides the fallbacks for all targets in your project:

```
<PackageTargetFallback>
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

The following example specifies the fallbacks only for the `netcoreapp2.1` target:

```
<PackageTargetFallback Condition="'$(TargetFramework)'=='netcoreapp2.1'">
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

NuGet metadata properties

With the move to MSBuild, we have moved the input metadata that is used when packing a NuGet package from *project.json* to *.csproj* files. The inputs are MSBuild properties so they have to go within a `<PropertyGroup>` group. The following is the list of properties that are used as inputs to the packing process when using the `dotnet pack` command or the `Pack` MSBuild target that is part of the SDK.

IsPackable

A Boolean value that specifies whether the project can be packed. The default value is `true`.

PackageVersion

Specifies the version that the resulting package will have. Accepts all forms of NuGet version string. Default is the

value of `$(Version)`, that is, of the property `Version` in the project.

PackageId

Specifies the name for the resulting package. If not specified, the `pack` operation will default to using the `AssemblyName` or directory name as the name of the package.

Title

A human-friendly title of the package, typically used in UI displays as on nuget.org and the Package Manager in Visual Studio. If not specified, the package ID is used instead.

Authors

A semicolon-separated list of packages authors, matching the profile names on nuget.org. These are displayed in the NuGet Gallery on nuget.org and are used to cross-reference packages by the same authors.

PackageDescription

A long description of the package for UI display.

Description

A long description for the assembly. If `PackageDescription` is not specified then this property is also used as the description of the package.

Copyright

Copyright details for the package.

PackageRequireLicenseAcceptance

A Boolean value that specifies whether the client must prompt the consumer to accept the package license before installing the package. The default is `false`.

PackageLicenseExpression

An SPDX license identifier or expression. For example, `Apache-2.0`.

Here is the complete list of SPDX license identifiers. NuGet.org accepts only OSI or FSF approved licenses when using license type expression.

The exact syntax of the license expressions is described below in ABNF.

```
license-id            = <short form license identifier from https://spdx.org/spdx-specification-21-web-
version#h.luq9dgcle9mo>

license-exception-id  = <short form license exception identifier from https://spdx.org/spdx-specification-21-web-
version#h.ruv3yl8g6czd>

simple-expression = license-id / license-id"+"

compound-expression =  1*1(simple-expression /
                simple-expression "WITH" license-exception-id /
                compound-expression "AND" compound-expression /
                compound-expression "OR" compound-expression ) /
                "(" compound-expression ")" )

license-expression =  1*1(simple-expression / compound-expression / UNLICENSED)
```

> **NOTE**
>
> Only one of `PackageLicenseExpression`, `PackageLicenseFile` and `PackageLicenseUrl` can be specified at a time.

PackageLicenseFile

Path to a license file within the package if you are using a license that hasn't been assigned an SPDX identifier, or it is a

custom license (Otherwise `PackageLicenseExpression` is preferred)

Replaces `PackageLicenseUrl`, can't be combined with `PackageLicenseExpression` and requires Visual Studio 15.9.4, .NET SDK 2.1.502 or 2.2.101, or newer.

You will need to ensure the license file is packed by adding it explicitly to the project, example usage:

```
<PropertyGroup>
  <PackageLicenseFile>LICENSE.txt</PackageLicenseFile>
</PropertyGroup>
<ItemGroup>
  <None Include="licenses\LICENSE.txt" Pack="true" PackagePath="$(PackageLicenseFile)"/>
</ItemGroup>
```

PackageLicenseUrl

An URL to the license that is applicable to the package. (*deprecated since Visual Studio 15.9.4, .NET SDK 2.1.502 and 2.2.101*)

PackageIconUrl

A URL for a 64x64 image with transparent background to use as the icon for the package in UI display.

PackageReleaseNotes

Release notes for the package.

PackageTags

A semicolon-delimited list of tags that designates the package.

PackageOutputPath

Determines the output path in which the packed package will be dropped. Default is `$(OutputPath)`.

IncludeSymbols

This Boolean value indicates whether the package should create an additional symbols package when the project is packed. The symbols package's format is controlled by the `SymbolPackageFormat` property.

SymbolPackageFormat

Specifies the format of the symbols package. If "symbols.nupkg", a legacy symbols package will be created with a *.symbols.nupkg* extension containing PDBs, DLLs, and other output files. If "snupkg", a snupkg symbol package will be created containing the portable PDBs. Default is "symbols.nupkg".

IncludeSource

This Boolean value indicates whether the pack process should create a source package. The source package contains the library's source code as well as PDB files. Source files are put under the `src/ProjectName` directory in the resulting package file.

IsTool

Specifies whether all output files are copied to the *tools* folder instead of the *lib* folder. Note that this is different from a `DotNetCliTool` which is specified by setting the `PackageType` in the *.csproj* file.

RepositoryUrl

Specifies the URL for the repository where the source code for the package resides and/or from which it's being built.

RepositoryType

Specifies the type of the repository. Default is "git".

NoPackageAnalysis

Specifies that pack should not run package analysis after building the package.

MinClientVersion

Specifies the minimum version of the NuGet client that can install this package, enforced by nuget.exe and the Visual Studio Package Manager.

IncludeBuildOutput

This Boolean values specifies whether the build output assemblies should be packed into the *.nupkg* file or not.

IncludeContentInPack

This Boolean value specifies whether any items that have a type of `Content` will be included in the resulting package automatically. The default is `true`.

BuildOutputTargetFolder

Specifies the folder where to place the output assemblies. The output assemblies (and other output files) are copied into their respective framework folders.

ContentTargetFolders

This property specifies the default location of where all the content files should go if `PackagePath` is not specified for them. The default value is "content;contentFiles".

NuspecFile

Relative or absolute path to the *.nuspec* file being used for packing.

> **NOTE**
> If the *.nuspec* file is specified, it's used **exclusively** for packaging information and any information in the projects is not used.

NuspecBasePath

Base path for the *.nuspec* file.

NuspecProperties

Semicolon separated list of key=value pairs.

AssemblyInfo properties

Assembly attributes that were typically present in an *AssemblyInfo* file are now automatically generated from properties.

Properties per attribute

Each attribute has a property that control its content and another to disable its generation as shown in the following table:

ATTRIBUTE	PROPERTY	PROPERTY TO DISABLE
AssemblyCompanyAttribute	`Company`	`GenerateAssemblyCompanyAttribute`
AssemblyConfigurationAttribute	`Configuration`	`GenerateAssemblyConfigurationAttribute`
AssemblyCopyrightAttribute	`Copyright`	`GenerateAssemblyCopyrightAttribute`
AssemblyDescriptionAttribute	`Description`	`GenerateAssemblyDescriptionAttribute`
AssemblyFileVersionAttribute	`FileVersion`	`GenerateAssemblyFileVersionAttribute`
AssemblyInformationalVersionAttribute	`InformationalVersion`	`GenerateAssemblyInformationalVersionAttribu`
AssemblyProductAttribute	`Product`	`GenerateAssemblyProductAttribute`

ATTRIBUTE	PROPERTY	PROPERTY TO DISABLE
AssemblyTitleAttribute	`AssemblyTitle`	`GenerateAssemblyTitleAttribute`
AssemblyVersionAttribute	`AssemblyVersion`	`GenerateAssemblyVersionAttribute`
NeutralResourcesLanguageAttribute	`NeutralLanguage`	`GenerateNeutralResourcesLanguageAttribute`

Notes:

- `AssemblyVersion` and `FileVersion` default is to take the value of `$(Version)` without suffix. For example, if `$(Version)` is `1.2.3-beta.4`, then the value would be `1.2.3`.
- `InformationalVersion` defaults to the value of `$(Version)`.
- `InformationalVersion` has `$(SourceRevisionId)` appended if the property is present. It can be disabled using `IncludeSourceRevisionInInformationalVersion`.
- `Copyright` and `Description` properties are also used for NuGet metadata.
- `Configuration` is shared with all the build process and set via the `--configuration` parameter of `dotnet` commands.

GenerateAssemblyInfo

A Boolean that enable or disable all the AssemblyInfo generation. The default value is `true`.

GeneratedAssemblyInfoFile

The path of the generated assembly info file. Default to a file in the `$(IntermediateOutputPath)` (obj) directory.

Migrate from .NET Core 2.0 to 2.1

1/23/2019 • 2 minutes to read • Edit Online

This article shows you the basic steps for migrating your .NET Core 2.0 app to 2.1. If you're looking to migrate your ASP.NET Core app to 2.1, see Migrate from ASP.NET Core 2.0 to 2.1.

For an overview of the new features in .NET Core 2.1, see What's new in .NET Core 2.1.

Update the project file to use 2.1 versions

- Open the project file (the *.csproj, *.vbproj, or *.fsproj file).
- Change the target framework value from `netcoreapp2.0` to `netcoreapp2.1`. The target framework is defined by the `<TargetFramework>` or `<TargetFrameworks>` element.

  For example, change `<TargetFramework>netcoreapp2.0</TargetFramework>` to `<TargetFramework>netcoreapp2.1</TargetFramework>`.
- Remove `<DotNetCliToolReference>` references for tools that are bundled in the .NET Core 2.1 SDK (v 2.1.300 or later). These references include:
  - dotnet-watch (Microsoft.DotNet.Watcher.Tools)
  - dotnet-user-secrets (Microsoft.Extensions.SecretManager.Tools)
  - dotnet-sql-cache (Microsoft.Extensions.Caching.SqlConfig.Tools)
  - dotnet-ef (Microsoft.EntityFrameworkCore.Tools.DotNet)

  In previous .NET Core SDK versions, the reference to one of these tools in your project file looks similar to the following example:

  ```
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  ```

  Since this entry isn't used by the .NET Core SDK any longer, you'll see a warning similar to the following if you still have references to one of these bundled tools in your project:

  ```
  The tool 'Microsoft.EntityFrameworkCore.Tools.DotNet' is now included in the .NET Core SDK. Here is information on resolving this warning.
  ```

  Removing the `<DotNetCliToolReference>` references for those tools from your project file fixes this issue.

See also

- Migrate from ASP.NET Core 2.0 to 2.1
- What's new in .NET Core 2.1

Migrating .NET Core projects from project.json

3/8/2019 • 4 minutes to read • Edit Online

This document will cover migration scenarios for .NET Core projects and will go over the following three migration scenarios:

1. Migration from a valid latest schema of *project.json* to *csproj*
2. Migration from DNX to csproj
3. Migration from RC3 and previous .NET Core csproj projects to the final format

This document is only applicable to older .NET Core projects that still use project.json. It is not applicable for migrating from .NET Framework to .NET Core.

Migration from project.json to csproj

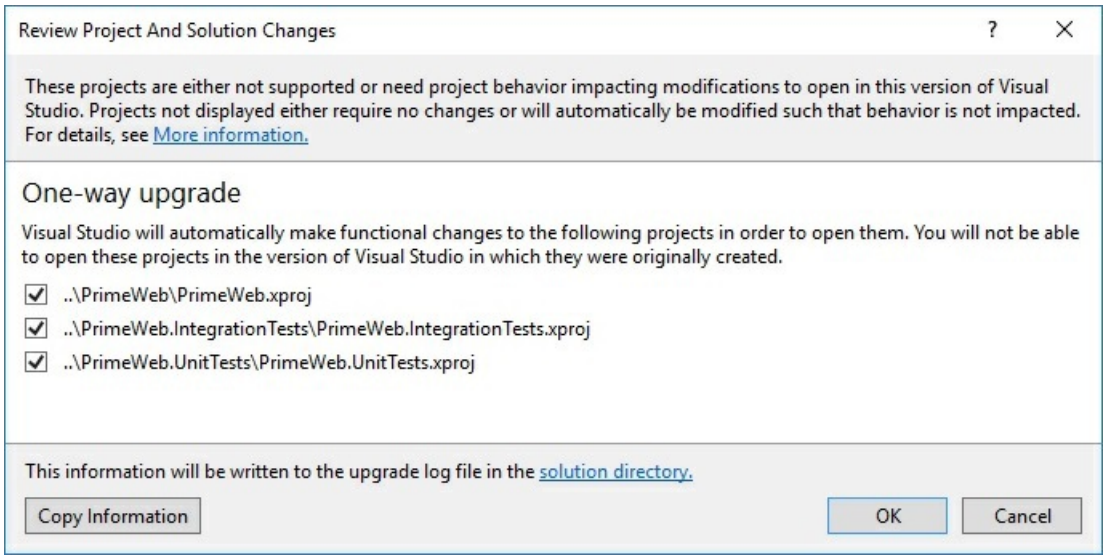
Migration from *project.json* to *.csproj* can be done using one of the following methods:

- Visual Studio 2017
- dotnet migrate command-line tool

Both methods use the same underlying engine to migrate the projects, so the results will be the same for both. In most cases, using one of these two ways to migrate the *project.json* to *csproj* is the only thing that is needed and no further manual editing of the project file is necessary. The resulting *.csproj* file will be named the same as the containing directory name.

Visual Studio 2017

When you open a *.xproj* file or a solution file which references *.xproj* files, the **One-way upgrade** dialog appears. The dialog displays the projects to be migrated. If you open a solution file, all the projects specified in the solution file will be listed. Review the list of projects to be migrated and select **OK**.



Visual Studio will migrate the projects chosen automatically. When migrating a solution, if you don't choose all projects, the same dialog will appear asking you to upgrade the remaining projects from that solution. After the project is migrated, you can see and modify its contents by right-clicking the project in the **Solution Explorer** window and selecting **Edit <project name>.csproj**.

Files that were migrated (*project.json*, *global.json*, *.xproj* and solution file) will be moved to a *Backup* folder. The

solution file that is migrated will be upgraded to Visual Studio 2017 and you won't be able to open that solution file in previous versions of Visual Studio. A file named *UpgradeLog.htm* is also saved and automatically opened that contains a migration report.

> **IMPORTANT**
> The new tooling is not available in Visual Studio 2015, so you cannot migrate your projects using that version of Visual Studio.

dotnet migrate

In the command-line scenario, you can use the `dotnet migrate` command. It will migrate a project, a solution or a set of folders in that order, depending on which ones were found. When you migrate a project, the project and all its dependencies are migrated.

Files that were migrated (*project.json*, *global.json* and *.xproj*) will be moved to a *backup* folder.

> **NOTE**
> If you are using Visual Studio Code, the `dotnet migrate` command will not modify Visual Studio Code-specific files such as `tasks.json`. These files need to be changed manually. This is also true if you are using Project Ryder or any editor or Integrated Development Environment (IDE) other than Visual Studio.

See A mapping between project.json and csproj properties for a comparison of project.json and csproj formats.

Common issues

- If you get an error: "No executable found matching command dotnet-migrate":

Run `dotnet --version` to see which version you are using. `dotnet migrate` requires .NET Core CLI RC3 or higher. You'll get this error if you have a *global.json* file in the current or parent directory and the `sdk` version is set to an older version.

Migration from DNX to csproj

If you are still using DNX for .NET Core development, your migration process should be done in two stages:

1. Use the existing DNX migration guidance to migrate from DNX to project-json enabled CLI.
2. Follow the steps from the previous section to migrate from *project.json* to *.csproj*.

> **NOTE**
> DNX has become officially deprecated during the Preview 1 release of the .NET Core CLI.

Migration from earlier .NET Core csproj formats to RTM csproj

The .NET Core csproj format has been changing and evolving with each new pre-release version of the tooling. There is no tool that will migrate your project file from earlier versions of csproj to the latest, so you need to manually edit the project file. The actual steps depend on the version of the project file you are migrating. The following is some guidance to consider based on the changes that happened between versions:

- Remove the tools version property from the `<Project>` element, if it exists.
- Remove the XML namespace (`xmlns`) from the `<Project>` element.
- If it doesn't exist, add the `Sdk` attribute to the `<Project>` element and set it to `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Web`. This attribute specifies that the project uses the SDK to be used. `Microsoft.NET.Sdk.Web`

is used for web apps.

- Remove the `<Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" />` and `<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />` statements from the top and bottom of the project. These import statements are implied by the SDK, so there is no need for them to be in the project.
- If you have `Microsoft.NETCore.App` or `NETStandard.Library` `<PackageReference>` items in your project, you should remove them. These package references are implied by the SDK.
- Remove the `Microsoft.NET.Sdk` `<PackageReference>` element, if it exists. The SDK reference comes through the `Sdk` attribute on the `<Project>` element.
- Remove the globs that are implied by the SDK. Leaving these globs in your project will cause an error on build because compile items will be duplicated.

After these steps your project should be fully compatible with the RTM .NET Core csproj format.

For examples of before and after the migration from old csproj format to the new one, see the Updating Visual Studio 2017 RC – .NET Core Tooling improvements article on the .NET blog.

See also

- Port, Migrate, and Upgrade Visual Studio Projects

A mapping between project.json and csproj properties

2/23/2019 • 6 minutes to read • Edit Online

By Nate McMaster

During the development of the .NET Core tooling, an important design change was made to no longer support *project.json* files and instead move the .NET Core projects to the MSBuild/csproj format.

This article shows how the settings in *project.json* are represented in the MSBuild/csproj format so you can learn how to use the new format and understand the changes made by the migration tools when you're upgrading your project to the latest version of the tooling.

The csproj format

The new format, *.csproj, is an XML-based format. The following example shows the root node of a .NET Core project using the `Microsoft.NET.Sdk`. For web projects, the SDK used is `Microsoft.NET.Sdk.Web`.

```
<Project Sdk="Microsoft.NET.Sdk">
...
</Project>
```

Common top-level properties

name

```
{
  "name": "MyProjectName"
}
```

No longer supported. In csproj, this is determined by the project filename, which usually matches the directory name. For example, `MyProjectName.csproj`.

By default, the project filename also specifies the value of the `<AssemblyName>` and `<PackageId>` properties.

```
<PropertyGroup>
  <AssemblyName>MyProjectName</AssemblyName>
  <PackageId>MyProjectName</PackageId>
</PropertyGroup>
```

The `<AssemblyName>` will have a different value than `<PackageId>` if `buildOptions\outputName` property was defined in project.json. For more information, see Other common build options.

version

```
{
  "version": "1.0.0-alpha-*"
}
```

Use the `VersionPrefix` and `VersionSuffix` properties:

```
<PropertyGroup>
  <VersionPrefix>1.0.0</VersionPrefix>
  <VersionSuffix>alpha</VersionSuffix>
</PropertyGroup>
```

You can also use the `Version` property, but this may override version settings during packaging:

```
<PropertyGroup>
  <Version>1.0.0-alpha</Version>
</PropertyGroup>
```

Other common root-level options

```
{
  "authors": [ "Anne", "Bob" ],
  "company": "Contoso",
  "language": "en-US",
  "title": "My library",
  "description": "This is my library.\r\nAnd it's really great!",
  "copyright": "Nugetizer 3000",
  "userSecretsId": "xyz123"
}
```

```
<PropertyGroup>
  <Authors>Anne;Bob</Authors>
  <Company>Contoso</Company>
  <NeutralLanguage>en-US</NeutralLanguage>
  <AssemblyTitle>My library</AssemblyTitle>
  <Description>This is my library.
And it's really great!</Description>
  <Copyright>Nugetizer 3000</Copyright>
  <UserSecretsId>xyz123</UserSecretsId>
</PropertyGroup>
```

frameworks

One target framework

```
{
  "frameworks": {
    "netcoreapp1.0": {}
  }
}
```

```
<PropertyGroup>
  <TargetFramework>netcoreapp1.0</TargetFramework>
</PropertyGroup>
```

Multiple target frameworks

```
{
  "frameworks": {
    "netcoreapp1.0": {},
    "net451": {}
  }
}
```

Use the `TargetFrameworks` property to define your list of target frameworks. Use semi-colon to separate multiple framework values.

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp1.0;net451</TargetFrameworks>
</PropertyGroup>
```

dependencies

> **IMPORTANT**
> If the dependency is a **project** and not a package, the format is different. For more information, see the dependency type section.

NETStandard.Library metapackage

```
{
  "dependencies": {
    "NETStandard.Library": "1.6.0"
  }
}
```

```
<PropertyGroup>
  <NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
</PropertyGroup>
```

Microsoft.NETCore.App metapackage

```
{
  "dependencies": {
    "Microsoft.NETCore.App": "1.0.0"
  }
}
```

```
<PropertyGroup>
  <RuntimeFrameworkVersion>1.0.3</RuntimeFrameworkVersion>
</PropertyGroup>
```

Note that the `<RuntimeFrameworkVersion>` value in the migrated project is determined by the version of the SDK you have installed.

Top-level dependencies

```
{
  "dependencies": {
    "Microsoft.AspNetCore": "1.1.0"
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.0" />
</ItemGroup>
```

Per-framework dependencies

```
{
  "framework": {
    "net451": {
      "dependencies": {
        "System.Collections.Immutable": "1.3.1"
      }
    },
    "netstandard1.5": {
      "dependencies": {
        "Newtonsoft.Json": "9.0.1"
      }
    }
  }
}
```

```
<ItemGroup Condition="'$(TargetFramework)'=='net451'">
  <PackageReference Include="System.Collections.Immutable" Version="1.3.1" />
</ItemGroup>

<ItemGroup Condition="'$(TargetFramework)'=='netstandard1.5'">
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
</ItemGroup>
```

imports

```
{
  "dependencies": {
    "YamlDotNet": "4.0.1-pre309"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": [
        "dnxcore50",
        "dotnet"
      ]
    }
  }
}
```

```
<PropertyGroup>
  <PackageTargetFallback>dnxcore50;dotnet</PackageTargetFallback>
</PropertyGroup>
<ItemGroup>
  <PackageReference Include="YamlDotNet" Version="4.0.1-pre309" />
</ItemGroup>
```

dependency type

type: project

```
{
  "dependencies": {
    "MyOtherProject": "1.0.0-*",
    "AnotherProject": {
      "type": "project"
    }
  }
}
```

```
<ItemGroup>
  <ProjectReference Include="..\MyOtherProject\MyOtherProject.csproj" />
  <ProjectReference Include="..\AnotherProject\AnotherProject.csproj" />
</ItemGroup>
```

> **NOTE**
>
> This will break the way that `dotnet pack --version-suffix $suffix` determines the dependency version of a project reference.

type: build

```
{
  "dependencies": {
    "Microsoft.EntityFrameworkCore.Design": {
      "version": "1.1.0",
      "type": "build"
    }
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="1.1.0" PrivateAssets="All" />
</ItemGroup>
```

type: platform

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.1.0",
      "type": "platform"
    }
  }
}
```

There is no equivalent in csproj.

runtimes

```
{
  "runtimes": {
    "win7-x64": {},
    "osx.10.11-x64": {},
    "ubuntu.16.04-x64": {}
  }
}
```

```
<PropertyGroup>
  <RuntimeIdentifiers>win7-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
</PropertyGroup>
```

Standalone apps (self-contained deployment)

In project.json, defining a `runtimes` section means the app was standalone during build and publish. In MSBuild, all projects are *portable* during build, but can be published as standalone.

```
dotnet publish --framework netcoreapp1.0 --runtime osx.10.11-x64
```

For more information, see Self-contained deployments (SCD).

tools

```
{
  "tools": {
    "Microsoft.EntityFrameworkCore.Tools.DotNet": "1.0.0-*"
  }
}
```

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
</ItemGroup>
```

> **NOTE**
>
> `imports` on tools are not supported in csproj. Tools that need imports will not work with the new `Microsoft.NET.Sdk`.

buildOptions

See also Files.

emitEntryPoint

```
{
  "buildOptions": {
    "emitEntryPoint": true
  }
}
```

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

If `emitEntryPoint` was `false`, the value of `OutputType` is converted to `Library`, which is the default value:

```
{
  "buildOptions": {
    "emitEntryPoint": false
  }
}
```

```
<PropertyGroup>
  <OutputType>Library</OutputType>
  <!-- or, omit altogether. It defaults to 'Library' -->
</PropertyGroup>
```

keyFile

```
{
  "buildOptions": {
    "keyFile": "MyKey.snk"
  }
}
```

The `keyFile` element expands to three properties in MSBuild:

```
<PropertyGroup>
  <AssemblyOriginatorKeyFile>MyKey.snk</AssemblyOriginatorKeyFile>
  <SignAssembly>true</SignAssembly>
  <PublicSign Condition="'$(OS)' != 'Windows_NT'">true</PublicSign>
</PropertyGroup>
```

Other common build options

```
{
  "buildOptions": {
    "warningsAsErrors": true,
    "nowarn": ["CS0168", "CS0219"],
    "xmlDoc": true,
    "preserveCompilationContext": true,
    "outputName": "Different.AssemblyName",
    "debugType": "portable",
    "allowUnsafe": true,
    "define": ["TEST", "OTHERCONDITION"]
  }
}
```

```
<PropertyGroup>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <NoWarn>$(NoWarn);CS0168;CS0219</NoWarn>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>Different.AssemblyName</AssemblyName>
  <DebugType>portable</DebugType>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  <DefineConstants>$(DefineConstants);TEST;OTHERCONDITION</DefineConstants>
</PropertyGroup>
```

packOptions

See also Files.

Common pack options

```
{
  "packOptions": {
    "summary": "numl is a machine learning library intended to ease the use of using standard modeling
techniques for both prediction and clustering.",
    "tags": ["machine learning", "framework"],
    "releaseNotes": "Version 0.9.12-beta",
    "iconUrl": "http://numl.net/images/ico.png",
    "projectUrl": "http://numl.net",
    "licenseUrl": "https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md",
    "requireLicenseAcceptance": false,
    "repository": {
      "type": "git",
      "url": "https://raw.githubusercontent.com/sethjuarez/numl"
    },
    "owners": ["Seth Juarez"]
  }
}
```

```
<PropertyGroup>
  <!-- summary is not migrated from project.json, but you can use the <Description> property for that if
needed. -->
  <PackageTags>machine learning;framework</PackageTags>
  <PackageReleaseNotes>Version 0.9.12-beta</PackageReleaseNotes>
  <PackageIconUrl>http://numl.net/images/ico.png</PackageIconUrl>
  <PackageProjectUrl>http://numl.net</PackageProjectUrl>
  <PackageLicenseUrl>https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md</PackageLicenseUrl>
  <PackageRequireLicenseAcceptance>false</PackageRequireLicenseAcceptance>
  <RepositoryType>git</RepositoryType>
  <RepositoryUrl>https://raw.githubusercontent.com/sethjuarez/numl</RepositoryUrl>
  <!-- owners is not supported in MSBuild -->
</PropertyGroup>
```

There is no equivalent for the `owners` element in MSBuild. For `summary`, you can use the MSBuild `<Description>` property, even though the value of `summary` is not migrated automatically to that property, since that property is mapped to the `description` element.

scripts

```
{
  "scripts": {
    "precompile": "generateCode.cmd",
    "postpublish": [ "obfuscate.cmd", "removeTempFiles.cmd" ]
  }
}
```

Their equivalent in MSBuild are targets:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="generateCode.cmd" />
</Target>

<Target Name="MyPostCompileTarget" AfterTargets="Publish">
  <Exec Command="obfuscate.cmd" />
  <Exec Command="removeTempFiles.cmd" />
</Target>
```

runtimeOptions

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true,
      "System.GC.Concurrent": true,
      "System.GC.RetainVM": true,
      "System.Threading.ThreadPool.MinThreads": 4,
      "System.Threading.ThreadPool.MaxThreads": 25
    }
  }
}
```

All settings in this group, except for the "System.GC.Server" property, are placed into a file called *runtimeconfig.template.json* in the project folder, with options lifted to the root object during the migration process:

```
{
  "configProperties": {
    "System.GC.Concurrent": true,
    "System.GC.RetainVM": true,
    "System.Threading.ThreadPool.MinThreads": 4,
    "System.Threading.ThreadPool.MaxThreads": 25
  }
}
```

The "System.GC.Server" property is migrated into the csproj file:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

However, you can set all those values in the csproj as well as MSBuild properties:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
  <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>
  <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
  <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
</PropertyGroup>
```

shared

```
{
  "shared": "shared/**/*.cs"
}
```

Not supported in csproj. You must instead create include content files in your *.nuspec* file. For more information, see Including content files.

files

In *project.json*, build and pack could be extended to compile and embed from different folders. In MSBuild, this is done using items. The following example is a common conversion:

```
{
  "buildOptions": {
    "compile": {
      "copyToOutput": "notes.txt",
      "include": "../Shared/*.cs",
      "exclude": "../Shared/Not/*.cs"
    },
    "embed": {
      "include": "../Shared/*.resx"
    }
  },
  "packOptions": {
    "include": "Views/",
    "mappings": {
      "some/path/in/project.txt": "in/package.txt"
    }
  },
  "publishOptions": {
    "include": [
      "files/",
      "publishnotes.txt"
    ]
  }
}
```

```
<ItemGroup>
  <Compile Include="..\Shared\*.cs" Exclude="..\Shared\Not\*.cs" />
  <EmbeddedResource Include="..\Shared\*.resx" />
  <Content Include="Views\**\*" PackagePath="%(Identity)" />
  <None Include="some/path/in/project.txt" Pack="true" PackagePath="in/package.txt" />

  <None Include="notes.txt" CopyToOutputDirectory="Always" />
  <!-- CopyToOutputDirectory = { Always, PreserveNewest, Never } -->

  <Content Include="files\**\*" CopyToPublishDirectory="PreserveNewest" />
  <None Include="publishnotes.txt" CopyToPublishDirectory="Always" />
  <!-- CopyToPublishDirectory = { Always, PreserveNewest, Never } -->
</ItemGroup>
```

NOTE

Many of the default globbing patterns are added automatically by the .NET Core SDK. For more information, see Default Compile Item Values.

All MSBuild `ItemGroup` elements support `Include`, `Exclude`, and `Remove`.

Package layout inside the .nupkg can be modified with `PackagePath="path"`.

Except for `Content`, most item groups require explicitly adding `Pack="true"` to be included in the package. `Content` will be put in the *content* folder in a package since the MSBuild `<IncludeContentInPack>` property is set to `true` by default. For more information, see Including content in a package.

`PackagePath="%(Identity)"` is a short way of setting package path to the project-relative file path.

testRunner

xUnit

```
{
  "testRunner": "xunit",
  "dependencies": {
    "dotnet-test-xunit": "<any>"
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />
  <PackageReference Include="xunit" Version="2.2.0-*" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0-*" />
</ItemGroup>
```

MSTest

```
{
  "testRunner": "mstest",
  "dependencies": {
    "dotnet-test-mstest": "<any>"
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.12-*" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.11-*" />
</ItemGroup>
```

See also

- High-level overview of changes in CLI

Migrating from DNX to .NET Core CLI (project.json)

4/8/2019 • 8 minutes to read • Edit Online

Overview

The RC1 release of .NET Core and ASP.NET Core 1.0 introduced DNX tooling. The RC2 release of .NET Core and ASP.NET Core 1.0 moved from DNX to the .NET Core CLI.

As a slight refresher, let's recap what DNX was about. DNX was a runtime and a toolset used to build .NET Core and, more specifically, ASP.NET Core 1.0 applications. It consisted of 3 main pieces:

1. DNVM - an install script for obtaining DNX
2. DNX (Dotnet Execution Runtime) - the runtime that executes your code
3. DNU (Dotnet Developer Utility) - tooling for managing dependencies, building and publishing your applications

With the introduction of the CLI, all of the above are now part of a single toolset. However, since DNX was available in RC1 timeframe, you might have projects that were built using it that you would want to move off to the new CLI tooling.

This migration guide will cover the essentials on how to migrate projects off of DNX and onto .NET Core CLI. If you are just starting a project on .NET Core from scratch, you can freely skip this document.

Main changes in the tooling

There are some general changes in the tooling that should be outlined first.

No more DNVM

DNVM, short for *DotNet Version Manager* was a bash/PowerShell script used to install a DNX on your machine. It helped users get the DNX they need from the feed they specified (or default ones) as well as mark a certain DNX "active", which would put it on the $PATH for the given session. This would allow you to use the various tools.

DNVM was discontinued because its feature set was made redundant by changes coming in the .NET Core CLI tools.

The CLI tools come packaged in two main ways:

1. Native installers for a given platform
2. Install script for other situations (like CI servers)

Given this, the DNVM install features are not needed. But what about the runtime selection features?

You reference a runtime in your `project.json` by adding a package of a certain version to your dependencies. With this change, your application will be able to use the new runtime bits. Getting these bits to your machine is the same as with the CLI: you install the runtime via one of the native installers it supports or via its install script.

Different commands

If you were using DNX, you used some commands from one of its three parts (DNX, DNU or DNVM). With the CLI, some of these commands change, some are not available and some are the same but have slightly different semantics.

The table below shows the mapping between the DNX/DNU commands and their CLI counterparts.

DNX COMMAND	CLI COMMAND	DESCRIPTION
dnx run	dotnet run	Run code from source.
dnu build	dotnet build	Build an IL binary of your code.
dnu pack	dotnet pack	Package up a NuGet package of your code.
dnx [command] (for example, "dnx web")	N/A*	In DNX world, run a command as defined in the project.json.
dnu install	N/A*	In the DNX world, install a package as a dependency.
dnu restore	dotnet restore	Restore dependencies specified in your project.json. (see note)
dnu publish	dotnet publish	Publish your application for deployment in one of the three forms (portable, portable with native and standalone).
dnu wrap	N/A*	In DNX world, wrap a project.json in csproj.
dnu commands	N/A*	In DNX world, manage the globally installed commands.

(*) - these features are not supported in the CLI by design.

DNX features that are not supported

As the table above shows, there are features from the DNX world that we decided not to support in the CLI, at least for the time being. This section will go through the most important ones and outline the rationale behind not supporting them as well as workarounds if you do need them.

Global commands

DNU came with a concept called "global commands". These were, essentially, console applications packaged up as NuGet packages with a shell script that would invoke the DNX you specified to run the application.

The CLI does not support this concept. It does, however, support the concept of adding per-project commands that can be invoked using the familiar `dotnet <command>` syntax.

Installing dependencies

As of v1, the .NET Core CLI tools don't have an `install` command for installing dependencies. In order to install a package from NuGet, you would need to add it as a dependency to your `project.json` file and then run `dotnet restore` (see note).

Running your code

There are two main ways to run your code. One is from source, with `dotnet run`. Unlike `dnx run`, this will not do any in-memory compilation. It will actually invoke `dotnet build` to build your code and then run the built binary.

Another way is using the `dotnet` itself to run your code. This is done by providing a path to your assembly: `dotnet path/to/an/assembly.dll`.

Migrating your DNX project to .NET Core CLI

In addition to using new commands when working with your code, there are three major things left in migrating from DNX:

1. Migrate the `global.json` file if you have it to be able to use CLI.
2. Migrating the project file (`project.json`) itself to the CLI tooling.
3. Migrating off of any DNX APIs to their BCL counterparts.

Changing the global.json file

The `global.json` file acts like a solution file for both the RC1 and RC2 (or later) projects. In order for the CLI tools (as well as Visual Studio) to differentiate between RC1 and later versions, they use the `"sdk": { "version" }` property to make the distinction which project is RC1 or later. If `global.json` doesn't have this node at all, it is assumed to be the latest.

In order to update the `global.json` file, either remove the property or set it to the exact version of the tools that you wish to use, in this case **1.0.0-preview2-003121**:

```
{
    "sdk": {
        "version": "1.0.0-preview2-003121"
    }
}
```

Migrating the project file

The CLI and DNX both use the same basic project system based on `project.json` file. The syntax and the semantics of the project file are pretty much the same, with small differences based on the scenarios. There are also some changes to the schema which you can see in the schema file.

If you are building a console application, you need to add the following snippet to your project file:

```
"buildOptions": {
    "emitEntryPoint": true
}
```

This instructs `dotnet build` to emit an entry point for your application, effectively making your code runnable. If you are building a class library, simply omit the above section. Of course, once you add the above snippet to your `project.json` file, you need to add a static entry point. With the move off DNX, the DI services it provided are no longer available and thus this needs to be a basic .NET entry point: `static void Main()` .

If you have a "commands" section in your `project.json` , you can remove it. Some of the commands that used to exist as DNU commands, such as Entity Framework CLI commands, are being ported to be per-project extensions to the CLI. If you built your own commands that you are using in your projects, you need to replace them with CLI extensions. In this case, the `commands` node in `project.json` needs to be replaced by the `tools` node and it needs to list the tools dependencies.

After these things are done, you need to decide which type of portability you wish for you app. With .NET Core, we have invested into providing a spectrum of portability options that you can choose from. For instance, you may want to have a fully *portable* application or you may want to have a *self-contained* application. The portable application option is more like .NET Framework applications work: it needs a shared component to execute it on the target machine (.NET Core). The self-contained application doesn't require .NET Core to be installed on the target, but you have to produce one application for each OS you wish to support. These portability types and more are discussed in the application portability type document.

Once you make a call on what type of portability you want, you need to change your targeted framework(s). If you

were writing applications for .NET Core, you were most likely using `dnxcore50` as your targeted framework. With the CLI and the changes that the new .NET Standard brought, the framework needs to be one of the following:

1. `netcoreapp1.0` - if you are writing applications on .NET Core (including ASP.NET Core applications)
2. `netstandard1.6` - if you are writing class libraries for .NET Core

If you are using other `dnx` targets, like `dnx451` you will need to change those as well. `dnx451` should be changed to `net451`. Please refer to the .NET Standard topic for more information.

Your `project.json` is now mostly ready. You need to go through your dependencies list and update the dependencies to their newer versions, especially if you are using ASP.NET Core dependencies. If you were using separate packages for BCL APIs, you can use the runtime package as explained in the application portability type document.

Once you are ready, you can try restoring with `dotnet restore` (see note). Depending on the version of your dependencies, you may encounter errors if NuGet cannot resolve the dependencies for one of the targeted frameworks above. This is a "point-in-time" problem; as time progresses, more and more packages will include support for these frameworks. For now, if you run into this, you can use the `imports` statement within the `framework` node to specify to NuGet that it can restore the packages targeting the framework within the "imports" statement. The restoring errors you get in this case should provide enough information to tell you which frameworks you need to import. If you are slightly lost or new to this, in general, specifying `dnxcore50` and `portable-net45+win8` in the `imports` statement should do the trick. The JSON snippet below shows how this looks like:

```
"frameworks": {
    "netcoreapp1.0": {
        "imports": ["dnxcore50", "portable-net45+win8"]
    }
}
```

Running `dotnet build` will show any eventual build errors, though there shouldn't be too many of them. After your code is building and running properly, you can test it out with the runner. Execute `dotnet <path-to-your-assembly>` and see it run.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

.NET Core application deployment

2/23/2019 • 5 minutes to read • Edit Online

You can create three types of deployments for .NET Core applications:

- Framework-dependent deployment. As the name implies, framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. Because .NET Core is already present, your app is also portable between installations of .NET Core. Your app contains only its own code and any third-party dependencies that are outside of the .NET Core libraries. FDDs contain *.dll* files that can be launched by using the dotnet utility from the command line. For example, `dotnet app.dll` runs an application named `app`.
- Self-contained deployment. Unlike FDD, a self-contained deployment (SCD) doesn't rely on the presence of shared components on the target system. All components, including both the .NET Core libraries and the .NET Core runtime, are included with the application and are isolated from other .NET Core applications. SCDs include an executable (such as *app.exe* on Windows platforms for an application named `app`), which is a renamed version of the platform-specific .NET Core host, and a *.dll* file (such as *app.dll*), which is the actual application.
- Framework-dependent executables. Produces an executable that runs on a target platform. Similar to FDDs, framework-dependent executables (FDE) are platform-specific and aren't self-contained. These deployments still rely on the presence of a shared system-wide version of .NET Core to run. Unlike an SCD, your app only contains your code and any third-party dependencies that are outside of the .NET Core libraries. FDEs produce an executable that runs on the target platform.

Framework-dependent deployments (FDD)

For an FDD, you deploy only your app and third-party dependencies. Your app will use the version of .NET Core that's present on the target system. This is the default deployment model for .NET Core and ASP.NET Core apps that target .NET Core.

Why create a framework-dependent deployment?

Deploying an FDD has a number of advantages:

- You don't have to define the target operating systems that your .NET Core app will run on in advance. Because .NET Core uses a common PE file format for executables and libraries regardless of operating system, .NET Core can execute your app regardless of the underlying operating system. For more information on the PE file format, see .NET Assembly File Format.
- The size of your deployment package is small. You only deploy your app and its dependencies, not .NET Core itself.
- Unless overridden, FDDs will use the latest serviced runtime installed on the target system. This allows your application to use the latest patched version of the .NET Core runtime.
- Multiple apps use the same .NET Core installation, which reduces both disk space and memory usage on host systems.

There are also a few disadvantages:

- Your app can run only if the version of .NET Core your app targets, or a later version, is already installed on the host system.

- It's possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.

Self-contained deployments (SCD)

For a self-contained deployment, you deploy your app and any required third-party dependencies along with the version of .NET Core that you used to build the app. Creating an SCD doesn't include the native dependencies of .NET Core on various platforms, so these must be present before the app runs. For more information on version binding at runtime, see the article on version binding in .NET Core.

Starting with NET Core 2.1 SDK (version 2.1.300), .NET Core supports *patch version roll forward*. When you create a self-contained deployment, .NET Core tools automatically include the latest serviced runtime of the .NET Core version that your application targets. (The latest serviced runtime includes security patches and other bug fixes.) The serviced runtime does not have to be present on your build system; it is downloaded automatically from NuGet.org. For more information, including instructions on how to opt out of patch version roll forward, see Self-contained deployment runtime roll forward.

FDD and SCD deployments use separate host executables, so you can sign a host executable for an SCD with your publisher signature.

Why deploy a self-contained deployment?

Deploying a Self-contained deployment has two major advantages:

- You have sole control of the version of .NET Core that is deployed with your app. .NET Core can be serviced only by you.
- You can be assured that the target system can run your .NET Core app, since you're providing the version of .NET Core that it will run on.

It also has a number of disadvantages:

- Because .NET Core is included in your deployment package, you must select the target platforms for which you build deployment packages in advance.
- The size of your deployment package is relatively large, since you have to include .NET Core as well as your app and its third-party dependencies.

  Starting with .NET Core 2.0, you can reduce the size of your deployment on Linux systems by approximately 28 MB by using .NET Core *globalization invariant mode*. Ordinarily, .NET Core on Linux relies on the ICU libraries for globalization support. In invariant mode, the libraries are not included with your deployment, and all cultures behave like the invariant culture.
- Deploying numerous self-contained .NET Core apps to a system can consume significant amounts of disk space, since each app duplicates .NET Core files.

Framework-dependent executables (FDE)

Starting with .NET Core 2.2, you can deploy your app as an FDE, along with any required third-party dependencies. Your app will use the version of .NET Core that's installed on the target system.

Why deploy a framework-dependent executable?

Deploying an FDE has a number of advantages:

- The size of your deployment package is small. You only deploy your app and its dependencies, not .NET Core itself.
- Multiple apps use the same .NET Core installation, which reduces both disk space and memory usage on host systems.

- Your app can be run by calling the published executable without invoking the `dotnet` utility directly.

There are also a few disadvantages:

- Your app can run only if the version of .NET Core your app targets, or a later version, is already installed on the host system.
- It's possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.
- You must publish your app for each target platform.

Step-by-step examples

For step-by-step examples of deploying .NET Core apps with CLI tools, see Deploying .NET Core Apps with CLI Tools. For step-by-step examples of deploying .NET Core apps with Visual Studio, see Deploying .NET Core Apps with Visual Studio.

See also

- Deploying .NET Core Apps with CLI Tools
- Deploying .NET Core Apps with Visual Studio
- Packages, Metapackages and Frameworks
- .NET Core Runtime IDentifier (RID) catalog

Publish .NET Core apps with the CLI

4/9/2019 • 7 minutes to read • Edit Online

This article demonstrates how you can publish your .NET Core application from the command line. .NET Core provides three ways to publish your applications. Framework-dependent deployment produces a cross-platform .dll file that uses the locally installed .NET Core runtime. Framework-dependent executable produces a platform-specific executable that uses the locally installed .NET Core runtime. Self-contained executable produces a platform-specific executable and includes a local copy of the .NET Core runtime.

For an overview of these publishing modes, see .NET Core Application Deployment.

Looking for some quick help on using the CLI? The following table shows some examples of how to publish your app. You can specify the target framework with the `-f <TFM>` parameter or by editing the project file. For more information, see Publishing basics.

PUBLISH MODE	SDK VERSION	COMMAND
Framework-dependent deployment	2.x	`dotnet publish -c Release`
Framework-dependent executable	2.2	`dotnet publish -c Release -r <RID> --self-contained false`
	3.0	`dotnet publish -c Release -r <RID> --self-contained false`
	3.0*	`dotnet publish -c Release`
Self-contained deployment	2.1	`dotnet publish -c Release -r <RID> --self-contained true`
	2.2	`dotnet publish -c Release -r <RID> --self-contained true`
	3.0	`dotnet publish -c Release -r <RID> --self-contained true`

* When using SDK version 3.0, framework-dependent executable is the default publishing mode when running the basic `dotnet publish` command. This only applies when the project targets either **.NET Core 2.1** or **.NET Core 3.0**.

Publishing basics

The `<TargetFramework>` setting of the project file specifies the default target framework when you publish your app. You can change the target framework to any valid Target Framework Moniker (TFM). For example, if your project uses `<TargetFramework>netcoreapp2.2</TargetFramework>`, a binary that targets .NET Core 2.2 is created. The TFM specified in this setting is the default target used by the `dotnet publish` command.

If you want to target more than one framework, you can set the `<TargetFrameworks>` setting to more than one TFM value separated by a semicolon. You can publish one of the frameworks with the `dotnet publish -f <TFM>` command. For example, if you have `<TargetFrameworks>netcoreapp2.1;netcoreapp2.2</TargetFrameworks>` and run `dotnet publish -f netcoreapp2.1`, a binary that targets .NET Core 2.1 is created.

Unless otherwise set, the output directory of the `dotnet publish` command is `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/`. The default **BUILD-CONFIGURATION** mode is **Debug** unless changed with the `-c` parameter. For example, `dotnet publish -c Release -f netcoreapp2.1` publishes to `myfolder/bin/Release/netcoreapp2.1/publish/`.

If you use .NET Core SDK 3.0, the default publish mode for apps that target .NET Core versions 2.1, 2.2, or 3.0 is framework-dependent executable.

If you use .NET Core SDK 2.1, the default publish mode for apps that target .NET Core versions 2.1, 2.2 is framework-dependent deployment.

Native dependencies

If your app has native dependencies, it may not run on a different operating system. For example, if your app uses the native Windows API, it won't run on macOS or Linux. You would need to provide platform-specific code and compile an executable for each platform.

Consider also, if a library you referenced has a native dependency, your app may not run on every platform. However, it's possible a NuGet package you're referencing has included platform-specific versions to handle the required native dependencies for you.

When distributing an app with native dependencies, you may need to use the `dotnet publish -r <RID>` switch to specify the target platform you want to publish for. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.

More information about platform-specific binaries is covered in the Framework-dependent executable and Self-contained deployment sections.

Sample app

You can use either the following app to explore the publishing commands. The app is created by running the following commands in your terminal:

```
mkdir apptest1
cd apptest1
dotnet new console
dotnet add package Figgle
```

The `Program.cs` or `Program.vb` file that is generated by the console template needs to be changed to the following:

```
using System;

namespace apptest1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"));
        }
    }
}
```

```
Imports System

Module Program
    Sub Main(args As String())
        Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"))
    End Sub
End Module
```

When you run the app (`dotnet run`), the following output is displayed:

```
  _   _      _ _        __        __         _     _ _
 | | | | ___| | | ___    \ \      / /__  _ __| | __| | |
 | |_| |/ _ \ | |/ _ \    \ \ /\ / / _ \| '__| |/ _` | |
 |  _  |  __/ | | (_) |    \ V  V / (_) | |  | | (_| |_|
 |_| |_|\___|_|_|\___( )    \_/\_/ \___/|_|  |_|\__,_(_)
                     |/
```

Framework-dependent deployment

For the .NET Core SDK 2.x CLI, framework-dependent deployment (FDD) is the default mode for the basic `dotnet publish` command.

When you publish your app as an FDD, a `<PROJECT-NAME>.dll` file is created in the `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/` folder. To run your app, navigate to the output folder and use the `dotnet <PROJECT-NAME>.dll` command.

Your app is configured to target a specific version of .NET Core. That targeted .NET Core runtime is required to be on the machine where you want to run your app. For example, if your app targets .NET Core 2.2, any machine that your app runs on must have the .NET Core 2.2 runtime installed. As stated in the Publishing basics section, you can edit your project file to change the default target framework or to target more than one framework.

Publishing an FDD creates an app that automatically rolls-forward to the latest .NET Core security patch available on the system that runs the app. For more information on version binding at compile time, see Select the .NET Core version to use.

Framework-dependent executable

For the .NET Core SDK 3.x CLI, framework-dependent executable (FDE) the default mode for the basic `dotnet publish` command. You don't need to specify any other parameters as long as you want to target the current operating system.

In this mode, a platform-specific executable host is created to host your cross-platform app. This mode is similar to FDD as FDD requires a host in the form of the `dotnet` command. The host executable filename varies per platform, and is named something similar to `<PROJECT-FILE>.exe`. You can run this executable directly instead of calling `dotnet <PROJECT-FILE>.dll` which is still an acceptable way to run the app.

Your app is configured to target a specific version of .NET Core. That targeted .NET Core runtime is required to be on the machine where you want to run your app. For example, if your app targets .NET Core 2.2, any machine that your app runs on must have the .NET Core 2.2 runtime installed. As stated in the Publishing basics section, you can edit your project file to change the default target framework or to target more than one framework.

Publishing an FDE creates an app that automatically rolls-forward to the latest .NET Core security patch available on the system that runs the app. For more information on version binding at compile time, see Select the .NET Core version to use.

You must (except for .NET Core 3.x when you target the current platform) use the following switches with the

`dotnet publish` command to publish an FDE:

- `-r <RID>` This switch uses an identifier (RID) to specify the target platform. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.
- `--self-contained false` This switch tells the .NET Core SDK to create an executable as an FDE.

Whenever you use the `-r` switch, the output folder path changes to: `./bin/<BUILD-CONFIGURATION>/<TFM>/<RID>/publish/`

If you use the example app, run `dotnet publish -f netcoreapp2.2 -r win10-x64 --self-contained false`. This command creates the following executable: `./bin/Debug/netcoreapp2.2/win10-x64/publish/apptest1.exe`

> **NOTE**
> You can reduce the total size of your deployment by enabling **globalization invariant mode**. This mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture. For more information about **globalization invariant mode** and how to enable it, see .NET Core Globalization Invariant Mode

Self-contained deployment

When you publish a self-contained deployment (SCD), the .NET Core SDK creates a platform-specific executable. Publishing an SCD includes all required .NET Core files to run your app but it doesn't include the native dependencies of .NET Core. These dependencies must be present on the system before the app runs.

Publishing an SCD creates an app that doesn't roll-forward to the latest available .NET Core security patch. For more information on version binding at compile time, see Select the .NET Core version to use.

You must use the following switches with the `dotnet publish` command to publish an SCD:

- `-r <RID>` This switch uses an identifier (RID) to specify the target platform. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.
- `--self-contained true` This switch tells the .NET Core SDK to create an executable as an SCD.

> **NOTE**
> You can reduce the total size of your deployment by enabling **globalization invariant mode**. This mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture. For more information about **globalization invariant mode** and how to enable it, see .NET Core Globalization Invariant Mode

See also

- .NET Core Application Deployment Overview
- .NET Core Runtime IDentifier (RID) catalog

Deploy .NET Core apps with Visual Studio

1/23/2019 • 15 minutes to read • Edit Online

You can deploy a .NET Core application either as a *framework-dependent deployment*, which includes your application binaries but depends on the presence of .NET Core on the target system, or as a *self-contained deployment*, which includes both your application and .NET Core binaries. For an overview of .NET Core application deployment, see .NET Core Application Deployment.

The following sections show how to use Microsoft Visual Studio to create the following kinds of deployments:

- Framework-dependent deployment
- Framework-dependent deployment with third-party dependencies
- Self-contained deployment
- Self-contained deployment with third-party dependencies

For information on using Visual Studio to develop .NET Core applications, see Prerequisites for .NET Core on Windows.

Framework-dependent deployment

Deploying a framework-dependent deployment with no third-party dependencies simply involves building, testing, and publishing the app. A simple example written in C# illustrates the process.

1. Create the project.

   Select **File** > **New** > **Project**. In the **New Project** dialog, expand your language's (C# or Visual Basic) project categories in the **Installed** project types pane, choose **.NET Core**, and then select the **Console App (.NET Core)** template in the center pane. Enter a project name, such as "FDD", in the **Name** text box. Select the **OK** button.

2. Add the application's source code.

   Open the *Program.cs* or *Program.vb* file in the editor and replace the auto-generated code with the following code. It prompts the user to enter text and displays the individual words entered by the user. It uses the regular expression `\w+` to separate the words in the input text.

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write("\nPress any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. Create a Debug build of your app.

   Select **Build** > **Build Solution**. You can also compile and run the Debug build of your application by selecting **Debug** > **Start Debugging**.

4. Deploy your app.

   After you've debugged and tested the program, create the files to be deployed with your app. To publish from Visual Studio, do the following:

   a. Change the solution configuration from **Debug** to **Release** on the toolbar to build a Release (rather than a Debug) version of your app.

   b. Right-click on the project (not the solution) in **Solution Explorer** and select **Publish**.

   c. In the **Publish** tab, select **Publish**. Visual Studio writes the files that comprise your application to the local file system.

   d. The **Publish** tab now shows a single profile, **FolderProfile**. The profile's configuration settings are shown in the **Summary** section of the tab.

   The resulting files are placed in a directory named `Publish` on Windows and `publish` on Unix systems that is in a subdirectory of your project's *.\bin\release\netcoreapp2.1* subdirectory.

Along with your application's files, the publishing process emits a program database (.pdb) file that contains debugging information about your app. The file is useful primarily for debugging exceptions. You can choose not to package it with your application's files. You should, however, save it in the event that you want to debug the Release build of your app.

Deploy the complete set of application files in any way you like. For example, you can package them in a Zip file,

use a simple `copy` command, or deploy them with any installation package of your choice. Once installed, users can then execute your application by using the `dotnet` command and providing the application filename, such as `dotnet fdd.dll`.

In addition to the application binaries, your installer should also either bundle the shared framework installer or check for it as a prerequisite as part of the application installation. Installation of the shared framework requires Administrator/root access since it is machine-wide.

Framework-dependent deployment with third-party dependencies

Deploying a framework-dependent deployment with one or more third-party dependencies requires that any dependencies be available to your project. The following additional steps are required before you can build your app:

1. Use the **NuGet Package Manager** to add a reference to a NuGet package to your project; and if the package is not already available on your system, install it. To open the package manager, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**.

2. Confirm that `Newtonsoft.Json` is installed on your system and, if it is not, install it. The **Installed** tab lists NuGet packages installed on your system. If `Newtonsoft.Json` is not listed there, select the **Browse** tab and enter "Newtonsoft.Json" in the search box. Select `Newtonsoft.Json` and, in the right pane, select your project before selecting **Install**.

3. If `Newtonsoft.Json` is already installed on your system, add it to your project by selecting your project in the right pane of the **Manage Packages for Solution** tab.

Note that a framework-dependent deployment with third-party dependencies is only as portable as its third-party dependencies. For example, if a third-party library only supports macOS, the app isn't portable to Windows systems. This happens if the third-party dependency itself depends on native code. A good example of this is Kestrel server, which requires a native dependency on libuv. When an FDD is created for an application with this kind of third-party dependency, the published output contains a folder for each Runtime Identifier (RID) that the native dependency supports (and that exists in its NuGet package).

Self-contained deployment without third-party dependencies

Deploying a self-contained deployment with no third-party dependencies involves creating the project, modifying the *csproj* file, building, testing, and publishing the app. A simple example written in C# illustrates the process. You begin by creating, coding, and testing your project just as you would a framework-dependent deployment:

1. Create the project.

   Select **File** > **New** > **Project**. In the **New Project** dialog, expand your language's (C# or Visual Basic) project categories in the **Installed** project types pane, choose **.NET Core**, and then select the **Console App (.NET Core)** template in the center pane. Enter a project name, such as "SCD", in the **Name** text box, and select the **OK** button.

2. Add the application's source code.

   Open the *Program.cs* or file in your editor, and replace the auto-generated code with the following code. It prompts the user to enter text and displays the individual words entered by the user. It uses the regular expression `\w+` to separate the words in the input text.

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write("\nPress any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. Determine whether you want to use globalization invariant mode.

   Particularly if your app targets Linux, you can reduce the total size of your deployment by taking advantage of globalization invariant mode. Globalization invariant mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture.

   To enable invariant mode, right-click on your project (not the solution) in **Solution Explorer**, and select **Edit SCD.csproj** or **Edit SCD.vbproj**. Then add the following highlighted lines to the file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Globalization.Invariant" Value="true" />
  </ItemGroup>
</Project>
```

1. Create a Debug build of your application.

   Select **Build** > **Build Solution**. You can also compile and run the Debug build of your application by selecting **Debug** > **Start Debugging**. This debugging step lets you identify problems with your application when it's running on your host platform. You still will have to test it on each of your target platforms.

   If you've enabled globalization invariant mode, be particularly sure to test whether the absence of culture-sensitive data is suitable for your application.

Once you've finished debugging, you can publish your self-contained deployment:

- Visual Studio 15.6 and earlier
- Visual Studio 15.7 and later

After you've debugged and tested the program, create the files to be deployed with your app for each platform that it targets.

To publish your app from Visual Studio, do the following:

1. Define the platforms that your app will target.

   a. Right-click on your project (not the solution) in **Solution Explorer** and select **Edit SCD.csproj**.

   b. Create a `<RuntimeIdentifiers>` tag in the `<PropertyGroup>` section of your *csproj* file that defines the platforms your app targets, and specify the runtime identifier (RID) of each platform that you target. Note that you also need to add a semicolon to separate the RIDs. See Runtime IDentifier catalog for a list of runtime identifiers.

   For example, the following example indicates that the app runs on 64-bit Windows 10 operating systems and the 64-bit OS X Version 10.11 operating system.

   ```
   <PropertyGroup>
      <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
   </PropertyGroup>
   ```

   Note that the `<RuntimeIdentifiers>` element can go into any `<PropertyGroup>` that you have in your *csproj* file. A complete sample *csproj* file appears later in this section.

2. Publish your app.

   After you've debugged and tested the program, create the files to be deployed with your app for each platform that it targets.

   To publish your app from Visual Studio, do the following:

   a. Change the solution configuration from **Debug** to **Release** on the toolbar to build a Release (rather than a Debug) version of your app.

   b. Right-click on the project (not the solution) in **Solution Explorer** and select **Publish**.

   c. In the **Publish** tab, select **Publish**. Visual Studio writes the files that comprise your application to the local file system.

   d. The **Publish** tab now shows a single profile, **FolderProfile**. The profile's configuration settings are shown in the **Summary** section of the tab. **Target Runtime** identifies which runtime has been published, and **Target Location** identifies where the files for the self-contained deployment were written.

   e. Visual Studio by default writes all published files to a single directory. For convenience, it's best to create separate profiles for each target runtime and to place published files in a platform-specific directory. This involves creating a separate publishing profile for each target platform. So now rebuild the application for each platform by doing the following:

      a. Select **Create new profile** in the **Publish** dialog.

      b. In the **Pick a publish target** dialog, change the **Choose a folder** location to *bin\Release\PublishOutput\win10-x64*. Select **OK**.

      c. Select the new profile (**FolderProfile1**) in the list of profiles, and make sure that the **Target Runtime** is `win10-x64`. If it isn't, select **Settings**. In the **Profile Settings** dialog, change the

Target Runtime to `win10-x64` and select **Save**. Otherwise, select **Cancel**.

d. Select **Publish** to publish your app for 64-bit Windows 10 platforms.

e. Follow the previous steps again to create a profile for the `osx.10.11-x64` platform. The **Target Location** is *bin\Release\PublishOutput\osx.10.11-x64*, and the **Target Runtime** is `osx.10.11-x64`. The name that Visual Studio assigns to this profile is **FolderProfile2**.

Note that each target location contains the complete set of files (both your app files and all .NET Core files) needed to launch your app.

Along with your application's files, the publishing process emits a program database (.pdb) file that contains debugging information about your app. The file is useful primarily for debugging exceptions. You can choose not to package it with your application's files. You should, however, save it in the event that you want to debug the Release build of your app.

Deploy the published files in any way you like. For example, you can package them in a Zip file, use a simple `copy` command, or deploy them with any installation package of your choice.

The following is the complete *csproj* file for this project.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

Self-contained deployment with third-party dependencies

Deploying a self-contained deployment with one or more third-party dependencies involves adding the dependencies. The following additional steps are required before you can build your app:

1. Use the **NuGet Package Manager** to add a reference to a NuGet package to your project; and if the package is not already available on your system, install it. To open the package manager, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**.

2. Confirm that `Newtonsoft.Json` is installed on your system and, if it is not, install it. The **Installed** tab lists NuGet packages installed on your system. If `Newtonsoft.Json` is not listed there, select the **Browse** tab and enter "Newtonsoft.Json" in the search box. Select `Newtonsoft.Json` and, in the right pane, select your project before selecting **Install**.

3. If `Newtonsoft.Json` is already installed on your system, add it to your project by selecting your project in the right pane of the **Manage Packages for Solution** tab.

The following is the complete *csproj* file for this project:

- Visual Studio 15.6 and earlier
- Visual Studio 15.7 and later

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.2" />
  </ItemGroup>
</Project>
```

When you deploy your application, any third-party dependencies used in your app are also contained with your application files. Third-party libraries aren't required on the system on which the app is running.

Note that you can only deploy a self-contained deployment with a third-party library to platforms supported by that library. This is similar to having third-party dependencies with native dependencies in your framework-dependent deployment, where the native dependencies won't exist on the target platform unless they were previously installed there.

See also

- .NET Core Application Deployment
- .NET Core Runtime IDentifier (RID) catalog

How to create a NuGet package with .NET Core command-line interface (CLI) tools

5/15/2019 • 2 minutes to read • Edit Online

> **NOTE**
> The following shows command-line samples using Unix. The `dotnet pack` command as shown here works the same way on Windows.

.NET Standard and .NET Core libraries are expected to be distributed as NuGet packages. This is in fact how all of the .NET Standard libraries are distributed and consumed. This is most easily done with the `dotnet pack` command.

Imagine that you just wrote an awesome new library that you would like to distribute over NuGet. You can create a NuGet package with cross platform tools to do exactly that! The following example assumes a library called **SuperAwesomeLibrary** which targets `netstandard1.0`.

If you have transitive dependencies; that is, a project which depends on another package, you'll need to make sure to restore packages for your entire solution with the `dotnet restore` command before creating a NuGet package. Failing to do so will result in the `dotnet pack` command to not work properly.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

After ensuring packages are restored, you can navigate to the directory where a library lives:

```
cd src/SuperAwesomeLibrary
```

Then it's just a single command from the command line:

```
dotnet pack
```

Your `/bin/Debug` folder will now look like this:

```
$ ls bin/Debug

netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

Note that this will produce a package which is capable of being debugged. If you want to build a NuGet package with release binaries, all you need to do is add the `--configuration` (or `-c`) switch and use `release` as the argument.

```
dotnet pack --configuration release
```

Your `/bin` folder will now have a `release` folder containing your NuGet package with release binaries:

```
$ ls bin/release

netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

And now you have the necessary files to publish a NuGet package!

Don't confuse `dotnet pack` with `dotnet publish`

It is important to note that at no point is the `dotnet publish` command involved. The `dotnet publish` command is for deploying applications with all of their dependencies in the same bundle -- not for generating a NuGet package to be distributed and consumed via NuGet.

See also

- Quickstart: Create and publish a package

Self-contained deployment runtime roll forward

12/20/2018 • 2 minutes to read • Edit Online

.NET Core self-contained application deployments include both the .NET Core libraries and the .NET Core runtime. Starting in .NET Core 2.1 SDK (version 2.1.300), a self-contained application deployment publishes the highest patch runtime on your machine. By default, `dotnet publish` for a self-contained deployment selects the latest version installed as part of the SDK on the publishing machine. This enables your deployed application to run with security fixes (and other fixes) available during `publish`. The application must be re-published to obtain a new patch. Self-contained applications are created by specifying `-r <RID>` on the `dotnet publish` command or by specifying the runtime identifier (RID) in the project file (csproj / vbproj) or on the command line.

Patch version roll forward overview

`restore`, `build` and `publish` are `dotnet` commands that can run separately. The runtime choice is part of the `restore` operation, not `publish` or `build`. If you call `publish`, the latest patch version will be chosen. If you call `publish` with the `--no-restore` argument, then you may not get the desired patch version because a prior `restore` may not have been executed with the new self-contained application publishing policy. In this case, a build error is generated with text similar to the following:

"The project was restored using Microsoft.NETCore.App version 2.0.0, but with current settings, version 2.0.6 would be used instead. To resolve this issue, make sure the same settings are used for restore and for subsequent operations such as build or publish. Typically this issue can occur if the RuntimeIdentifier property is set during build or publish but not during restore."

> **NOTE**
> `restore` and `build` can be run implicitly as part of another command, like `publish`. When run implicitly as part of another command, they are provided with additional context so that the right artifacts are produced. When you `publish` with a runtime (for example, `dotnet publish -r linux-x64`), the implicit `restore` restores packages for the linux-x64 runtime. If you call `restore` explicitly, it does not restore runtime packages by default, because it doesn't have that context.

How to avoid restore during publish

Running `restore` as part of the `publish` operation may be undesirable for your scenario. To avoid `restore` during `publish` while creating self-contained applications, do the following:

- Set the `RuntimeIdentifiers` property to a semicolon-separated list of all the RIDs to be published.
- Set the `TargetLatestRuntimePatch` property to `true`.

No-restore argument with dotnet publish options

If you want to create both self-contained applications and framework-dependent applications with the same project file, and you want to use the `--no-restore` argument with `dotnet publish`, then choose one of the following:

1. Prefer the framework-dependent behavior. If the application is framework-dependent, this is the default behavior. If the application is self-contained, and can use an unpatched 2.1.0 local runtime, set the `TargetLatestRuntimePatch` to `false` in the project file.

2. Prefer the self-contained behavior. If the application is self-contained, this is the default behavior. If the

application is framework-dependent, and requires the latest patch installed, set `TargetLatestRuntimePatch` to `true` in the project file.

3. Take explicit control of the runtime framework version by setting `RuntimeFrameworkVersion` to the specific patch version in the project file.

Runtime package store

1/23/2019 • 5 minutes to read • Edit Online

Starting with .NET Core 2.0, it's possible to package and deploy apps against a known set of packages that exist in the target environment. The benefits are faster deployments, lower disk space usage, and improved startup performance in some cases.

This feature is implemented as a *runtime package store*, which is a directory on disk where packages are stored (typically at */usr/local/share/dotnet/store* on macOS/Linux and *C:/Program Files/dotnet/store* on Windows). Under this directory, there are subdirectories for architectures and target frameworks. The file layout is similar to the way that NuGet assets are laid out on disk:

```
\dotnet
    \store
        \x64
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
        \x86
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
```

A *target manifest* file lists the packages in the runtime package store. Developers can target this manifest when publishing their app. The target manifest is typically provided by the owner of the targeted production environment.

Preparing a runtime environment

The administrator of a runtime environment can optimize apps for faster deployments and lower disk space use by building a runtime package store and the corresponding target manifest.

The first step is to create a *package store manifest* that lists the packages that compose the runtime package store. This file format is compatible with the project file format (*csproj*).

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="<NUGET_PACKAGE>" Version="<VERSION>" />
    <!-- Include additional packages here -->
  </ItemGroup>
</Project>
```

Example

The following example package store manifest (*packages.csproj*) is used to add `Newtonsoft.Json` and `Moq` to a runtime package store:

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.3" />
    <PackageReference Include="Moq" Version="4.7.63" />
  </ItemGroup>
</Project>
```

Provision the runtime package store by executing `dotnet store` with the package store manifest, runtime, and framework:

```
dotnet store --manifest <PATH_TO_MANIFEST_FILE> --runtime <RUNTIME_IDENTIFIER> --framework <FRAMEWORK>
```

Example

```
dotnet store --manifest packages.csproj --runtime win10-x64 --framework netcoreapp2.0 --framework-version
2.0.0
```

You can pass multiple target package store manifest paths to a single `dotnet store` command by repeating the option and path in the command.

By default, the output of the command is a package store under the *.dotnet/store* subdirectory of the user's profile. You can specify a different location using the `--output <OUTPUT_DIRECTORY>` option. The root directory of the store contains a target manifest *artifact.xml* file. This file can be made available for download and be used by app authors who want to target this store when publishing.

Example

The following *artifact.xml* file is produced after running the previous example. Note that `Castle.Core` is a dependency of `Moq`, so it's included automatically and appears in the *artifacts.xml* manifest file.

```
<StoreArtifacts>
  <Package Id="Newtonsoft.Json" Version="10.0.3" />
  <Package Id="Castle.Core" Version="4.1.0" />
  <Package Id="Moq" Version="4.7.63" />
</StoreArtifacts>
```

Publishing an app against a target manifest

If you have a target manifest file on disk, you specify the path to the file when publishing your app with the `dotnet publish` command:

```
dotnet publish --manifest <PATH_TO_MANIFEST_FILE>
```

Example

```
dotnet publish --manifest manifest.xml
```

You deploy the resulting published app to an environment that has the packages described in the target manifest. Failing to do so results in the app failing to start.

Specify multiple target manifests when publishing an app by repeating the option and path (for example, `--manifest manifest1.xml --manifest manifest2.xml`). When you do so, the app is trimmed for the union of packages specified in the target manifest files provided to the command.

Specifying target manifests in the project file

An alternative to specifying target manifests with the `dotnet publish` command is to specify them in the project file as a semicolon-separated list of paths under a **<TargetManifestFiles>** tag.

```
<PropertyGroup>
  <TargetManifestFiles>manifest1.xml;manifest2.xml</TargetManifestFiles>
</PropertyGroup>
```

Specify the target manifests in the project file only when the target environment for the app is well-known, such as for .NET Core projects. This isn't the case for open-source projects. The users of an open-source project typically deploy it to different production environments. These production environments generally have different sets of packages pre-installed. You can't make assumptions about the target manifest in such environments, so you should use the `--manifest` option of `dotnet publish`.

ASP.NET Core implicit store

The ASP.NET Core implicit store applies only to ASP.NET Core 2.0. We strongly recommend applications use ASP.NET Core 2.1 and later, which does **not** use the implicit store. ASP.NET Core 2.1 and later use the shared framework.

The runtime package store feature is used implicitly by an ASP.NET Core app when the app is deployed as a framework-dependent deployment (FDD) app. The targets in `Microsoft.NET.Sdk.Web` include manifests referencing the implicit package store on the target system. Additionally, any FDD app that depends on the `Microsoft.AspNetCore.All` package results in a published app that contains only the app and its assets and not the packages listed in the `Microsoft.AspNetCore.All` metapackage. It's assumed that those packages are present on the target system.

The runtime package store is installed on the host when the .NET Core SDK is installed. Other installers may provide the runtime package store, including Zip/tarball installations of the .NET Core SDK, `apt-get`, Red Hat Yum, the .NET Core Windows Server Hosting bundle, and manual runtime package store installations.

When deploying a framework-dependent deployment (FDD) app, make sure that the target environment has the .NET Core SDK installed. If the app is deployed to an environment that doesn't include ASP.NET Core, you can opt out of the implicit store by specifying **<PublishWithAspNetCoreTargetManifest>** set to `false` in the project file as in the following example:

```
<PropertyGroup>
  <PublishWithAspNetCoreTargetManifest>false</PublishWithAspNetCoreTargetManifest>
</PropertyGroup>
```

> **NOTE**
>
> For self-contained deployment (SCD) apps, it's assumed that the target system doesn't necessarily contain the required manifest packages. Therefore, **<PublishWithAspNetCoreTargetManifest>** cannot be set to `true` for an SCD app.

If you deploy an application with a manifest dependency that's present in the deployment (the assembly is present in the *bin* folder), the runtime package store *isn't used* on the host for that assembly. The *bin* folder assembly is used regardless of its presence in the runtime package store on the host.

The version of the dependency indicated in the manifest must match the version of the dependency in the runtime package store. If you have a version mismatch between the dependency in the target manifest and the version that exists in the runtime package store and the app doesn't include the required version of the package in its

deployment, the app fails to start. The exception includes the name of the target manifest that called for the runtime package store assembly, which helps you troubleshoot the mismatch.

When the deployment is *trimmed* on publish, only the specific versions of the manifest packages you indicate are withheld from the published output. The packages at the versions indicated must be present on the host for the app to start.

See also

- dotnet-publish
- dotnet-store

Introduction to .NET and Docker

7/30/2019 • 3 minutes to read • Edit Online

.NET Core can easily run in a Docker container. Containers provide a lightweight way to isolate your application from the rest of the host system, sharing just the kernel, and using resources given to your application. If you're unfamiliar with Docker, it's highly recommended that you read through Docker's overview documentation.

For more information about how to install Docker, see the download page for Docker Desktop: Community Edition.

Docker basics

There are a few concepts you should be familiar with. The Docker client has a command line interface program you use to manage images and containers. As previously stated, you should take the time to read through the Docker overview documentation.

Images

An image is an ordered collection of filesystem changes that form the basis of a container. The image doesn't have a state and is read-only. Much the time an image is based on another image, but with some customization. For example, when you create an new image for your application, you would base it on an existing image that already contains the .NET Core runtime.

Because containers are created from images, images have a set of run parameters (such as a starting executable) that run when the container starts.

Containers

A container is a runnable instance of an image. As you build your image, you deploy your application and dependencies. Then, multiple containers can be instantiated, each isolated from one another. Each container instance has its own filesystem, memory, and network interface.

Registries

Container registries are a collection of image repositories. You can base your images on a registry image. You can create containers directly from an image in a registry. The relationship between Docker containers, images, and registries is an important concept when architecting and building containerized applications or microservices. This approach greatly shortens the time between development and deployment.

Docker has a public registry hosted at the Docker Hub that you can use. .NET Core related images are listed at the Docker Hub.

The Microsoft Container Registry (MCR) is the official source of Microsoft-provided container images. The MCR is built on Azure CDN to provide globally-replicated images. However, the MCR does not have a public-facing website and the primary way to learn about Microsoft-provided container images is through the Microsoft Docker Hub pages.

Dockerfile

A **Dockerfile** is a file that defines a set of instructions that creates an image. Each instruction in the **Dockerfile** creates a layer in the image. For the most part, when you rebuild the image only the layers that have changed are rebuilt. The **Dockerfile** can be distributed to others and allows them to recreate to create a new image in the same manner you created it. While this allows you to distribute the *instructions* on how to create the image, the main way to distribute your image is to publish it to a registry.

.NET Core images

Official .NET Core Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the Microsoft .NET Core Docker Hub repository. Each repository contains images for different combinations of the .NET (SDK or Runtime) and OS that you can use.

Microsoft provides images that are tailored for specific scenarios. For example, the ASP.NET Core repository provides images that are built for running ASP.NET Core apps in production.

Azure services

Various Azure services support containers. You create a Docker image for your application and deploy it to one of the following services:

- Azure Kubernetes Service (AKS)
  Scale and orchestrate Linux containers using Kubernetes.
- Azure App Service
  Deploy web apps or APIs using Linux containers in a PaaS environment.
- Azure Container Instances
  Host your container in the cloud without any higher-level management services.
- Azure Batch
  Run repetitive compute jobs using containers.
- Azure Service Fabric
  Lift, shift, and modernize .NET applications to microservices using Windows Server containers
- Azure Container Registry
  Store and manage container images across all types of Azure deployments.

Next steps

- Learn how to containerize a .NET Core application.
- Learn how to containerize an ASP.NET Core application.
- Try the Learn ASP.NET Core Microservice tutorial.
- Learn about Container Tools in Visual Studio

Tutorial: Containerize a .NET Core app

7/19/2019 • 11 minutes to read • Edit Online

This tutorial teaches you how to build a Docker image that contains your .NET Core application. The image can be used to create containers for your local development environment, private cloud, or public cloud.

You'll learn to:

- Create and publish a simple .NET Core app
- Create and configure a Dockerfile for .NET Core
- Build a Docker image
- Create and run a Docker container

You'll understand the Docker container build and deploy tasks for a .NET Core application. The *Docker platform* uses the *Docker engine* to quickly build and package apps as *Docker images*. These images are written in the *Dockerfile* format to be deployed and run in a layered container.

Prerequisites

Install the following prerequisites:

- .NET Core 2.2 SDK
  If you have .NET Core installed, use the `dotnet --info` command to determine which SDK you're using.
- Docker Community Edition
- A temporary working folder for the *Dockerfile* and .NET Core example app. In this tutorial, the name `docker-working` is used as the working folder.

Use SDK version 2.2

If you're using an SDK that is newer, like 3.0, make sure that your app is forced to use the 2.2 SDK. Create a file named `global.json` in your working folder and paste in the following json code:

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

Save this file. The presence of file will force .NET Core to use version 2.2 for any `dotnet` command called from this folder and below.

Create .NET Core app

You need a .NET Core app that the Docker container will run. Open your terminal, create a working folder if you haven't already, and enter it. In the working folder, run the following command to create a new project in a subdirectory named app:

```
dotnet new console -o app -n myapp
```

Your folder tree will look like the following:

```
docker-working
|   global.json
|
└──app
    |   myapp.csproj
    |   Program.cs
    |
    └──obj
            myapp.csproj.nuget.cache
            myapp.csproj.nuget.g.props
            myapp.csproj.nuget.g.targets
            project.assets.json
```

The `dotnet new` command creates a new folder named *app* and generates a "Hello World" app. Enter the *app* folder and run the command `dotnet run`. You'll see the following output:

```
> dotnet run
Hello World!
```

The default template creates an app that prints to the terminal and then exits. For this tutorial, you'll use an app that loops indefinitely. Open the **Program.cs** file in a text editor. It should currently look like the following code:

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Replace the file with the following code that counts numbers every second:

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) : -1;
            while(max == -1 || counter < max)
            {
                counter++;
                Console.WriteLine($"Counter: {counter}");
                System.Threading.Tasks.Task.Delay(1000).Wait();
            }
        }
    }
}
```

Save the file and test the program again with `dotnet run`. Remember that this app runs indefinitely. Use the cancel command CTRL + C to stop it. You'll see the following output:

```
> dotnet run
Counter: 1
Counter: 2
Counter: 3
Counter: 4
^C
```

If you pass a number on the command line to the app, it will only count up to that amount and then exit. Try it with `dotnet run -- 5` to count to five.

> **NOTE**
> Any parameters after `--` are not passed to the `dotnet run` command and instead are passed to your application.

Publish .NET Core app

Before you add your .NET Core app to the Docker image, publish it. You want to make sure that the container runs the published version of the app when it's started.

From the working folder, enter the **app** folder with the example source code and run the following command:

```
dotnet publish -c Release
```

This command compiles your app to the **publish** folder. The path to the **publish** folder from the working folder should be `.\app\bin\Release\netcoreapp2.2\publish\`

Get a directory listing of the publish folder to verify that the **myapp.dll** was created. From the **app** folder, run one of the following commands:

```
> dir bin\Release\netcoreapp2.2\publish
 Directory of C:\docker-working\app\bin\Release\netcoreapp2.2\publish

04/05/2019  11:00 AM    <DIR>          .
04/05/2019  11:00 AM    <DIR>          ..
04/05/2019  11:00 AM               447 myapp.deps.json
04/05/2019  11:00 AM             4,608 myapp.dll
04/05/2019  11:00 AM               448 myapp.pdb
04/05/2019  11:00 AM               154 myapp.runtimeconfig.json
```

```
me@DESKTOP:/docker-working/app$ ls bin/Release/netcoreapp2.2/publish
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
```

Create the Dockerfile

The *Dockerfile* file is used by the `docker build` command to create a container image. This file is a plaintext file named *Dockerfile* that does not have an extension.

In your terminal, navigate up a directory to the working folder you created at the start. Create a file named *Dockerfile* in your working folder and open it in a text editor. Add the following command as the first line of the file:

```
FROM mcr.microsoft.com/dotnet/core/runtime:2.2
```

The `FROM` command tells Docker to pull down the image tagged **2.2** from the

mcr.microsoft.com/dotnet/core/runtime repository. Make sure that you pull the .NET Core runtime that matches the runtime targeted by your SDK. For example, the app created in the previous section used the .NET Core 2.2 SDK and created an app that targeted .NET Core 2.2. So the base image referred to in the *Dockerfile* is tagged with **2.2**.

Save the *Dockerfile* file. The directory structure of the working folder should look like the following. Some of the deeper-level files and folders have been cut to save space in the article:

```
docker-working
│   Dockerfile
│   global.json
│
└───app
    │   myapp.csproj
    │   Program.cs
    │
    ├───bin
    │   └───Release
    │       └───netcoreapp2.2
    │           └───publish
    │                   myapp.deps.json
    │                   myapp.dll
    │                   myapp.pdb
    │                   myapp.runtimeconfig.json
    │
    └───obj
```

From your terminal, run the following command:

```
docker build -t myimage -f Dockerfile .
```

Docker will process each line in the *Dockerfile*. The `.` in the `docker build` command tells Docker to use the current folder to find a *Dockerfile*. This command builds the image and creates a local repository named **myimage** that points to that image. After this command finishes, run `docker images` to see a list of images installed:

```
> docker images
REPOSITORY                                  TAG                 IMAGE ID            CREATED             SIZE
mcr.microsoft.com/dotnet/core/runtime       2.2                 d51bb4452469        2 days ago          314MB
myimage                                     latest              d51bb4452469        2 days ago          314MB
```

Notice that the two images share the same **IMAGE ID** value. The value is the same between both images because the only command in the *Dockerfile* was to base the new image on an existing image. Let's add two commands to the *Dockerfile*. Each command creates a new image layer with the final command representing the image the **myimage** repository will point to.

```
COPY app/bin/Release/netcoreapp2.2/publish/ app/

ENTRYPOINT ["dotnet", "app/myapp.dll"]
```

The `COPY` command tells Docker to copy the specified folder on your computer to a folder in the container. In this example, the **publish** folder is copied to a folder named **app** in the container.

The next command, `ENTRYPOINT`, tells Docker to configure the container to run as an executable. When the container starts, the `ENTRYPOINT` command runs. When this command ends, the container will automatically stop.

From your terminal, run `docker build -t myimage -f Dockerfile .` and when that command finishes, run `docker images`.

```
> docker build -t myimage -f Dockerfile .
Sending build context to Docker daemon  819.7kB
Step 1/3 : FROM mcr.microsoft.com/dotnet/core/runtime:2.2
 ---> d51bb4452469
Step 2/3 : COPY app/bin/Release/netcoreapp2.2/publish/ app/
 ---> a1e98ac62017
Step 3/3 : ENTRYPOINT ["dotnet", "app/myapp.dll"]
 ---> Running in f34da5c18e7c
Removing intermediate container f34da5c18e7c
 ---> ddcc6646461b
Successfully built ddcc6646461b
Successfully tagged myimage:latest

> docker images
REPOSITORY                                TAG                 IMAGE ID            CREATED             SIZE
myimage                                   latest              ddcc6646461b        10 seconds ago      314MB
mcr.microsoft.com/dotnet/core/runtime   2.2                 d51bb4452469        2 days ago          314MB
```

Each command in the *Dockerfile* generated a layer and created an **IMAGE ID**. The final **IMAGE ID** (yours will be different) is **ddcc6646461b** and next you'll create a container based on this image.

Create a container

Now that you have an image that contains your app, you can create a container. You can create a container in two ways. First, create a new container that is stopped.

```
> docker create myimage
0e8f3c2ca32ce773712a5cca38750f41259a4e54e04bdf0946087e230ad7066c
```

The `docker create` command from above will create a container based on the **myimage** image. The output of that command shows you the **CONTAINER ID** (yours will be different) of the created container. To see a list of *all* containers, use the `docker ps -a` command:

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS        PORTS
NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   4 seconds ago       Created
boring_matsumoto
```

Manage the container

Each container is assigned a random name that you can use to refer to that container instance. For example, the container that was created automatically chose the name **boring_matsumoto** (yours will be different) and that name can be used to start the container. You override the automatic name with a specific one by using the `docker create --name` parameter.

The following example uses the `docker start` command to start the container, and then uses the `docker ps` command to only show containers that are running:

```
> docker start boring_matsumoto
boring_matsumoto

> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS         PORTS
NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   7 minutes ago       Up 8 seconds
boring_matsumoto
```

Similarly, the `docker stop` command will stop the container. The following example uses the `docker stop` command to stop the container, and then uses the `docker ps` command to show that no containers are running.

```
> docker stop boring_matsumoto
boring_matsumoto

> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS      PORTS    NAMES
```

Connect to a container

After a container is running, you can connect to it to see the output. Use the `docker start` and `docker attach` commands to start the container and peek at the output stream. In this example, the `CTRL + C` command is used to detach from the running container. This may actually end the process in the container, which will stop the container. The `--sig-proxy=false` parameter ensures that `CTRL + C` won't stop the process in the container.

After you detach from the container, reattach to verify that it's still running and counting.

```
> docker start boring_matsumoto
boring_matsumoto

> docker attach --sig-proxy=false boring_matsumoto
Counter: 7
Counter: 8
Counter: 9
^C

> docker attach --sig-proxy=false boring_matsumoto
Counter: 17
Counter: 18
Counter: 19
^C
```

Delete a container

For the purposes of this article you don't want containers just hanging around doing nothing. Delete the container you previously created. If the container is running, stop it.

```
> docker stop boring_matsumoto
```

The following example lists all containers. It then uses the `docker rm` command to delete the container, and then checks a second time for any running containers.

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS      PORTS    NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   19 minutes ago      Exited
boring_matsumoto

> docker rm boring_matsumoto
boring_matsumoto

> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS      PORTS     NAMES
```

Single run

Docker provides the `docker run` command to create and run the container as a single command. This command eliminates the need to run `docker create` and then `docker start`. You can also set this command to automatically delete the container when the container stops. For example, use `docker run -it --rm` to do two things, first,

automatically use the current terminal to connect to the container, and then when the container finishes, remove it:

```
> docker run -it --rm myimage
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

With `docker run -it`, the CTRL + C command will stop process that is running in the container, which in turn, stops the container. Since the `--rm` parameter was provided, the container is automatically deleted when the process is stopped. Verify that it does not exist:

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS    PORTS   NAMES
```

Change the ENTRYPOINT

The `docker run` command also lets you modify the `ENTRYPOINT` command from the *Dockerfile* and run something else, but only for that container. For example, use the following command to run `bash` or `cmd.exe`. Edit the command as necessary.

Windows

In this example, `ENTRYPOINT` is changed to `cmd.exe`. CTRL + C is pressed to end the process and stop the container.

```
> docker run -it --rm --entrypoint "cmd.exe" myimage

Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 3005-1E84

 Directory of C:\

04/09/2019  08:46 AM    <DIR>          app
03/07/2019  10:25 AM             5,510 License.txt
04/02/2019  01:35 PM    <DIR>          Program Files
04/09/2019  01:06 PM    <DIR>          Users
04/02/2019  01:35 PM    <DIR>          Windows
               1 File(s)          5,510 bytes
               4 Dir(s)  21,246,517,248 bytes free

C:\>^C
```

Linux

In this example, `ENTRYPOINT` is changed to `bash`. The `quit` command is run which ends the process and stop the container.

```
root@user:~# docker run -it --rm --entrypoint "bash" myimage
root@8515e897c893:/# ls app
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
root@8515e897c893:/# exit
exit
```

Essential commands

Docker has many different commands that cover what you want to do with your container and images. These Docker commands are essential to managing your containers:

- docker build
- docker run
- docker ps
- docker stop
- docker rm
- docker rmi
- docker image

Clean up resources

During this tutorial you created containers and images. If you want, delete these resources. Use the following commands to

1. List all containers

   ```
   > docker ps -a
   ```

2. Stop containers that are running. The `CONTAINER_NAME` represents the name automatically assigned to the container.

   ```
   > docker stop CONTAINER_NAME
   ```

3. Delete the container

   ```
   > docker rm CONTAINER_NAME
   ```

Next, delete any images that you no longer want on your machine. Delete the image created by your *Dockerfile* and then delete the .NET Core image the *Dockerfile* was based on. You can use the **IMAGE ID** or the **REPOSITORY:TAG** formatted string.

```
docker rmi myimage:latest
docker rmi mcr.microsoft.com/dotnet/core/runtime:2.2
```

Use the `docker images` command to see a list of images installed.

> **NOTE**
> Image files can be large. Typically, you would remove temporary containers you created while testing and developing your app. You usually keep the base images with the runtime installed if you plan on building other images based on that runtime.

Next steps

- Try the ASP.NET Core Microservice Tutorial.
- Review the Azure services that support containers.
- Read about Dockerfile commands.

- Explore the Container Tools for Visual Studio

What diagnostic tools are available in .NET Core?

8/13/2019 • 2 minutes to read • Edit Online

Software doesn't always behave as you would expect, but .NET Core has tools and APIs that will help you diagnose these issues quickly and effectively.

This article helps you find the various tools you need.

Managed debuggers

Managed debuggers allow you to interact with your program. Pausing, incrementally executing, examining, and resuming gives you insight into the behavior of your code. A debugger is the first choice for diagnosing functional problems that can be easily reproduced.

Logging and tracing

Logging and tracing are related techniques. They refer to instrumenting code to create log files. The files record the details of what a program does. These details can be used to diagnose the most complex problems. When combined with time stamps, these techniques are also valuable in performance investigations.

Unit testing

Unit testing is a key component of continuous integration and deployment of high-quality software. Unit tests are designed to give you an early warning when you break something.

.NET Core managed debuggers

8/13/2019 • 2 minutes to read • Edit Online

Debuggers allow programs to be paused or executed step-by-step. When paused, the current state of the process can be viewed. By stepping through key sections, you gain understanding of your code and why it produces the result that it does.

Microsoft provides debuggers for managed code in **Visual Studio** and **Visual Studio Code**.

Visual Studio managed debugger

Visual Studio is an integrated development environment with the most comprehensive debugger available. Visual Studio is an excellent choice for developers working on Windows.

- Tutorial - Debugging a .NET Core application on Windows with Visual Studio

While Visual Studio is a Windows application, it can still be used to debug Linux and macOS apps remotely.

- Debugging a .NET Core application on Linux/OSX with Visual Studio

Debugging ASP.NET Core apps require slightly different instructions.

- Debug ASP.NET Core apps in Visual Studio

Visual Studio Code managed debugger

Visual Studio Code is a lightweight cross-platform code editor. It uses the same .NET Core debugger implementation as Visual Studio, but with a simplified user interface.

- Tutorial - Debugging a .NET Core application with Visual Studio Code
- Debugging in Visual Studio Code

.NET Core logging and tracing

8/13/2019 • 3 minutes to read • Edit Online

Logging and tracing are really two names for the same technique. The simple technique has been used since the early days of computers. It simply involves instrumenting an application to write output to be consumed later.

Reasons to use logging and tracing

This simple technique is surprisingly powerful. It can be used in situations where a debugger fails:

- Issues occurring over long periods of time, can be difficult to debug with a traditional debugger. Logs allow for detailed post-mortem review spanning long periods of time. In contrast, debuggers are constrained to real-time analysis.
- Multi-threaded applications and distributed applications are often difficult to debug. Attaching a debugger tends to modify behaviors. Detailed logs can be analyzed as needed to understand complex systems.
- Issues in distributed applications may arise from a complex interaction between many components and it may not be reasonable to connect a debugger to every part of the system.
- Many services shouldn't be stalled. Attaching a debugger often causes timeout failures.
- Issues aren't always foreseen. Logging and tracing are designed for low overhead so that programs can always be recording in case an issue occurs.

.NET Core APIs

Print style APIs

The System.Console, System.Diagnostics.Trace, and System.Diagnostics.Debug classes each provide similar print style APIs convenient for logging.

The choice of which print style API to use is up to you. The key differences are:

- System.Console
  - Always enabled and always writes to the console.
  - Useful for information that your customer may need to see in the release.
  - Because it's the simplest approach, it's often used for ad-hoc temporary debugging. This debug code is often never checked in to source control.
- System.Diagnostics.Trace
  - Only enabled when `TRACE` is defined.
  - Writes to attached Listeners, by default the DefaultTraceListener.
  - Use this API when creating logs that will be enabled in most builds.
- System.Diagnostics.Debug
  - Only enabled when `DEBUG` is defined.
  - Writes to an attached debugger.
  - On `*nix` writes to stderr if `COMPlus_DebugWriteToStdErr` is set.
  - Use this API when creating logs that will be enabled only in debug builds.

Logging events

The following APIs are more event oriented. Rather than logging simple strings they log event objects.

- System.Diagnostics.Tracing.EventSource

    - EventSource is the primary root .NET Core tracing API.
    - Available in all .NET Standard versions.
    - Only allows tracing serializable objects.
    - Writes to the attached event listeners.
    - .NET Core provides listeners for:
        - .NET Core's EventPipe on all platforms
        - Event Tracing for Windows (ETW)
        - LTTng tracing framework for Linux
- System.Diagnostics.DiagnosticSource
    - Included in .NET Core and as a NuGet package for .NET Framework.
    - Allows in-process tracing of non-serializable objects.
    - Includes a bridge to allow selected fields of logged objects to be written to an EventSource.
- System.Diagnostics.Activity
    - Provides a definitive way to identify log messages resulting from a specific activity or transaction. This object can be used to correlate logs across different services.
- System.Diagnostics.EventLog
    - Windows only.
    - Writes messages to the Windows Event Log.
    - System administrators expect fatal application error messages to appear in the Windows Event Log.

ILogger and logging frameworks

The low-level APIs may not be the right choice for your logging needs. You may want to consider a logging framework.

The ILogger interface has been used to create a common logging interface where the loggers can be inserted through dependency injection.

For instance, to allow you to make the best choice for your application `ASP.NET` offers support for a selection of built-in and third-party frameworks:

- ASP.NET built in logging providers
- ASP.NET Third-party logging providers

Logging related references

- How to: Compile Conditionally with Trace and Debug
- How to: Add Trace Statements to Application Code
- ASP.NET Logging provides an overview of the logging techniques it supports.
- C# String Interpolation can simplify writing logging code.
- The Exception.Message property is useful for logging exceptions.
- The System.Diagnostics.StackTrace class can be useful to provide stack info in your logs.

Performance considerations

String formatting can take noticeable CPU processing time.

In performance critical applications, it's recommended that you:

- Avoid lots of logging when no one is listening. Avoid constructing costly logging messages by checking if logging is enabled first.
- Only log what's useful.
- Defer fancy formatting to the analysis stage.

Unit testing in .NET Core and .NET Standard

3/19/2019 • 2 minutes to read • Edit Online

.NET Core makes it easy to create unit tests. This article introduces unit tests and illustrates how they differ from other kinds of tests. The linked resources near the bottom of the page show you how to add a test project to your solution. After you set up your test project, you will be able to run your unit tests using the command line or Visual Studio.

.NET Core 2.0 and later supports .NET Standard 2.0, and we will use its libraries to demonstrate unit tests.

You are able to use built-in .NET Core 2.0 and later unit test project templates for C#, F# and Visual Basic as a starting point for your personal project.

What are unit tests?

Having automated tests is a great way to ensure a software application does what its authors intend it to do. There are multiple types of tests for software applications. These include integration tests, web tests, load tests, and others. **Unit tests** test individual software components and methods. Unit tests should only test code within the developer's control. They should not test infrastructure concerns. Infrastructure concerns include databases, file systems, and network resources.

Also, keep in mind there are best practices for writing tests. For example, Test Driven Development (TDD) is when a unit test is written before the code it is meant to check. TDD is like creating an outline for a book before we write it. It is meant to help developers write simpler, more readable, and efficient code.

> **NOTE**
> The ASP.NET team follows these conventions to help developers come up with good names for test classes and methods.

Try not to introduce dependencies on infrastructure when writing unit tests. These make the tests slow and brittle, and should be reserved for integration tests. You can avoid these dependencies in your application by following the Explicit Dependencies Principle and using Dependency Injection. You can also keep your unit tests in a separate project from your integration tests. This ensures your unit test project doesn't have references to or dependencies on infrastructure packages.

Next steps

More information on unit testing in .NET Core projects:

.NET Core unit test projects are supported for:

- C#
- F#
- Visual Basic

You can also choose between:

- xUnit
- NUnit
- MSTest

You can learn more in the following walkthroughs:

- Create unit tests using *xUnit* and *C#* with the .NET Core CLI.
- Create unit tests using *NUnit* and *C#* with the .NET Core CLI.
- Create unit tests using *MSTest* and *C#* with the .NET Core CLI.
- Create unit tests using *xUnit* and *F#* with the .NET Core CLI.
- Create unit tests using *NUnit* and *F#* with the .NET Core CLI.
- Create unit tests using *MSTest* and *F#* with the .NET Core CLI.
- Create unit tests using *xUnit* and *Visual Basic* with the .NET Core CLI.
- Create unit tests using *NUnit* and *Visual Basic* with the .NET Core CLI.
- Create unit tests using *MSTest* and *Visual Basic* with the .NET Core CLI.

You can learn more in the following articles:

- Visual Studio Enterprise offers great testing tools for .NET Core. Check out Live Unit Testing or code coverage to learn more.
- For more information on how to run selective unit tests, see Running selective unit tests, or including and excluding tests with Visual Studio.
- How to use xUnit with .NET Core and Visual Studio.

Unit testing best practices with .NET Core and .NET Standard

6/27/2019 • 14 minutes to read • Edit Online

There are numerous benefits to writing unit tests; they help with regression, provide documentation, and facilitate good design. However, hard to read and brittle unit tests can wreak havoc on your code base. This article describes some best practices regarding unit test design for your .NET Core and .NET Standard projects.

In this guide, you'll learn some best practices when writing unit tests to keep your tests resilient and easy to understand.

By John Reese with special thanks to Roy Osherove

Why unit test?

Less time performing functional tests

Functional tests are expensive. They typically involve opening up the application and performing a series of steps that you (or someone else), must follow in order to validate the expected behavior. These steps may not always be known to the tester, which means they will have to reach out to someone more knowledgeable in the area in order to carry out the test. Testing itself could take seconds for trivial changes, or minutes for larger changes. Lastly, this process must be repeated for every change that you make in the system.

Unit tests, on the other hand, take milliseconds, can be run at the press of a button and do not necessarily require any knowledge of the system at large. Whether or not the test passes or fails is up to the test runner, not the individual.

Protection against regression

Regression defects are defects that are introduced when a change is made to the application. It is common for testers to not only test their new feature but also features that existed beforehand in order to verify that previously implemented features still function as expected.

With unit testing, it's possible to rerun your entire suite of tests after every build or even after you change a line of code. Giving you confidence that your new code does not break existing functionality.

Executable documentation

It may not always be obvious what a particular method does or how it behaves given a certain input. You may ask yourself: How does this method behave if I pass it a blank string? Null?

When you have a suite of well-named unit tests, each test should be able to clearly explain the expected output for a given input. In addition, it should be able to verify that it actually works.

Less coupled code

When code is tightly coupled, it can be difficult to unit test. Without creating unit tests for the code that you're writing, coupling may be less apparent.

Writing tests for your code will naturally decouple your code, because it would be more difficult to test otherwise.

Characteristics of a good unit test

- **Fast**. It is not uncommon for mature projects to have thousands of unit tests. Unit tests should take very little time to run. Milliseconds.

- **Isolated**. Unit tests are standalone, can be run in isolation, and have no dependencies on any outside factors such as a file system or database.
- **Repeatable**. Running a unit test should be consistent with its results, that is, it always returns the same result if you do not change anything in between runs.
- **Self-Checking**. The test should be able to automatically detect if it passed or failed without any human interaction.
- **Timely**. A unit test should not take a disproportionately long time to write compared to the code being tested. If you find testing the code taking a large amount of time compared to writing the code, consider a design that is more testable.

Let's speak the same language

The term *mock* is unfortunately very misused when talking about testing. The following defines the most common types of *fakes* when writing unit tests:

Fake - A fake is a generic term which can be used to describe either a stub or a mock object. Whether it is a stub or a mock depends on the context in which it's used. So in other words, a fake can be a stub or a mock.

Mock - A mock object is a fake object in the system that decides whether or not a unit test has passed or failed. A mock starts out as a Fake until it is asserted against.

Stub - A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly. By default, a fake starts out as a stub.

Consider the following code snippet:

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

This would be an example of stub being referred to as a mock. In this case, it is a stub. You're just passing in the Order as a means to be able to instantiate `Purchase` (the system under test). The name `MockOrder` is also very misleading because again, the order is not a mock.

A better approach would be

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

By renaming the class to `FakeOrder`, you've made the class a lot more generic, the class can be used as a mock or a stub. Whichever is better for the test case. In the above example, `FakeOrder` is used as a stub. You're not using the `FakeOrder` in any shape or form during the assert. `FakeOrder` was just passed into the `Purchase` class to satisfy the requirements of the constructor.

To use it as a Mock, you could do something like this

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

In this case, you are checking a property on the Fake (asserting against it), so in the above code snippet, the `mockOrder` is a Mock.

> **IMPORTANT**
> It's important to get this terminology correct. If you call your stubs "mocks", other developers are going to make false assumptions about your intent.

The main thing to remember about mocks versus stubs is that mocks are just like stubs, but you assert against the mock object, whereas you do not assert against a stub.

Best practices

Naming your tests

The name of your test should consist of three parts:

- The name of the method being tested.
- The scenario under which it's being tested.
- The expected behavior when the scenario is invoked.

Why?

- Naming standards are important because they explicitly express the intent of the test.

Tests are more than just making sure your code works, they also provide documentation. Just by looking at the suite of unit tests, you should be able to infer the behavior of your code without even looking at the code itself. Additionally, when tests fail, you can see exactly which scenarios do not meet your expectations.

Bad:

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Better:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Arranging your tests

Arrange, Act, Assert is a common pattern when unit testing. As the name implies, it consists of three main actions:

- *Arrange* your objects, creating and setting them up as necessary.
- *Act* on an object.
- *Assert* that something is as expected.

Why?

- Clearly separates what is being tested from the *arrange* and *assert* steps.
- Less chance to intermix assertions with "Act" code.

Readability is one of the most important aspects when writing a test. Separating each of these actions within the test clearly highlight the dependencies required to call your code, how your code is being called, and what you are trying to assert. While it may be possible to combine some steps and reduce the size of your test, the primary goal is to make the test as readable as possible.

Bad:

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

Better:

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

Write minimally passing tests

The input to be used in a unit test should be the simplest possible in order to verify the behavior that you are currently testing.

Why?

- Tests become more resilient to future changes in the codebase.
- Closer to testing behavior over implementation.

Tests that include more information than required to pass the test have a higher chance of introducing errors into the test and can make the intent of the test less clear. When writing tests you want to focus on the behavior. Setting extra properties on models or using non-zero values when not required, only detracts from what you are trying to prove.

Bad:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

Better:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Avoid magic strings

Naming variables in unit tests is as important, if not more important, than naming variables in production code. Unit tests should not contain magic strings.

Why?

- Prevents the need for the reader of the test to inspect the production code in order to figure out what makes the value special.
- Explicitly shows what you're trying to *prove* rather than trying to *accomplish*.

Magic strings can cause confusion to the reader of your tests. If a string looks out of the ordinary, they may wonder why a certain value was chosen for a parameter or return value. This may lead them to take a closer look at the implementation details, rather than focus on the test.

> **TIP**
> When writing tests, you should aim to express as much intent as possible. In the case of magic strings, a good approach is to assign these values to constants.

Bad:

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

Better:

```
        [Fact]
        void Add_MaximumSumResult_ThrowsOverflowException()
        {
            var stringCalculator = new StringCalculator();
            const string MAXIMUM_RESULT = "1001";

            Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

            Assert.Throws<OverflowException>(actual);
        }
```

Avoid logic in tests

When writing your unit tests avoid manual string concatenation and logical conditions such as `if`, `while`, `for`, `switch`, etc.

Why?

- Less chance to introduce a bug inside of your tests.
- Focus on the end result, rather than implementation details.

When you introduce logic into your test suite, the chance of introducing a bug into it increases dramatically. The last place that you want to find a bug is within your test suite. You should have a high level of confidence that your tests work, otherwise, you will not trust them. Tests that you do not trust, do not provide any value. When a test fails, you want to have a sense that something is actually wrong with your code and that it cannot be ignored.

> **TIP**
> If logic in your test seems unavoidable, consider splitting the test up into two or more different tests.

Bad:

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }

}
```

Better:

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

Prefer helper methods to setup and teardown

If you require a similar object or state for your tests, prefer a helper method than leveraging Setup and Teardown attributes if they exist.

Why?

- Less confusion when reading the tests since all of the code is visible from within each test.
- Less chance of setting up too much or too little for the given test.
- Less chance of sharing state between tests which creates unwanted dependencies between them.

In unit testing frameworks, `Setup` is called before each and every unit test within your test suite. While some may see this as a useful tool, it generally ends up leading to bloated and hard to read tests. Each test will generally have different requirements in order to get the test up and running. Unfortunately, `Setup` forces you to use the exact same requirements for each test.

> **NOTE**
> xUnit has removed both SetUp and TearDown as of version 2.x

Bad:

```
private readonly StringCalculator stringCalculator;
public StringCalculatorTests()
{
    stringCalculator = new StringCalculator();
}
```

```
// more tests...
```

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var result = stringCalculator.Add("0,1");

    Assert.Equal(1, result);
}
```

Better:

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var stringCalculator = CreateDefaultStringCalculator();

    var actual = stringCalculator.Add("0,1");

    Assert.Equal(1, actual);
}
```

```
// more tests...
```

```
private StringCalculator CreateDefaultStringCalculator()
{
    return new StringCalculator();
}
```

Avoid multiple asserts

When writing your tests, try to only include one Assert per test. Common approaches to using only one assert include:

- Create a separate test for each assert.
- Use parameterized tests.

Why?

- If one Assert fails, the subsequent Asserts will not be evaluated.
- Ensures you are not asserting multiple cases in your tests.
- Gives you the entire picture as to why your tests are failing.

When introducing multiple asserts into a test case, it is not guaranteed that all of the asserts will be executed. In most unit testing frameworks, once an assertion fails in a unit test, the proceeding tests are automatically considered to be failing. This can be confusing as functionality that is actually working, will be shown as failing.

> **NOTE**
> A common exception to this rule is when asserting against an object. In this case, it is generally acceptable to have multiple asserts against each property to ensure the object is in the state that you expect it to be in.

Bad:

```
[Fact]
public void Add_EdgeCases_ThrowsArgumentExceptions()
{
    Assert.Throws<ArgumentException>(() => stringCalculator.Add(null));
    Assert.Throws<ArgumentException>(() => stringCalculator.Add("a"));
}
```

Better:

```
    [Theory]
    [InlineData(null)]
    [InlineData("a")]
    public void Add_InputNullOrAlphabetic_ThrowsArgumentException(string input)
    {
        var stringCalculator = new StringCalculator();

        Action actual = () => stringCalculator.Add(input);

        Assert.Throws<ArgumentException>(actual);
    }
```

Validate private methods by unit testing public methods

In most cases, there should not be a need to test a private method. Private methods are an implementation detail. You can think of it this way: private methods never exist in isolation. At some point, there is going to be a public facing method that calls the private method as part of its implementation. What you should care about is the end result of the public method that calls into the private one.

Consider the following case

```
public string ParseLogLine(string input)
{
    var sanitizedInput = TrimInput(input);
    return sanitizedInput;
}

private string TrimInput(string input)
{
    return input.Trim();
}
```

Your first reaction may be to start writing a test for `TrimInput` because you want to make sure that the method is working as expected. However, it is entirely possible that `ParseLogLine` manipulates `sanitizedInput` in such a way that you do not expect, rendering a test against `TrimInput` useless.

The real test should be done against the public facing method `ParseLogLine` because that is what you should ultimately care about.

```
public void ParseLogLine_ByDefault_ReturnsTrimmedResult()
{
    var parser = new Parser();

    var result = parser.ParseLogLine(" a ");

    Assert.Equals("a", result);
}
```

With this viewpoint, if you see a private method, find the public method and write your tests against that method. Just because a private method returns the expected result, does not mean the system that eventually calls the private method uses the result correctly.

Stub static references

One of the principles of a unit test is that it must have full control of the system under test. This can be problematic when production code includes calls to static references (e.g. `DateTime.Now`). Consider the following code

```
public int GetDiscountedPrice(int price)
{
    if(DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

How can this code possibly be unit tested? You may try an approach such as

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(2, actual)
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(1, actual);
}
```

Unfortunately, you will quickly realize that there are a couple problems with your tests.

- If the test suite is run on a Tuesday, the second test will pass, but the first test will fail.
- If the test suite is run on any other day, the first test will pass, but the second test will fail.

To solve these problems, you'll need to introduce a *seam* into your production code. One approach is to wrap the code that you need to control in an interface and have the production code depend on that interface.

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if(dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Your test suite now becomes

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Monday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(2, actual);
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Tuesday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(1, actual);
}
```

Now the test suite has full control over `DateTime.Now` and can stub any value when calling into the method.

Unit testing C# in .NET Core using dotnet test and xUnit

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-using-dotnet-test* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. Having a solution makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. The directory and file structure thus far should be as follows:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib` to create the source project. Rename *Class1.cs* to *PrimeService.cs*. You first create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-dotnet-test* directory.

Run the dotnet sln command to add the class library project to the solution:

```
dotnet sln add ./PrimeService/PrimeService.csproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new xunit`. This command creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.csproj* file similar to the following code:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following shows the final solution layout:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

To add the test project to the solution, run the dotnet sln command in the *unit-testing-using-dotnet-test* directory:

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.cs* from the *PrimeService.Tests* directory and create a new C# file named *PrimeService_IsPrimeShould.cs*. Add the following code:

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Fact]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.False(result, "1 should not be prime");
        }
    }
}
```

The `[Fact]` attribute indicates a test method that is run by the test runner. From the *PrimeService.Tests* folder, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works. Replace the existing `IsPrime` method implementation with the following code:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *PrimeService.Tests* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `[Fact]` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests:

- `[Theory]` represents a suite of tests that execute the same code but have different input arguments.
- `[InlineData]` attribute specifies values for those inputs.

Instead of creating new tests, apply these two attributes, `[Theory]` and `[InlineData]`, to create a single theory in the *PrimeService_IsPrimeShould.cs* file. The theory is a method that tests several values less than two, which is the lowest prime number:

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

Run `dotnet test` again, and two of these tests should fail. To make all of the tests pass, change the `if` clause at the beginning of the `IsPrime` method in the *PrimeService.cs* file:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

Additional resources

- xUnit.net official site
- Testing controller logic in ASP.NET Core

Unit testing C# with NUnit and .NET Core

8/2/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-using-nunit* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *PrimeService* directory. The following outline shows the directory and file structure so far:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
```

Make *PrimeService* the current directory and run the following command to create the source project:

```
dotnet new classlib
```

Rename *Class1.cs* to *PrimeService.cs*. You create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-nunit* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add PrimeService/PrimeService.csproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit
```

The dotnet new command creates a test project that uses NUnit as the test library. The generated template configures the test runner in the *PrimeService.Tests.csproj* file:

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.15.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added the Microsoft test SDK, the NUnit test framework, and the NUnit test adapter. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following outline shows the final solution layout:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.csproj
```

Execute the following command in the *unit-testing-using-nunit* directory:

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. In the *PrimeService.Tests* directory, rename the *UnitTest1.cs* file to *PrimeService_IsPrimeShould.cs* and replace its entire contents with the following code:

```
using NUnit.Framework;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestFixture]
    public class PrimeService_IsPrimeShould
    {
        [Test]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            PrimeService primeService = CreatePrimeService();
            var result = primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }

        /*
        More tests
        */

        private PrimeService CreatePrimeService()
        {
             return new PrimeService();
        }
    }
}
```

The `[TestFixture]` attribute denotes a class that contains unit tests. The `[Test]` attribute indicates a method is a test method.

Save this file and execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *unit-testing-using-nunit* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add new tests with the `[Test]` attribute, but that quickly becomes tedious. There are other NUnit attributes that enable you to write a suite of similar tests. A `[TestCase]` attribute is used to create a suite of tests that execute the same code but have different input arguments. You can use the `[TestCase]` attribute to specify values for those inputs.

Instead of creating new tests, apply this attribute to create a single data driven test. The data driven test is a

method that tests several values less than two, which is the lowest prime number:

```
[TestCase(-1)]
[TestCase(0)]
[TestCase(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the `Main` method in the *PrimeService.cs* file:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing C# with MSTest and .NET Core

8/15/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Create the source project

Open a shell window. Create a directory called *unit-testing-using-mstest* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution file for the class library and the test project. Next, create a *PrimeService* directory. The following outline shows the directory and file structure thus far:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib` to create the source project. Rename *Class1.cs* to *PrimeService.cs*. You create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-mstest* directory. Run `dotnet sln add PrimeService/PrimeService.csproj` to add the class library project to the solution.

Create the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new mstest`. The dotnet new command creates a test project that uses MSTest as the test library. The generated template configures the test runner in the *PrimeServiceTests.csproj* file:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added the MSTest SDK, the MSTest test framework, and the MSTest runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following outline shows the final solution layout:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.csproj` in the *unit-testing-using-mstest* directory.

Create the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.cs* from the *PrimeService.Tests* directory and create a new C# file named *PrimeService_IsPrimeShould.cs* with the following content:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestClass]
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [TestMethod]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}
```

The TestClass attribute denotes a class that contains unit tests. The TestMethod attribute indicates a method is a test method.

Save this file and execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *unit-testing-using-mstest* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Add more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add new tests with the TestMethod attribute, but that quickly becomes tedious. There are other MSTest attributes that enable you to write a suite of similar tests. A DataTestMethod attribute represents a suite of tests that execute the same code but have different input arguments. You can use the DataRow attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single data driven test. The data driven test is a method that tests several values less than two, which is the lowest prime number:

```
[DataTestMethod]
[DataRow(-1)]
[DataRow(0)]
[DataRow(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

See also

- Microsoft.VisualStudio.TestTools.UnitTesting
- Use the MSTest framework in unit tests
- MSTest V2 test framework docs

Unit testing F# libraries in .NET Core using dotnet test and xUnit

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *MathService* directory. The directory and file structure thus far is shown below:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run `dotnet new classlib -lang F#` to create the source project. You'll create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run `dotnet sln add .\MathService\MathService.fsproj` to add the class library project to the solution.

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using `dotnet new xunit -lang F#`. This creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

Execute `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` in the *unit-testing-with-fsharp* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *Tests.fs* and add the following code:

```
[<Fact>]
let ``My test`` () =
    Assert.True(true)

[<Fact>]
let ``Fail every time`` () = Assert.True(false)
```

The `[<Fact>]` attribute denotes a test method that is run by the test runner. From the *unit-testing-with-fsharp*, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a sequence of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<Fact>]
let ``Sequence of Evens returns empty collection`` () =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `MathService` class that works:

```
let squaresOfOdds xs =
    Seq.empty<int>
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs this single test. It passes.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<Fact>]
let ``Sequences of Ones and Evens returns Ones`` () =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

Executing `dotnet test` runs your tests and shows you that the new test fails. Now, update the `squaresOfOdds` method to handle this new test. You filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<Fact>]
let ``SquaresOfOdds works`` () =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.Equal(expected, actual)
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing F# libraries in .NET Core using dotnet test and NUnit

7/30/2019 • 5 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *MathService* directory. The following outline shows the directory and file structure so far:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run the following command to create the source project:

```
dotnet new classlib -lang F#
```

You create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add .\MathService\MathService.fsproj
```

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit -lang F#
```

This creates a test project that uses NUnit as the test framework. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="NUnit" Version="3.9.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.9.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added NUnit and the NUnit test adapter. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathService.Tests.fsproj
```

Execute the following command in the *unit-testing-with-fsharp* directory:

```
dotnet sln add .\MathService.Tests\MathService.Tests.fsproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *UnitTest1.fs* and add the following code:

```
namespace MathService.Tests

open System
open NUnit.Framework
open MathService

[<TestFixture>]
type TestClass () =

    [<Test>]
    member this.TestMethodPassing() =
        Assert.True(true)

    [<Test>]
     member this.FailEveryTime() = Assert.True(false)
```

The `[<TestFixture>]` attribute denotes a class that contains tests. The `[<Test>]` attribute denotes a test method that is run by the test runner. From the *unit-testing-with-fsharp* directory, execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a sequence of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<Test>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

Notice that the `expected` sequence has been converted to a list. The NUnit framework relies on many standard .NET types. That dependency means that your public interface and expected results support ICollection rather than IEnumerable.

When you run the test, you see that your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the *Library.fs* class in your MathService project that works:

```
let squaresOfOdds xs =
    Seq.empty<int>
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs your tests. Two tests pass now.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<Test>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

Executing `dotnet test` fails the new test. You must update the `squaresOfOdds` method to handle this new test. You must filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

Notice the call to `Seq.toList`. That creates a list, which implements the ICollection interface.

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<Test>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing F# libraries in .NET Core using dotnet test and MSTest

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *MathService* directory. The directory and file structure thus far is shown below:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run `dotnet new classlib -lang F#` to create the source project. You'll create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run `dotnet sln add .\MathService\MathService.fsproj` to add the class library project to the solution.

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using `dotnet new mstest -lang F#`. This creates a test project that uses MSTest as the test framework. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added MSTest and the MSTest runner. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

Execute `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` in the *unit-testing-with-fsharp* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *Tests.fs* and add the following code:

```
namespace MathService.Tests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open MathService

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing() =
        Assert.IsTrue(true)

    [<TestMethod>]
     member this.FailEveryTime() = Assert.IsTrue(false)
```

The `[<TestClass>]` attribute denotes a class that contains tests. The `[<TestMethod>]` attribute denotes a test method that is run by the test runner. From the *unit-testing-with-fsharp* directory, execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a list of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<TestMethod>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int> |> Seq.toList
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.AreEqual(expected, actual)
```

Notice that the `expected` sequence has been converted to a list. The MSTest library relies on many standard .NET types. That dependency means that your public interface and expected results support ICollection rather than IEnumerable.

When you run the test, you see that your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `Mathservice` class that works:

```
let squaresOfOdds xs =
    Seq.empty<int> |> Seq.toList
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs this single test. It passes.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<TestMethod>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.AreEqual(expected, actual)
```

Executing `dotnet test` fails the new test. You must update the `squaresOfOdds` method to handle this new test. You must filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd |> Seq.toList
```

Notice the call to `Seq.toList`. That creates a list, which implements the ICollection interface.

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<TestMethod>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.AreEqual(expected, actual)
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
    |> Seq.toList
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and xUnit

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-using-dotnet-test* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This practice makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. You have the following directory and file structure thus far:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib -lang VB` to create the source project. Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-dotnet-test* directory. Run `dotnet sln add .\PrimeService\PrimeService.vbproj` to add the class library project to the solution.

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-using-dotnet-test
    unit-testing-vb-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new xunit -lang VB`. This command creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final folder layout:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.vbproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj` in the *unit-testing-vb-using-dotnet-test* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.vb* from the *PrimeService.Tests* directory and create a new Visual Basic file named *PrimeService_IsPrimeShould.VB*. Add the following code:

```
Imports Xunit

Namespace PrimeService.Tests
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Fact>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<Fact>` attribute denotes a test method that is run by the test runner. From the *unit-testing-using-dotnet-test*, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-using-dotnet-test* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<Fact>` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests. A `<Theory>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<InlineData>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single theory. The theory is a method that tests several values less than two, which is the lowest prime number:

```
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Theory>
        <InlineData(-1)>
        <InlineData(0)>
        <InlineData(1)>
        Sub IsPrime_ValueLessThan2_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.False(result, $"{value} should not be prime")
        End Sub

        <Theory>
        <InlineData(2)>
        <InlineData(3)>
        <InlineData(5)>
        <InlineData(7)>
        Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.True(result, $"{value} should be prime")
        End Sub

        <Theory>
        <InlineData(4)>
        <InlineData(6)>
        <InlineData(8)>
        <InlineData(9)>
        Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.False(result, $"{value} should not be prime")
        End Sub
    End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and NUnit

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-nunit* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *PrimeService* directory. The following outline shows the file structure so far:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
```

Make *PrimeService* the current directory and run the following command to create the source project:

```
dotnet new classlib -lang VB
```

Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Imports System

Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first.")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-mstest* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add .\PrimeService\PrimeService.vbproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit -lang VB
```

The dotnet new command creates a test project that uses NUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj* file:

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.15.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added NUnit and the NUnit test adapter. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.vbproj
```

Execute the following command in the *unit-testing-vb-nunit* directory:

```
dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. In the *PrimeService.Tests* directory, rename the *UnitTest1.vb* file to *PrimeService_IsPrimeShould.VB* and replace its entire contents with the following code:

```
Imports NUnit.Framework

Namespace PrimeService.Tests
    <TestFixture>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Test>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<TestFixture>` attribute indicates a class that contains tests. The `<Test>` attribute denotes a method that is run by the test runner. From the *unit-testing-vb-nunit*, execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-nunit* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<Test>` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests. A `<TestCase>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<TestCase>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a series of tests that test several values less than two, which is the lowest prime number:

```
<TestFixture>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <TestCase(-1)>
    <TestCase(0)>
    <TestCase(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <TestCase(2)>
    <TestCase(3)>
    <TestCase(5)>
    <TestCase(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <TestCase(4)>
    <TestCase(6)>
    <TestCase(8)>
    <TestCase(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the `Main` method in the *PrimeServices.cs* file:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and MSTest

7/30/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-mstest* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This practice makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. You have the following directory and file structure thus far:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib -lang VB` to create the source project. Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Imports System

Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-mstest* directory. Run `dotnet sln add .\PrimeService\PrimeService.vbproj` to add the class library project to the solution.

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new mstest -lang VB`. This command creates a test project that uses MSTest as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added MSTest and the MSTest runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.vbproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj` in the *unit-testing-vb-mstest* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.vb* from the *PrimeService.Tests* directory and create a new Visual Basic file named *PrimeService_IsPrimeShould.VB*. Add the following code:

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace PrimeService.Tests
    <TestClass>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <TestMethod>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.IsFalse(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<TestClass>` attribute indicates a class that contains tests. The `<TestMethod>` attribute denotes a method that is run by the test runner. From the *unit-testing-vb-mstest*, execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-mstest* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<TestMethod>` attribute, but that quickly becomes tedious. There are other MSTest attributes that enable you to write a suite of similar tests. A `<DataTestMethod>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<DataRow>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single theory. The theory is a method that tests several values less than two, which is the lowest prime number:

```
<TestClass>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <DataTestMethod>
    <DataRow(-1)>
    <DataRow(0)>
    <DataRow(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <DataTestMethod>
    <DataRow(2)>
    <DataRow(3)>
    <DataRow(5)>
    <DataRow(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <DataTestMethod>
    <DataRow(4)>
    <DataRow(6)>
    <DataRow(8)>
    <DataRow(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Running selective unit tests

3/1/2019 • 2 minutes to read • Edit Online

With the `dotnet test` command in .NET Core, you can use a filter expression to run selective tests. This article demonstrates how to filter which test are run. The following examples use `dotnet test`. If you're using `vstest.console.exe`, replace `--filter` with `--testcasefilter:`.

MSTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MSTestNamespace
{
    [TestClass]
    public class UnitTest1
    {
        [TestCategory("CategoryA")]
        [Priority(1)]
        [TestMethod]
        public void TestMethod1()
        {
        }

        [Priority(2)]
        [TestMethod]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter Method`	Runs tests whose `FullyQualifiedName` contains `Method`. Available in `vstest 15.1+`.
`dotnet test --filter Name~TestMethod1`	Runs tests whose name contains `TestMethod1`.
`dotnet test --filter ClassName=MSTestNamespace.UnitTest1`	Runs tests which are in class `MSTestNamespace.UnitTest1`. **Note:** The `ClassName` value should have a namespace, so `ClassName=UnitTest1` won't work.
`dotnet test --filter FullyQualifiedName!=MSTestNamespace.UnitTest1.TestMethod1`	Runs all tests except `MSTestNamespace.UnitTest1.TestMethod1`.
`dotnet test --filter TestCategory=CategoryA`	Runs tests which are annotated with `[TestCategory("CategoryA")]`.
`dotnet test --filter Priority=2`	Runs tests which are annotated with `[Priority(2)]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **or** `TestCategory` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **and** `TestCategory` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `UnitTest1` **and** `TestCategory` is `CategoryA` **or** `Priority` is 1.

xUnit

```
using Xunit;

namespace XUnitNamespace
{
    public class TestClass1
    {
        [Trait("Category", "CategoryA")]
        [Trait("Priority", "1")]
        [Fact]
        public void Test1()
        {
        }

        [Trait("Priority", "2")]
        [Fact]
        public void Test2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter DisplayName=XUnitNamespace.TestClass1.Test1`	Runs only one test, `XUnitNamespace.TestClass1.Test1`.
`dotnet test --filter FullyQualifiedName!=XUnitNamespace.TestClass1.Test1`	Runs all tests except `XUnitNamespace.TestClass1.Test1`.
`dotnet test --filter DisplayName~TestClass1`	Runs tests whose display name contains `TestClass1`.

In the code example, the defined traits with keys `Category` and `Priority` can be used for filtering.

EXPRESSION	RESULT
`dotnet test --filter XUnit`	Runs tests whose `FullyQualifiedName` contains `XUnit`. Available in `vstest 15.1+`.
`dotnet test --filter Category=CategoryA`	Runs tests which have `[Trait("Category", "CategoryA")]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~TestClass1|Category=CategoryA"`	Runs tests which has `TestClass1` in `FullyQualifiedName` **or** `Category` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~TestClass1&Category=CategoryA"`	Runs tests which has `TestClass1` in `FullyQualifiedName` **and** `Category` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~TestClass1&Category=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `TestClass1` **and** `Category` is `CategoryA` **or** `Priority` is 1.

NUnit

```
using NUnit.Framework;

namespace NUnitNamespace
{
    public class UnitTest1
    {
        [Category("CategoryA")]
        [Property("Priority", 1)]
        [Test]
        public void TestMethod1()
        {
        }

        [Property("Priority", 2)]
        [Test]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter Method`	Runs tests whose `FullyQualifiedName` contains `Method`. Available in `vstest 15.1+`.
`dotnet test --filter Name~TestMethod1`	Runs tests whose name contains `TestMethod1`.
`dotnet test --filter FullyQualifiedName~NUnitNamespace.UnitTest1`	Runs tests which are in class `NUnitNamespace.UnitTest1`.
`dotnet test --filter FullyQualifiedName!=NUnitNamespace.UnitTest1.TestMethod1`	Runs all tests except `NUnitNamespace.UnitTest1.TestMethod1`.
`dotnet test --filter TestCategory=CategoryA`	Runs tests which are annotated with `[Category("CategoryA")]`.
`dotnet test --filter Priority=2`	Runs tests which are annotated with `[Priority(2)]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **or** `TestCategory` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **and** `TestCategory` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `UnitTest1` **and** `TestCategory` is `CategoryA` **or** `Priority` is 1.

Test published output with dotnet vstest

4/8/2019 • 2 minutes to read • Edit Online

You can run tests on already published output by using the `dotnet vstest` command. This will work on xUnit, MSTest, and NUnit tests. Simply locate the DLL file that was part of your published output and run:

```
dotnet vstest <MyPublishedTests>.dll
```

Where `<MyPublishedTests>` is the name of your published test project.

Example

The commands below demonstrate running tests on a published DLL.

```
dotnet new mstest -o MyProject.Tests
cd MyProject.Tests
dotnet publish -o out
dotnet vstest out/MyProject.Tests.dll
```

NOTE

Note: If your app is targeting a framework other than `netcoreapp` you can still run the `dotnet vstest` command by passing in the targeted framework with a framework flag. For example, `dotnet vstest <MyPublishedTests>.dll --Framework:".NETFramework,Version=v4.6"`. In Visual Studio 2017 Update 5 the desired framework is automatically detected.

See also

- Unit Testing with dotnet test and xUnit
- Unit Testing with dotnet test and NUnit
- Unit Testing with dotnet test and MSTest

Using .NET Core SDK and tools in Continuous Integration (CI)

5/29/2019 • 8 minutes to read • Edit Online

This document outlines using the .NET Core SDK and its tools on a build server. The .NET Core toolset works both interactively, where a developer types commands at a command prompt, and automatically, where a Continuous Integration (CI) server runs a build script. The commands, options, inputs, and outputs are the same, and the only things you supply are a way to acquire the tooling and a system to build your app. This document focuses on scenarios of tool acquisition for CI with recommendations on how to design and structure your build scripts.

Installation options for CI build servers

Using the native installers

Native installers are available for macOS, Linux, and Windows. The installers require admin (sudo) access to the build server. The advantage of using a native installer is that it installs all of the native dependencies required for the tooling to run. Native installers also provide a system-wide installation of the SDK.

macOS users should use the PKG installers. On Linux, there's a choice of using a feed-based package manager, such as apt-get for Ubuntu or yum for CentOS, or using the packages themselves, DEB or RPM. On Windows, use the MSI installer.

The latest stable binaries are found at .NET downloads. If you wish to use the latest (and potentially unstable) pre-release tooling, use the links provided at the dotnet/core-sdk GitHub repository. For Linux distributions, `tar.gz` archives (also known as `tarballs`) are available; use the installation scripts within the archives to install .NET Core.

Using the installer script

Using the installer script allows for non-administrative installation on your build server and easy automation for obtaining the tooling. The script takes care of downloading the tooling and extracting it into a default or specified location for use. You can also specify a version of the tooling that you wish to install and whether you want to install the entire SDK or only the shared runtime.

The installer script is automated to run at the start of the build to fetch and install the desired version of the SDK. The *desired version* is whatever version of the SDK your projects require to build. The script allows you to install the SDK in a local directory on the server, run the tools from the installed location, and then clean up (or let the CI service clean up) after the build. This provides encapsulation and isolation to your entire build process. The installation script reference is found in the dotnet-install article.

> **NOTE**
>
> **Azure DevOps Services**
>
> When using the installer script, native dependencies aren't installed automatically. You must install the native dependencies if the operating system doesn't have them. For more information, see Prerequisites for .NET Core on Linux.

CI setup examples

This section describes a manual setup using a PowerShell or bash script, along with a description of several software as a service (SaaS) CI solutions. The SaaS CI solutions covered are Travis CI, AppVeyor, and Azure Pipelines.

Manual setup

Each SaaS service has its own methods for creating and configuring a build process. If you use different SaaS solution than those listed or require customization beyond the pre-packaged support, you must perform at least some manual configuration.

In general, a manual setup requires you to acquire a version of the tools (or the latest nightly builds of the tools) and run your build script. You can use a PowerShell or bash script to orchestrate the .NET Core commands or use a project file that outlines the build process. The orchestration section provides more detail on these options.

After you create a script that performs a manual CI build server setup, use it on your dev machine to build your code locally for testing purposes. Once you confirm that the script is running well locally, deploy it to your CI build server. A relatively simple PowerShell script demonstrates how to obtain the .NET Core SDK and install it on a Windows build server:

```
$ErrorActionPreference="Stop"
$ProgressPreference="SilentlyContinue"

# $LocalDotnet is the path to the locally-installed SDK to ensure the
#   correct version of the tools are executed.
$LocalDotnet=""
# $InstallDir and $CliVersion variables can come from options to the
#   script.
$InstallDir = "./cli-tools"
$CliVersion = "1.0.1"

# Test the path provided by $InstallDir to confirm it exists. If it
#   does, it's removed. This is not strictly required, but it's a
#   good way to reset the environment.
if (Test-Path $InstallDir)
{
    rm -Recurse $InstallDir
}
New-Item -Type "directory" -Path $InstallDir

Write-Host "Downloading the CLI installer..."

# Use the Invoke-WebRequest PowerShell cmdlet to obtain the
#   installation script and save it into the installation directory.
Invoke-WebRequest `
    -Uri "https://dot.net/v1/dotnet-install.ps1" `
    -OutFile "$InstallDir/dotnet-install.ps1"

Write-Host "Installing the CLI requested version ($CliVersion) ..."

# Install the SDK of the version specified in $CliVersion into the
#   specified location ($InstallDir).
& $InstallDir/dotnet-install.ps1 -Version $CliVersion `
    -InstallDir $InstallDir

Write-Host "Downloading and installation of the SDK is complete."

# $LocalDotnet holds the path to dotnet.exe for future use by the
#   script.
$LocalDotnet = "$InstallDir/dotnet"

# Run the build process now. Implement your build script here.
```

You provide the implementation for your build process at the end of the script. The script acquires the tools and then executes your build process. For UNIX machines, the following bash script performs the actions described in the PowerShell script in a similar manner:

```
#!/bin/bash
INSTALLDIR="cli-tools"
CLI_VERSION=1.0.1
DOWNLOADER=$(which curl)
if [ -d "$INSTALLDIR" ]
then
    rm -rf "$INSTALLDIR"
fi
mkdir -p "$INSTALLDIR"
echo Downloading the CLI installer.
$DOWNLOADER https://dot.net/v1/dotnet-install.sh > "$INSTALLDIR/dotnet-install.sh"
chmod +x "$INSTALLDIR/dotnet-install.sh"
echo Installing the CLI requested version $CLI_VERSION. Please wait, installation may take a few minutes.
"$INSTALLDIR/dotnet-install.sh" --install-dir "$INSTALLDIR" --version $CLI_VERSION
if [ $? -ne 0 ]
then
    echo Download of $CLI_VERSION version of the CLI failed. Exiting now.
    exit 0
fi
echo The CLI has been installed.
LOCALDOTNET="$INSTALLDIR/dotnet"
# Run the build process now. Implement your build script here.
```

Travis CI

You can configure Travis CI to install the .NET Core SDK using the `csharp` language and the `dotnet` key. For more information, see the official Travis CI docs on Building a C#, F#, or Visual Basic Project. Note as you access the Travis CI information that the community-maintained `language: csharp` language identifier works for all .NET languages, including F#, and Mono.

Travis CI runs both macOS and Linux jobs in a *build matrix*, where you specify a combination of runtime, environment, and exclusions/inclusions to cover your build combinations for your app. For more information, see the Customizing the Build article in the Travis CI documentation. The MSBuild-based tools include the LTS (1.0.x) and Current (1.1.x) runtimes in the package; so by installing the SDK, you receive everything you need to build.

AppVeyor

AppVeyor installs the .NET Core 1.0.1 SDK with the `Visual Studio 2017` build worker image. Other build images with different versions of the .NET Core SDK are available. For more information, see the appveyor.yml example and the Build worker images article in the AppVeyor docs.

The .NET Core SDK binaries are downloaded and unzipped in a subdirectory using the install script, and then they're added to the `PATH` environment variable. Add a build matrix to run integration tests with multiple versions of the .NET Core SDK:

```
environment:
  matrix:
    - CLI_VERSION: 1.0.1
    - CLI_VERSION: Latest

install:
  # See appveyor.yml example for install script
```

Azure DevOps Services

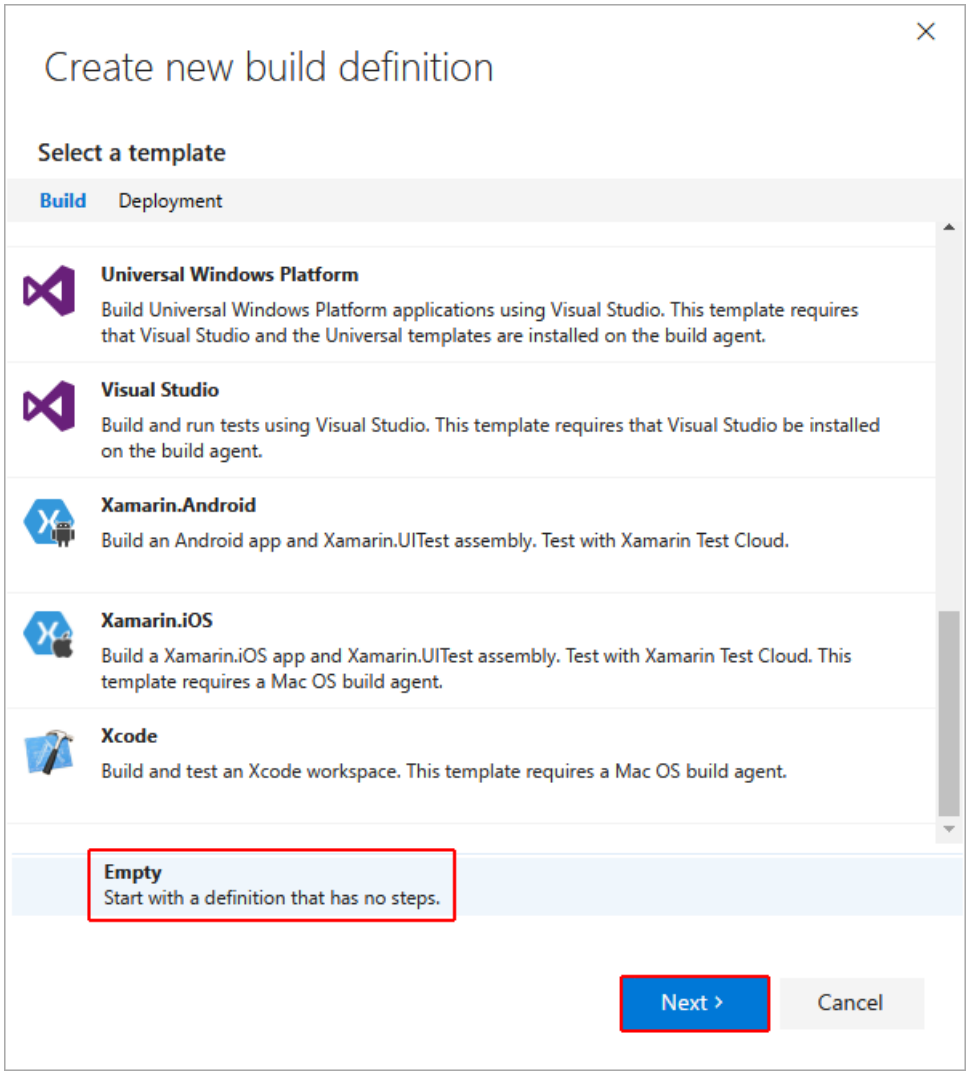
Configure Azure DevOps Services to build .NET Core projects using one of these approaches:

1. Run the script from the manual setup step using your commands.
2. Create a build composed of several Azure DevOps Services built-in build tasks that are configured to use .NET Core tools.

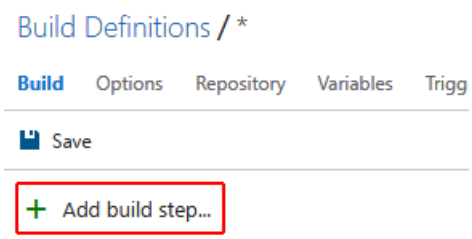
Both solutions are valid. Using a manual setup script, you control the version of the tools that you receive, since you download them as part of the build. The build is run from a script that you must create. This article only covers the manual option. For more information on composing a build with Azure DevOps Services build tasks, see the Azure Pipelines documentation.

To use a manual setup script in Azure DevOps Services, create a new build definition and specify the script to run for the build step. This is accomplished using the Azure DevOps Services user interface:

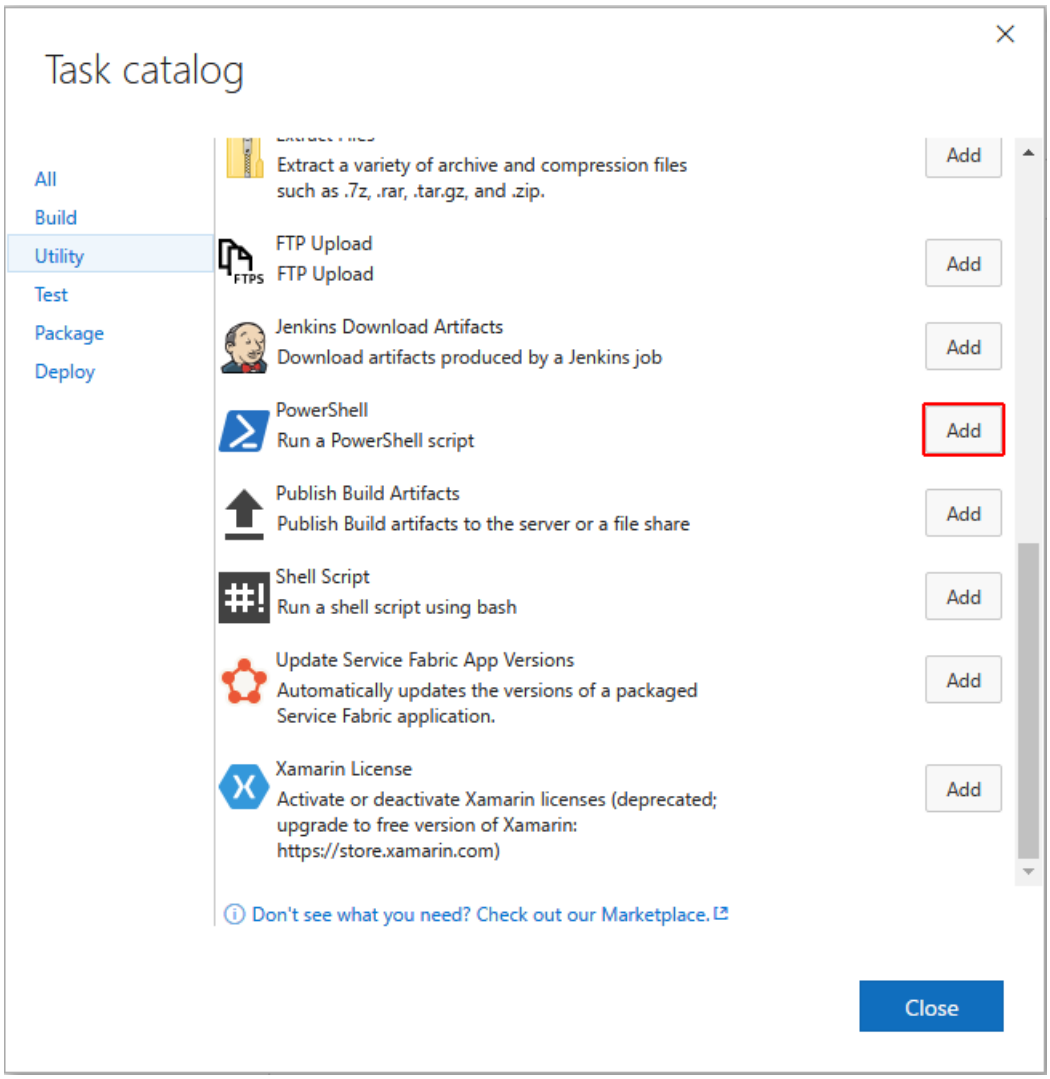
1. Start by creating a new build definition. Once you reach the screen that provides you an option to define what kind of a build you wish to create, select the **Empty** option.



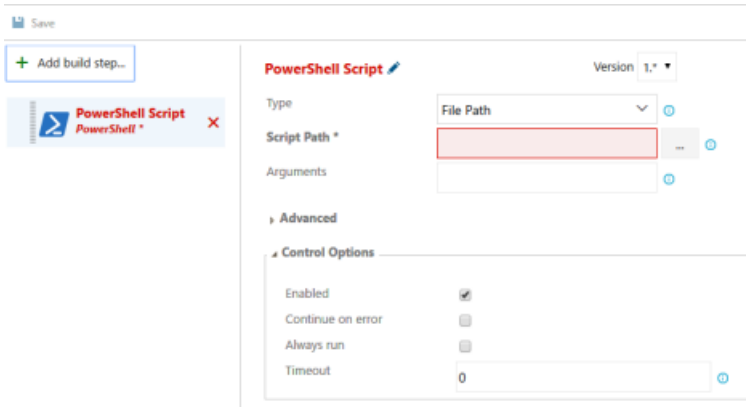
2. After configuring the repository to build, you're directed to the build definitions. Select **Add build step**:



3. You're presented with the **Task catalog**. The catalog contains tasks that you use in the build. Since you have a script, select the **Add** button for **PowerShell: Run a PowerShell script**.



4. Configure the build step. Add the script from the repository that you're building:



Orchestrating the build

Most of this document describes how to acquire the .NET Core tools and configure various CI services without providing information on how to orchestrate, or *actually build*, your code with .NET Core. The choices on how to structure the build process depend on many factors that can't be covered in a general way here. For more information on orchestrating your builds with each technology, explore the resources and samples provided in the documentation sets of Travis CI, AppVeyor, and Azure Pipelines.

Two general approaches that you take in structuring the build process for .NET Core code using the .NET Core tools are using MSBuild directly or using the .NET Core command-line commands. Which approach you should take is determined by your comfort level with the approaches and trade-offs in complexity. MSBuild provides you the ability to express your build process as tasks and targets, but it comes with the added complexity of learning

MSBuild project file syntax. Using the .NET Core command-line tools is perhaps simpler, but it requires you to write orchestration logic in a scripting language like `bash` or PowerShell.

See also

- .NET downloads - Linux

Overview of how .NET Core is versioned

4/4/2019 • 4 minutes to read • Edit Online

.NET Core refers to the .NET Core Runtime and the .NET Core SDK, which contains the tools you need to develop applications. .NET Core SDKs are designed to work with any previous version of the .NET Core Runtime. This article explains the runtime and the SDK version strategy. An explanation of version numbers for .NET Standard can be found in the article introducing .NET Standard.

The .NET Core Runtime and .NET Core SDK add new features at a different rate - in general the .NET Core SDK provides updated tools more quickly than the .NET Core Runtime changes the runtime you use in production.

Versioning details

".NET Core 2.1" refers to the .NET Core Runtime version number. The .NET Core Runtime has a major/minor/patch approach to versioning that follows semantic versioning.

The .NET Core SDK doesn't follow semantic versioning. The .NET Core SDK releases faster and its versions must communicate both the aligned runtime and the SDK's own minor and patch releases. The first two positions of the .NET Core SDK version are locked to the .NET Core Runtime it released with. Each version of the SDK can create applications for this runtime or any lower version.

The third position of the SDK version number communicates both the minor and patch number. The minor version is multiplied by 100. Minor version 1, patch version 2 would be represented as 102. The final two digits represent the patch number. For example, the release of .NET Core 2.2 may create releases like the following table:

CHANGE	.NET CORE RUNTIME	.NET CORE SDK (*)
Initial release	2.2.0	2.2.100
SDK Patch	2.2.0	2.2.101
Runtime and SDK Patch	2.2.1	2.2.102
SDK Feature change	2.2.1	2.2.200

(*) This chart uses a future 2.2 .NET Core Runtime as the example because a historic artifact meant the first SDK for .NET Core 2.1 is 2.1.300. For more information, See the .NET Core version selection.

NOTES:

- If the SDK has 10 feature updates before a runtime feature update, version numbers roll into the 1000 series with numbers like 2.2.1000 as the feature release following 2.2.900. This situation isn't expected to occur.
- 99 patch releases without a feature release won't occur. If a release approaches this number, it forces a feature release.

You can see more details in the initial proposal at the dotnet/designs repository.

Semantic versioning

The .NET Core *Runtime* roughly adheres to Semantic Versioning (SemVer), adopting the use of `MAJOR.MINOR.PATCH` versioning, using the various parts of the version number to describe the degree and type of change.

```
MAJOR.MINOR.PATCH[-PRERELEASE-BUILDNUMBER]
```

The optional `PRERELEASE` and `BUILDNUMBER` parts are never part of supported releases and only exist on nightly builds, local builds from source targets, and unsupported preview releases.

Understand runtime version number changes

`MAJOR` is incremented when:

- Significant changes occur to the product, or a new product direction.
- Breaking changes were taken. There's a high bar to accepting breaking changes.
- An old version is no longer supported.
- A newer `MAJOR` version of an existing dependency is adopted.

`MINOR` is incremented when:

- Public API surface area is added.
- A new behavior is added.
- A newer `MINOR` version of an existing dependency is adopted.
- A new dependency is introduced.

`PATCH` is incremented when:

- Bug fixes are made.
- Support for a newer platform is added.
- A newer `PATCH` version of an existing dependency is adopted.
- Any other change doesn't fit one of the previous cases.

When there are multiple changes, the highest element affected by individual changes is incremented, and the following ones are reset to zero. For example, when `MAJOR` is incremented, `MINOR` and `PATCH` are reset to zero. When `MINOR` is incremented, `PATCH` is reset to zero while `MAJOR` is left untouched.

Version numbers in file names

The files downloaded for .NET Core carry the version, for example, `dotnet-sdk-2.1.300-win10-x64.exe`.

Preview versions

Preview versions have a `-preview[number]-([build]|"final")` appended to the version. For example, `2.0.0-preview1-final`.

Servicing versions

After a release goes out, the release branches generally stop producing daily builds and instead start producing servicing builds. Servicing versions have a `-servicing-[number]` appended to the version. For example, `2.0.1-servicing-006924`.

Relationship to .NET Standard versions

.NET Standard consists of a .NET reference assembly. There are multiple implementations specific to each platform. The reference assembly contains the definition of .NET APIs which are part of a given .NET Standard version. Each implementation fulfills the .NET Standard contract on the specific platform. You can learn more about .NET Standard in the article on .NET Standard in the .NET Guide.

The .NET Standard reference assembly uses a `MAJOR.MINOR` versioning scheme. `PATCH` level isn't useful for .NET Standard because it exposes only an API specification (no implementation) and by definition any change to the

API would represent a change in the feature set, and thus a new `MINOR` version.

The implementations on each platform may be updated, typically as part of the platform release, and thus not evident to the programmers using .NET Standard on that platform.

Each version of .NET Core implements a version of .NET Standard. Implementing a version of .NET Standard implies support for previous versions of .NET Standard. .NET Standard and .NET Core version independently. It's a coincidence that .NET Core 2.0 implements .NET Standard 2.0. .NET Core 2.1 also implements .NET Standard 2.0. .NET Core will support future versions of .NET Standard as they become available.

.NET CORE	.NET STANDARD
1.0	up to 1.6
2.0	up to 2.0
2.1	up to 2.0

See also

- Target frameworks
- .NET Core distribution packaging
- .NET Core Support Lifecycle Fact Sheet
- .NET Core 2+ Version Binding
- Docker images for .NET Core

Select the .NET Core version to use

7/3/2019 • 5 minutes to read • Edit Online

This article explains the policies used by the .NET Core tools, SDK, and runtime for selecting versions. These policies provide a balance between running applications using the specified versions and enabling ease of upgrading both developer and end-user machines. These policies perform the following actions:

- Easy and efficient deployment of .NET Core, including security and reliability updates.
- Use the latest tools and commands independent of target runtime.

Version selection occurs:

- When you run an SDK command, the SDK uses the latest installed version.
- When you build an assembly, target framework monikers define build time APIs.
- When you run a .NET Core application, target framework dependent apps roll forward.
- When you publish a self-contained application, self-contained deployments include the selected runtime.

The rest of this document examines those four scenarios.

The SDK uses the latest installed version

SDK commands include `dotnet new` and `dotnet run`. The .NET Core CLI must choose an SDK version for every `dotnet` command. It uses the latest SDK installed on the machine by default, even if:

- The project targets an earlier version of the .NET Core runtime.
- The latest version of the .NET Core SDK is a preview version.

You can take advantage of the latest SDK features and improvements while targeting earlier .NET Core runtime versions. You can target multiple runtime versions of .NET Core on different projects, using the same SDK tools for all projects.

On rare occasions, you may need to use an earlier version of the SDK. You specify that version in a *global.json* file. The "use latest" policy means you only use *global.json* to specify a .NET Core SDK version earlier than the latest installed version.

global.json can be placed anywhere in the file hierarchy. The CLI searches upward from the project directory for the first *global.json* it finds. You control which projects a given *global.json* applies to by its place in the file system. The .NET CLI searches for a *global.json* file iteratively navigating the path upward from the current working directory. The first *global.json* file found specifies the version used. If that version is installed, that version is used. If the SDK specified in the *global.json* is not found, the .NET CLI rolls forward to the latest SDK installed. Roll-forward is the same as the default behavior, when no *global.json* file is found.

The following example shows the *global.json* syntax:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

The process for selecting an SDK version is:

1. `dotnet` searches for a *global.json* file iteratively reverse-navigating the path upward from the current working

directory.
2. `dotnet` uses the SDK specified in the first *global.json* found.
3. `dotnet` uses the latest installed SDK if no *global.json* is found.

You can learn more about selecting an SDK version in the Matching rules section of the article on *global.json*.

Target Framework Monikers define build time APIs

You build your project against APIs defined in a **Target Framework Moniker** (TFM). You specify the target framework in the project file. Set the `TargetFramework` element in your project file as shown in the following example:

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

You may build your project against multiple TFMs. Setting multiple target frameworks is more common for libraries but can be done with applications as well. You specify a `TargetFrameworks` property (plural of `TargetFramework`). The target frameworks are semicolon-delimited as shown in the following example:

```
<TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
```

A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET Core 2.0 SDK includes the .NET Core 2.0 runtime, which is an implementation of the `netcoreapp2.0` target framework. The .NET Core 2.0 SDK supports `netcoreapp1.0`, `netcoreapp1.1`, and `netcoreapp2.0` but not `netcoreapp2.1` (or higher). You install the .NET Core 2.1 SDK to build for `netcoreapp2.1`.

.NET Standard target frameworks are also capped to the target framework of the runtime the SDK ships with. The .NET Core 2.0 SDK is capped to `netstandard2.0`.

Framework-dependent apps roll forward

When you run an application from source with `dotnet run`, from a **framework-dependent deployment** with `dotnet myapp.dll`, or from a **framework-dependent executable** with `myapp.exe`, the `dotnet` executable is the **host** for the application.

The host chooses the latest patch version installed on the machine. For example, if you specified `netcoreapp2.0` in your project file, and `2.0.4` is the latest .NET runtime installed, the `2.0.4` runtime is used.

If no acceptable `2.0.*` version is found, a new `2.*` version is used. For example, if you specified `netcoreapp2.0` and only `2.1.0` is installed, the application runs using the `2.1.0` runtime. This behavior is referred to as "minor version roll-forward." Lower versions also won't be considered. When no acceptable runtime is installed, the application won't run.

A few usage examples demonstrate the behavior, if you target 2.0:

- 2.0 is specified. 2.0.5 is the highest patch version installed. 2.0.5 is used.
- 2.0 is specified. No 2.0.* versions are installed. 1.1.1 is the highest runtime installed. An error message is displayed.
- 2.0 is specified. No 2.0.* versions are installed. 2.2.2 is the highest 2.x runtime version installed. 2.2.2 is used.
- 2.0 is specified. No 2.x versions are installed. 3.0.0 is installed. An error message is displayed.

Minor version roll-forward has one side-effect that may affect end users. Consider the following scenario:

1. The application specifies that 2.0 is required.
2. When run, version 2.0.* is not installed, however, 2.2.2 is. Version 2.2.2 will be used.

3. Later, the user installs 2.0.5 and runs the application again, 2.0.5 will now be used.

It's possible that 2.0.5 and 2.2.2 behave differently, particularly for scenarios like serializing binary data.

Self-contained deployments include the selected runtime

You can publish an application as a **self-contained distribution**. This approach bundles the .NET Core runtime and libraries with your application. Self-contained deployments don't have a dependency on runtime environments. Runtime version selection occurs at publishing time, not run time.

The publishing process selects the latest patch version of the given runtime family. For example, `dotnet publish` will select .NET Core 2.0.4 if it is the latest patch version in the .NET Core 2.0 runtime family. The target framework (including the latest installed security patches) is packaged with the application.

It's an error if the minimum version specified for an application isn't satisfied. `dotnet publish` binds to the latest runtime patch version (within a given major.minor version family). `dotnet publish` doesn't support the roll-forward semantics of `dotnet run`. For more information about patches and self-contained deployments, see the article on runtime patch selection in deploying .NET Core applications.

Self-contained deployments may require a specific patch version. You can override the minimum runtime patch version (to higher or lower versions) in the project file, as shown in the following example:

```
<RuntimeFrameworkVersion>2.0.4</RuntimeFrameworkVersion>
```

The `RuntimeFrameworkVersion` element overrides the default version policy. For self-contained deployments, the `RuntimeFrameworkVersion` specifies the *exact* runtime framework version. For framework-dependent applications, the `RuntimeFrameworkVersion` specifies the *minimum* required runtime framework version.

How to remove the .NET Core Runtime and SDK

2/28/2019 • 5 minutes to read • Edit Online

Over time, as you install updated versions of the .NET Core runtime and SDK, you may want to remove outdated versions of .NET Core from your machine. Removing older versions of the runtime may change the runtime chosen to run shared framework applications, as detailed in the article on .NET Core version selection.

Should I remove a version?

The .NET Core version selection behaviors and the runtime compatibility of .NET Core across updates enables safe removal of previous versions. .NET Core runtime updates are compatible within a major version 'band' such as 1.x and 2.x. Additionally, newer releases of the .NET Core SDK generally maintain the ability to build applications that target previous versions of the runtime in a compatible manner.

In general, you only need the latest SDK and latest patch version of the runtimes required for your application. Instances where retaining older SDK or Runtime versions include maintaining **project.json**-based applications. Unless your application has specific reasons for earlier SDKs or runtimes, you may safely remove older versions.

Determine what is installed

Starting with .NET Core 2.1, the .NET CLI has options you can use to list the versions of the SDK and runtime that are installed on your machine. Use `dotnet --list-sdks` to see the list of SDKs installed on your machine. Use `dotnet --list-runtimes` to see the list of runtimes installed on your machine. The following text shows typical output for Windows, macOS, or Linux:

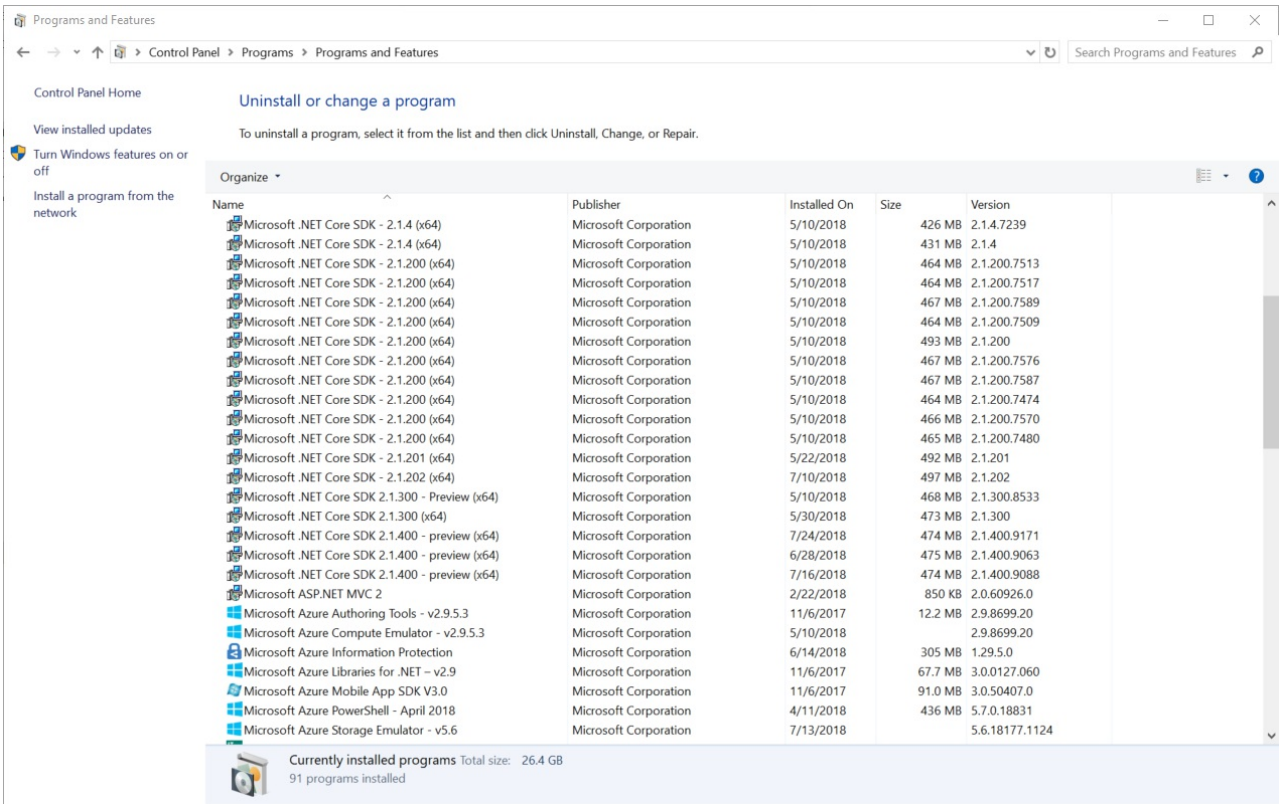
- Windows
- Linux
- macOS

```
C:\> dotnet --list-sdks
2.1.200-preview-007474 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007480 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007509 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007570 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007576 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007587 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007589 [C:\Program Files\dotnet\sdk]
2.1.200 [C:\Program Files\dotnet\sdk]
2.1.201 [C:\Program Files\dotnet\sdk]
2.1.202 [C:\Program Files\dotnet\sdk]
2.1.300-preview2-008533 [C:\Program Files\dotnet\sdk]
2.1.300 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009063 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009088 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009171 [C:\Program Files\dotnet\sdk]

C:\> dotnet --list-runtimes
Microsoft.AspNetCore.All 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.0.6 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.7 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.9 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0-preview2-26406-04 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
```

Uninstalling .NET Core

- Windows
- Linux
- macOS

.NET Core uses the Windows **Add/Remove Programs** dialog to remove versions of the .NET Core runtime and SDK. The following figure shows the **Add/Remove Programs** dialog with several versions of the .NET runtime and SDK installed.



Select any versions you want to remove from your machine and click **Uninstall**.

.NET Core RID Catalog

2/23/2019 • 4 minutes to read • Edit Online

RID is short for *Runtime IDentifier*. RID values are used to identify target platforms where the application runs. They're used by .NET packages to represent platform-specific assets in NuGet packages. The following values are examples of RIDs: `linux-x64`, `ubuntu.14.04-x64`, `win7-x64`, or `osx.10.12-x64`. For the packages with native dependencies, the RID designates on which platforms the package can be restored.

A single RID can be set in the `<RuntimeIdentifier>` element of your project file. Multiple RIDs can be defined as a semicolon-delimited list in the project file's `<RuntimeIdentifiers>` element. They're also used via the `--runtime` option with the following .NET Core CLI commands:

- dotnet build
- dotnet clean
- dotnet pack
- dotnet publish
- dotnet restore
- dotnet run
- dotnet store

RIDs that represent concrete operating systems usually follow this pattern: `[os].[version]-[architecture]-[additional qualifiers]` where:

- `[os]` is the operating/platform system moniker. For example, `ubuntu`.
- `[version]` is the operating system version in the form of a dot-separated (`.`) version number. For example, `15.10`.
  - The version **shouldn't** be marketing versions, as they often represent multiple discrete versions of the operating system with varying platform API surface area.
- `[architecture]` is the processor architecture. For example: `x86`, `x64`, `arm`, or `arm64`.
- `[additional qualifiers]` further differentiate different platforms. For example: `aot`.

RID graph

The RID graph or runtime fallback graph is a list of RIDs that are compatible with each other. The RIDs are defined in the Microsoft.NETCore.Platforms package. You can see the list of supported RIDs and the RID graph in the *runtime.json* file, which is located at the CoreFX repo. In this file, you can see that all RIDs, except for the base one, contain an `"#import"` statement. These statements indicate compatible RIDs.

When NuGet restores packages, it tries to find an exact match for the specified runtime. If an exact match is not found, NuGet walks back the graph until it finds the closest compatible system according to the RID graph.

The following example is the actual entry for the `osx.10.12-x64` RID:

```
"osx.10.12-x64": {
    "#import": [ "osx.10.12", "osx.10.11-x64" ]
}
```

The above RID specifies that `osx.10.12-x64` imports `osx.10.11-x64`. So, when NuGet restores packages, it tries to find an exact match for `osx.10.12-x64` in the package. If NuGet cannot find the specific runtime, it can restore packages that specify `osx.10.11-x64` runtimes, for example.

The following example shows a slightly bigger RID graph also defined in the *runtime.json* file:

```
win7-x64    win7-x86
   |   \   /    |
   |   win7     |
   |     |      |
win-x64  |  win-x86
      \  |  /
        win
         |
        any
```

All RIDs eventually map back to the root `any` RID.

There are some considerations about RIDs that you have to keep in mind when working with them:

- RIDs are **opaque strings** and should be treated as black boxes.
- Don't build RIDs programmatically.
- Use RIDs that are already defined for the platform.
- The RIDs need to be specific, so don't assume anything from the actual RID value.

Using RIDs

To be able to use RIDs, you have to know which RIDs exist. New values are added regularly to the platform. For the latest and complete version, see the runtime.json file on CoreFX repo.

.NET Core 2.0 SDK introduces the concept of portable RIDs. They are new values added to the RID graph that aren't tied to a specific version or OS distribution and are the preferred choice when using .NET Core 2.0 and higher. They're particularly useful when dealing with multiple Linux distros since most distribution RIDs are mapped to the portable RIDs.

The following list shows a small subset of the most common RIDs used for each OS.

Windows RIDs

Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo.

- Portable (.NET Core 2.0 or later versions)
  - `win-x64`
  - `win-x86`
  - `win-arm`
  - `win-arm64`
- Windows 7 / Windows Server 2008 R2
  - `win7-x64`
  - `win7-x86`
- Windows 8.1 / Windows Server 2012 R2
  - `win81-x64`
  - `win81-x86`
  - `win81-arm`
- Windows 10 / Windows Server 2016

- `win10-x64`
- `win10-x86`
- `win10-arm`
- `win10-arm64`

See Prerequisites for .NET Core on Windows for more information.

Linux RIDs

Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo. Devices running a distribution not listed below may work with one of the Portable RIDs. For example, Raspberry Pi devices running a Linux distribution not listed can be targeted with `linux-arm`.

- Portable (.NET Core 2.0 or later versions)
  - `linux-x64` (Most desktop distributions like CentOS, Debian, Fedora, Ubuntu and derivatives)
  - `linux-musl-x64` (Lightweight distributions using musl like Alpine Linux)
  - `linux-arm` (Linux distributions running on ARM like Raspberry Pi)
- Red Hat Enterprise Linux
  - `rhel-x64` (Superseded by `linux-x64` for RHEL above version 6)
  - `rhel.6-x64` (.NET Core 2.0 or later versions)
- Tizen (.NET Core 2.0 or later versions)
  - `tizen`
  - `tizen.4.0.0`
  - `tizen.5.0.0`

See Prerequisites for .NET Core on Linux for more information.

macOS RIDs

macOS RIDs use the older "OSX" branding. Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo.

- Portable (.NET Core 2.0 or later versions)
  - `osx-x64` (Minimum OS version is macOS 10.12 Sierra)
- macOS 10.10 Yosemite
  - `osx.10.10-x64`
- macOS 10.11 El Capitan
  - `osx.10.11-x64`
- macOS 10.12 Sierra (.NET Core 1.1 or later versions)
  - `osx.10.12-x64`
- macOS 10.13 High Sierra (.NET Core 1.1 or later versions)
  - `osx.10.13-x64`
- macOS 10.14 Mojave (.NET Core 1.1 or later versions)
  - `osx.10.14-x64`

See Prerequisites for .NET Core on macOS for more information.

See also

- Runtime IDs

.NET Core SDK overview

8/17/2019 • 2 minutes to read • Edit Online

The .NET Core SDK is a set of libraries and tools that allow developers to create .NET Core applications and libraries. It contains the following components that are used to build and run applications:

- The .NET Core CLI tools.
- .NET Core libraries and runtime.
- The `dotnet` driver.

Acquiring the .NET Core SDK

As with any tooling, the first thing is to get the tools to your machine. Depending on your scenario, you can install the SDK using one of the following methods:

- Use the native installers.
- Use the installation shell script.

The native installers are primarily meant for developer's machines. The SDK is distributed using each supported platform's native install mechanism, such as DEB packages on Ubuntu or MSI bundles on Windows. These installers install and set up the environment as needed for the user to use the SDK immediately after the install. However, they also require administrative privileges on the machine. You can find the SDK to install on the .NET downloads page.

Install scripts, on the other hand, don't require administrative privileges. However, they also don't install any prerequisites on the machine; you need to install all of the prerequisites manually. The scripts are meant mostly for setting up build servers or when you wish to install the tools without admin privileges (do note the prerequisites caveat above). You can find more information in the install script reference article. If you're interested in how to set up the SDK on your CI build server, see the Using .NET Core SDK and tools in Continuous Integration (CI) article.

By default, the SDK installs in a "side-by-side" (SxS) manner, which means multiple versions of the CLI tools can coexist at any given time on a single machine. How the version gets picked when you're running CLI commands is explained in more detail in the Select the .NET Core version to use article.

See also

- .NET Core CLI
- .NET Core versioning overview
- How to remove the .NET Core runtime and SDK
- Select the .NET Core version to use

.NET Core command-line interface (CLI) tools

7/5/2019 • 3 minutes to read • Edit Online

The .NET Core command-line interface (CLI) is a new cross-platform toolchain for developing .NET applications. The CLI is a foundation upon which higher-level tools, such as Integrated Development Environments (IDEs), editors, and build orchestrators, can rest.

Installation

Either use the native installers or use the installation shell scripts:

- The native installers are primarily used on developer's machines and use each supported platform's native install mechanism, for instance, DEB packages on Ubuntu or MSI bundles on Windows. These installers install and configure the environment for immediate use by the developer but require administrative privileges on the machine. You can view the installation instructions in the .NET Core installation guide.
- Shell scripts are primarily used for setting up build servers or when you wish to install the tools without administrative privileges. Install scripts don't install prerequisites on the machine, which must be installed manually. For more information, see the install script reference topic. For information on how to set up CLI on your continuous integration (CI) build server, see Using .NET Core SDK and tools in Continuous Integration (CI).

By default, the CLI installs in a side-by-side (SxS) manner, so multiple versions of the CLI tools can coexist on a single machine. Determining which version is used on a machine where multiple versions are installed is explained in more detail in the Driver section.

CLI commands

The following commands are installed by default:

- .NET Core 2.x
- .NET Core 1.x

Basic commands

- new
- restore
- build
- publish
- run
- test
- vstest
- pack
- migrate
- clean
- sln
- help
- store

Project modification commands

- add package
- add reference
- remove package
- remove reference
- list reference

Advanced commands

- nuget delete
- nuget locals
- nuget push
- msbuild
- dotnet install script

The CLI adopts an extensibility model that allows you to specify additional tools for your projects. For more information, see the .NET Core CLI extensibility model topic.

Command structure

CLI command structure consists of the driver ("dotnet"), the command, and possibly command arguments and options. You see this pattern in most CLI operations, such as creating a new console app and running it from the command line as the following commands show when executed from a directory named *my_app*:

- .NET Core 2.x
- .NET Core 1.x

```
dotnet new console
dotnet build --output /build_output
dotnet /build_output/my_app.dll
```

Driver

The driver is named dotnet and has two responsibilities, either running a framework-dependent app or executing a command.

To run a framework-dependent app, specify the app after the driver, for example, `dotnet /path/to/my_app.dll`. When executing the command from the folder where the app's DLL resides, simply execute `dotnet my_app.dll`. If you want to use a specific version of the .NET Core Runtime, use the `--fx-version <VERSION>` option (see the dotnet command reference).

When you supply a command to the driver, `dotnet.exe` starts the CLI command execution process. For example:

```
> dotnet build
```

First, the driver determines the version of the SDK to use. If there is no 'global.json', the latest version of the SDK available is used. This might be either a preview or stable version, depending on what is latest on the machine. Once the SDK version is determined, it executes the command.

Command

The command performs an action. For example, `dotnet build` builds code. `dotnet publish` publishes code. The commands are implemented as a console application using a `dotnet {command}` convention.

Arguments

The arguments you pass on the command line are the arguments to the command invoked. For example when you execute `dotnet publish my_app.csproj`, the `my_app.csproj` argument indicates the project to publish and is passed to the `publish` command.

Options

The options you pass on the command line are the options to the command invoked. For example when you execute `dotnet publish --output /build_output`, the `--output` option and its value are passed to the `publish` command.

Migration from project.json

If you used Preview 2 tooling to produce *project.json*-based projects, consult the dotnet migrate topic for information on migrating your project to MSBuild/*.csproj* for use with release tooling. For .NET Core projects created prior to the release of Preview 2 tooling, either manually update the project following the guidance in Migrating from DNX to .NET Core CLI (project.json) and then use `dotnet migrate` or directly upgrade your projects.

See also

- dotnet/CLI GitHub Repository
- .NET Core installation guide

.NET Core Global Tools overview

5/15/2019 • 5 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

A .NET Core Global Tool is a special NuGet package that contains a console application. A Global Tool can be installed on your machine on a default location that is included in the PATH environment variable or on a custom location.

If you want to use a .NET Core Global Tool:

- Find information about the tool (usually a website or GitHub page).
- Check the author and statistics in the home for the feed (usually NuGet.org).
- Install the tool.
- Call the tool.
- Update the tool.
- Uninstall the tool.

> **IMPORTANT**
> .NET Core Global Tools appear on your path and run in full trust. Do not install .NET Core Global Tools unless you trust the author.

Find a .NET Core Global Tool

Currently, there isn't a Global Tool search feature in the .NET Core Command-line Interface (CLI).

You can find .NET Core Global Tools on NuGet. However, NuGet doesn't yet allow you to search specifically for .NET Core Global Tools.

You may also find tool recommendations in blog posts or in the natemcmaster/dotnet-tools GitHub repository.

You can also see the source code for the Global Tools created by the ASP.NET team at the aspnet/DotNetTools GitHub repository.

Check the author and statistics

Since .NET Core Global Tools run in full trust and are generally installed on your path, they can be very powerful. Don't download tools from people you don't trust.

If the tool is hosted on NuGet, you can check the author and statistics by searching for the tool.

Install a Global Tool

To install a Global Tool, you use the dotnet tool install .NET Core CLI command. The following example shows how to install a Global Tool in the default location:

```
dotnet tool install -g dotnetsay
```

If the tool can't be installed, error messages are displayed. Check that the feeds you expected are being checked.

If you're trying to install a pre-release version or a specific version of the tool, you can specify the version number using the following format:

```
dotnet tool install -g <package-name> --version <version-number>
```

If installation is successful, a message is displayed showing the command used to call the tool and the version installed, similar to the following example:

```
You can invoke the tool using the following command: dotnetsay
Tool 'dotnetsay' (version '2.0.0') was successfully installed.
```

Global Tools can be installed in the default directory or in a specific location. The default directories are:

OS	PATH
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

These locations are added to the user's path when the SDK is first run, so Global Tools installed there can be called directly.

Note that the Global Tools are user-specific, not machine global. Being user-specific means you cannot install a Global Tool that is available to all users of the machine. The tool is only available for each user profile where the tool was installed.

Global Tools can also be installed in a specific directory. When installed in a specific directory, the user must ensure the command is available, by including that directory in the path, by calling the command with the directory specified, or calling the tool from within the specified directory. In this case, the .NET Core CLI doesn't add this location automatically to the PATH environment variable.

Use the tool

Once the tool is installed, you can call it by using its command. Note that the command may not be the same as the package name.

If the command is `dotnetsay`, you call it with:

```
dotnetsay
```

If the tool author wanted the tool to appear in the context of the `dotnet` prompt, they may have written it in a way that you call it as `dotnet <command>`, such as:

```
dotnet doc
```

You can find which tools are included in an installed Global Tool package by listing the installed packages using the dotnet tool list command.

You can also look for usage instructions at the tool's website or by typing one of the following commands:

```
<command> --help
dotnet <command> --help
```

What could go wrong

Global Tools are framework-dependent applications, which means they rely on a .NET Core runtime installed on your machine. If the expected runtime is not found, they follow normal .NET Core runtime roll-forward rules such as:

- An application rolls forward to the highest patch release of the specified major and minor version.
- If there is no matching runtime with a matching major and minor version number, the next higher minor version is used.
- Roll forward doesn't occur between preview versions of the runtime or between preview versions and release versions. Thus, Global Tools created using preview versions must be rebuilt and republished by the author and reinstalled.
- Additional issues can occur with Global Tools created in .NET Core 2.1 Preview 1. For more information, see .NET Core 2.1 Preview 2 Known Issues.

If an application cannot find an appropriate runtime, it fails to run and reports an error.

Another issue that might happen is that a Global Tool that was created during an earlier preview may not run with your currently installed .NET Core runtimes. You can see which runtimes are installed on your machine using the following command:

```
dotnet --list-runtimes
```

Contact the author of the Global Tool and see if they can recompile and republish their tool package to NuGet with an updated version number. Once they have updated the package on NuGet, you can update your copy.

The .NET Core CLI tries to add the default locations to the PATH environment variable on its first usage. However, there are a couple of scenarios where the location might not be added to PATH automatically, such as:

- If you've set the `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` environment variable.
- On macOS, if you've installed the .NET Core SDK using *.tar.gz* files and not *.pkg*.
- On Linux, you need to edit the shell environment file to configure the PATH.

Other CLI commands

The .NET Core SDK contains other commands that support .NET Core Global Tools. Use any of the `dotnet tool` commands with one of the following options:

- `--global` or `-g` specifies that the command is applicable to user-wide Global Tools.
- `--tool-path` specifies a custom location for Global Tools.

To find out which commands are available for Global Tools:

```
dotnet tool --help
```

Updating a Global Tool involves uninstalling and reinstalling it with the latest stable version. To update a Global Tool, use the dotnet tool update command:

```
dotnet tool update -g <packagename>
```

Remove a Global Tool using the dotnet tool uninstall:

```
dotnet tool uninstall -g <packagename>
```

To display all of the Global Tools currently installed on the machine, along with their version and commands, use the dotnet tool list command:

```
dotnet tool list -g
```

Create a .NET Core Global Tool using the .NET Core CLI

4/16/2019 • 3 minutes to read • Edit Online

This article teaches you how to create and package a .NET Core Global Tool. The .NET Core CLI allows you to create a console application as a Global Tool, which others can easily install and run. .NET Core Global Tools are NuGet packages that are installed from the .NET Core CLI. For more information about Global Tools, see .NET Core Global Tools overview.

This article applies to: ✓ .NET Core 2.1 SDK

Create a project

This article uses the .NET Core CLI to create and manage a project.

Our example tool will be a console application that generates an ASCII bot and prints a message. First, create a new .NET Core Console Application.

```
dotnet new console -o botsay
```

Navigate to the `botsay` directory created by the previous command.

Add the code

Open the `Program.cs` file with your favorite text editor, such as `vim` or Visual Studio Code.

Add the following `using` directive to the top of the file, this helps shorten the code to display the version information of the application.

```
using System.Reflection;
```

Next, move down to the `Main` method. Replace the method with the following code to process the command-line arguments for your application. If no arguments were passed, a short help message is displayed. Otherwise, all of those arguments are transformed into a string and printed with the bot.

```
static void Main(string[] args)
{
    if (args.Length == 0)
    {
        var versionString = Assembly.GetEntryAssembly()
                                .GetCustomAttribute<AssemblyInformationalVersionAttribute>()
                                .InformationalVersion
                                .ToString();

        Console.WriteLine($"botsay v{versionString}");
        Console.WriteLine("-------------");
        Console.WriteLine("\nUsage:");
        Console.WriteLine("  botsay <message>");
        return;
    }

    ShowBot(string.Join(' ', args));
}
```

Create the bot

Next, add a new method named `ShowBot` that takes a string parameter. This method prints out the message and the ASCII bot. The ASCII bot code was taken from the dotnetbot sample.

```
static void ShowBot(string message)
{
    string bot = $"\n        {message}";
    bot += @"
    ___________________
                      \
                       \
                          ....
                          ....'
                           ....
                        ..........
                     ..............'..'..
                   .................'..'.....
                 .......'..........'..'..'....
                ........'..........'..'..'.....
               .'....'..'..........'..'.......'.
               .'...................'...   ......
               .  ......'.........         .....
               .    _            __         ......
              ..    #            ##         ......
             ....       .                  .......
             ......  .......          ............
              ................  .......................
              .........................'................
             ......................'..'......    .......
           ........................'..'.....       .......
     ........    ..'..............'..'....      ..........
   ..'..'...      ................'.......      ..........
  ...'......     ......  ..........  ......         .......
 ...........   .......              ........        ......
.......        '...'.'.              '.'.'.'         ....
.......       .....'..               ..'.....
   ..       ..........               ..'........
          ............               ..............
         ..............              '..............
        ............'..             .'.'............
       ................             .'.'..............
      ..............'..              ..'..'...........
      ................                .'..............
       .........                       ..............
        .....
";
    Console.WriteLine(bot);
}
```

Test the tool

Run the project and see the output. Try these variations of the command-line to see different results:

```
dotnet run
dotnet run -- "Hello from the bot"
dotnet run -- hello from the bot
```

All arguments after the `--` delimiter are passed to your application.

Setup the global tool

Before you can pack and distribute the application as a Global Tool, you need to modify the project file. Open the `botsay.csproj` file and add three new XML nodes to the `<Project><PropertyGroup>` node:

- `<PackAsTool>`
  [REQUIRED] Indicates that the application will be packaged for install as a Global Tool.

- `<ToolCommandName>`
  [OPTIONAL] An alternative name for the tool, otherwise the command name for the tool will be named after the project file. You can have multiple tools in a package, choosing a unique and friendly name helps differentiate from other tools in the same package.

- `<PackageOutputPath>`
  [OPTIONAL] Where the NuGet package will be produced. The NuGet package is what the .NET Core CLI Global Tools uses to install your tool.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>

    <PackAsTool>true</PackAsTool>
    <ToolCommandName>botsay</ToolCommandName>
    <PackageOutputPath>./nupkg</PackageOutputPath>

  </PropertyGroup>

</Project>
```

Even though `<PackageOutputPath>` is optional, use it in this example. Make sure you set it: `<PackageOutputPath>./nupkg</PackageOutputPath>`.

Next, create a NuGet package for your application.

```
dotnet pack
```

The `botsay.1.0.0.nupkg` file is created in the folder identified by the `<PackageOutputPath>` XML value from the `botsay.csproj` file, which in this example is the `./nupkg` folder. This makes it easy to install and test. When you want to release a tool publicly, upload it to https://www.nuget.org. Once the tool is available on NuGet, developers can perform a user-wide installation of the tool using the `--global` option of the dotnet tool install command.

Now that you have a package, install the tool from that package:

```
dotnet tool install --global --add-source ./nupkg botsay
```

The `--add-source` parameter tells the .NET Core CLI to temporarily use the `./nupkg` folder (our `<PackageOutputPath>` folder) as an additional source feed for NuGet packages. For more information about installing Global Tools, see .NET Core Global Tools overview.

If installation is successful, a message is displayed showing the command used to call the tool and the version installed, similar to the following example:

```
You can invoke the tool using the following command: botsay
Tool 'botsay' (version '1.0.0') was successfully installed.
```

You should now be able to type `botsay` and get a response from the tool.

> **NOTE**
>
> If the install was successful, but you cannot use the `botsay` command, you may need to open a new terminal to refresh the PATH.

Remove the tool

Once you're done experimenting with the tool, you can remove it with the following command:

```
dotnet tool uninstall -g botsay
```

Elevated access for dotnet commands

6/27/2019 • 4 minutes to read • Edit Online

Software development best practices guide developers to writing software that requires the least amount of privilege. However, some software, like performance monitoring tools, requires admin permission because of operating system rules. The following guidance describes supported scenarios for writing such software with .NET Core.

The following commands can be run elevated:

- `dotnet tool` commands, such as dotnet tool install.
- `dotnet run --no-build`

We don't recommend running other commands elevated. In particular, we don't recommend elevation with commands that use MSBuild, such as dotnet restore, dotnet build, and dotnet run. The primary issue is permission management problems when a user transitions back and forth between root and a restricted account after issuing dotnet commands. You may find as a restricted user that you don't have access to the file built by a root user. There are ways to resolve this situation, but they're unnecessary to get into in the first place.

You can run commands as root as long as you don't transition back and forth between root and a restricted account. For example, Docker containers run as root by default, so they have this characteristic.

Global tool installation

The following instructions demonstrate the recommended way to install, run, and uninstall .NET Core tools that require elevated permissions to execute.

- Windows
- Linux
- macOS

Install the global tool

If the folder `%ProgramFiles%\dotnet-tools` already exists, do the following to check whether the "Users" group has permission to write or modify that directory:

- Right-click the `%ProgramFiles%\dotnet-tools` folder and select **Properties**. The **Common Properties** dialog box opens.
- Select the **Security** tab. Under **Group or user names**, check whether the "Users" group has permission to write or modify the directory.
- If the "Users" group can write or modify the directory, use a different directory name when installing the tools rather than *dotnet-tools*.

To install tools, run the following command in elevated prompt. It will create the *dotnet-tools* folder during the installation.

```
dotnet tool install PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools".
```

Run the global tool

Option 1 Use the full path with elevated prompt:

```
"%ProgramFiles%\dotnet-tools\TOOLCOMMAND"
```

Option 2 Add the newly created folder to `%Path%`. You only need to do this operation once.

```
setx Path "%Path%;%ProgramFiles%\dotnet-tools\"
```

And run with:

```
TOOLCOMMAND
```

Uninstall the global tool

In an elevated prompt, type the following command:

```
dotnet tool uninstall PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools"
```

Local tools

Local tools are scoped per subdirectory tree, per user. When run elevated, local tools share a restricted user environment to the elevated environment. In Linux and macOS, this results in files being set with root user-only access. If the user switches back to a restricted account, the user can no longer access or write to the files. So installing tools that require elevation as local tools isn't recommended. Instead, use the `--tool-path` option and the previous guidelines for global tools.

Elevation during development

During development, you may need elevated access to test your application. This scenario is common for IoT apps, for example. We recommend that you build the application without elevation and then run it with elevation. There are a few patterns, as follows:

- Using generated executable (it provides the best startup performance):

  ```
  dotnet build
  sudo ./bin/Debug/netcoreapp3.0/APPLICATIONNAME
  ```

- Using the dotnet run command with the `–no-build` flag to avoid generating new binaries:

  ```
  dotnet build
  sudo dotnet run --no-build
  ```

See also

- .NET Core Global Tools overview

.NET Core CLI tools extensibility model

8/14/2019 • 8 minutes to read • Edit Online

This document covers the different ways you can extend the .NET Core Command-line Interface (CLI) tools and explain the scenarios that drive each one of them. You'll see how to consume the tools as well as how to build the different types of tools.

How to extend CLI tools

The CLI tools can be extended in three main ways:

1. Via NuGet packages on a per-project basis

   Per-project tools are contained within the project's context, but they allow easy installation through restoration.

2. Via NuGet packages with custom targets

   Custom targets allow you to easily extend the build process with custom tasks.

3. Via the system's PATH

   PATH-based tools are good for general, cross-project tools that are usable on a single machine.

The three extensibility mechanisms outlined above are not exclusive. You can use one, or all, or a combination of them. Which one to pick depends largely on the goal you are trying to achieve with your extension.

Per-project based extensibility

Per-project tools are framework-dependent deployments that are distributed as NuGet packages. Tools are only available in the context of the project that references them and for which they are restored. Invocation outside of the context of the project (for example, outside of the directory that contains the project) will fail because the command cannot be found.

These tools are perfect for build servers, since nothing outside of the project file is needed. The build process runs restore for the project it builds and tools will be available. Language projects, such as F#, are also in this category since each project can only be written in one specific language.

Finally, this extensibility model provides support for creation of tools that need access to the built output of the project. For instance, various Razor view tools in ASP.NET MVC applications fall into this category.

Consuming per-project tools

Consuming these tools requires you to add a `<DotNetCliToolReference>` element to your project file for each tool you want to use. Inside the `<DotNetCliToolReference>` element, you reference the package in which the tool resides and specify the version you need. After running `dotnet restore`, the tool and its dependencies are restored.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

For tools that need to load the build output of the project for execution, there is usually another dependency which is listed under the regular dependencies in the project file. Since CLI uses MSBuild as its build engine, we recommend that these parts of the tool be written as custom MSBuild targets and tasks, since they can then take part in the overall build process. Also, they can get any and all data easily that is produced via the build, such as the location of the output files, the current configuration being built, and so on. All this information becomes a set of MSBuild properties that can be read from any target. You can see how to add a custom target using NuGet later in this document.

Let's review an example of adding a simple tools-only tool to a simple project. Given an example command called `dotnet-api-search` that allows you to search through the NuGet packages for the specified API, here is a console application's project file that uses that tool:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <!-- The tools reference -->
  <ItemGroup>
    <DotNetCliToolReference Include="dotnet-api-search" Version="1.0.0" />
  </ItemGroup>
</Project>
```

The `<DotNetCliToolReference>` element is structured in a similar way as the `<PackageReference>` element. It needs the package ID of the package containing the tool and its version to be able to restore.

Building tools

As mentioned, tools are just portable console applications. You build tools as you would build any other console application. After you build it, you use the `dotnet pack` command to create a NuGet package (.nupkg file) that contains your code, information about its dependencies, and so on. You can give any name to the package, but the application inside, the actual tool binary, has to conform to the convention of `dotnet-<command>` in order for `dotnet` to be able to invoke it.

> **NOTE**
> In pre-RC3 versions of the .NET Core command-line tools, the `dotnet pack` command had a bug that caused the *.runtimeconfig.json* to not be packed with the tool. Lacking that file results in errors at runtime. If you encounter this behavior, be sure to update to the latest tooling and try the `dotnet pack` again.

Since tools are portable applications, the user consuming the tool must have the version of the .NET Core libraries that the tool was built against in order to run the tool. Any other dependency that the tool uses and that is not contained within the .NET Core libraries is restored and placed in the NuGet cache. The entire tool is, therefore, run using the assemblies from the .NET Core libraries as well as assemblies from the NuGet cache.

These kinds of tools have a dependency graph that is completely separate from the dependency graph of the project that uses them. The restore process first restores the project's dependencies and then restores each of the tools and their dependencies.

You can find richer examples and different combinations of this in the .NET Core CLI repo. You can also see the implementation of tools used in the same repo.

Custom targets

NuGet has the capability to package custom MSBuild targets and props files. With the move of the .NET Core CLI

tools to use MSBuild, the same mechanism of extensibility now applies to .NET Core projects. You would use this type of extensibility when you want to extend the build process, or when you want to access any of the artifacts in the build process, such as generated files, or you want to inspect the configuration under which the build is invoked, and so on.

In the following example, you can see the target's project file using the `csproj` syntax. This instructs the `dotnet pack` command what to package, placing the targets files as well as the assemblies into the *build* folder inside the package. Notice the `<ItemGroup>` element that has the `Label` property set to `dotnet pack instructions`, and the Target defined beneath it.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <Description>Sample Packer</Description>
    <VersionPrefix>0.1.0-preview</VersionPrefix>
    <TargetFramework>netstandard1.3</TargetFramework>
    <DebugType>portable</DebugType>
    <AssemblyName>SampleTargets.PackerTarget</AssemblyName>
  </PropertyGroup>
  <ItemGroup>
    <EmbeddedResource Include="Resources\Pkg\dist-template.xml;compiler\resources\**\*"
Exclude="bin\**;obj\**;**\*.xproj;packages\**" />
    <None Include="build\SampleTargets.PackerTarget.targets" />
  </ItemGroup>
  <ItemGroup Label="dotnet pack instructions">
    <Content Include="build\*.targets">
      <Pack>true</Pack>
      <PackagePath>build\</PackagePath>
    </Content>
  </ItemGroup>
  <Target Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">
    <!-- Collect these items inside a target that runs after build but before packaging. -->
    <ItemGroup>
      <Content Include="$(OutputPath)\*.dll;$(OutputPath)\*.json">
        <Pack>true</Pack>
        <PackagePath>build\</PackagePath>
      </Content>
    </ItemGroup>
  </Target>
  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.DependencyModel" Version="1.0.1-beta-000933"/>
    <PackageReference Include="Microsoft.Build.Framework" Version="0.1.0-preview-00028-160627" />
    <PackageReference Include="Microsoft.Build.Utilities.Core" Version="0.1.0-preview-00028-160627" />
    <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
  </ItemGroup>
  <ItemGroup />
  <PropertyGroup Label="Globals">
    <ProjectGuid>463c66f0-921d-4d34-8bde-7c9d0bffaf7b</ProjectGuid>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(TargetFramework)' == 'netstandard1.3' ">
    <DefineConstants>$(DefineConstants);NETSTANDARD1_3</DefineConstants>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)' == 'Release' ">
    <DefineConstants>$(DefineConstants);RELEASE</DefineConstants>
  </PropertyGroup>
</Project>
```

Consuming custom targets is done by providing a `<PackageReference>` that points to the package and its version inside the project that is being extended. Unlike the tools, the custom targets package does get included into the consuming project's dependency closure.

Using the custom target depends solely on how you configure it. Since it's an MSBuild target, it can depend on a given target, run after another target and can also be manually invoked using the `dotnet msbuild -t:<target-name>` command.

However, if you want to provide a better user experience to your users, you can combine per-project tools and custom targets. In this scenario, the per-project tool would essentially just accept whatever needed parameters and would translate that into the required `dotnet msbuild` invocation that would execute the target. You can see a sample of this kind of synergy on the MVP Summit 2016 Hackathon samples repo in the `dotnet-packer` project.

PATH-based extensibility

PATH-based extensibility is usually used for development machines where you need a tool that conceptually covers more than a single project. The main drawback of this extension mechanism is that it's tied to the machine where the tool exists. If you need it on another machine, you would have to deploy it.

This pattern of CLI toolset extensibility is very simple. As covered in the .NET Core CLI overview, `dotnet` driver can run any command that is named after the `dotnet-<command>` convention. The default resolution logic first probes several locations and finally falls back to the system PATH. If the requested command exists in the system PATH and is a binary that can be invoked, `dotnet` driver will invoke it.

The file must be executable. On Unix systems, this means anything that has the execute bit set via `chmod +x`. On Windows, you can use *cmd* files.

Let's take a look at the very simple implementation of a "Hello World" tool. We will use both `bash` and `cmd` on Windows. The following command will simply echo "Hello World" to the console.

```
#!/bin/bash

echo "Hello World!"
```

```
echo "Hello World"
```

On macOS, we can save this script as `dotnet-hello` and set its executable bit with `chmod +x dotnet-hello`. We can then create a symbolic link to it in `/usr/local/bin` using the command `ln -s <full_path>/dotnet-hello /usr/local/bin/`. This will make it possible to invoke the command using the `dotnet hello` syntax.

On Windows, we can save this script as `dotnet-hello.cmd` and put it in a location that is in a system path (or you can add it to a folder that is already in the path). After this, you can just use `dotnet hello` to run this example.

Custom templates for dotnet new

8/20/2019 • 9 minutes to read • Edit Online

The .NET Core SDK comes with many templates already installed and ready for you to use. The `dotnet new` command isn't only the way to use a template, but also how to install and uninstall templates. Starting with .NET Core 2.0, you can create your own custom templates for any type of project, such as an app, service, tool, or class library. You can even create a template that outputs one or more independent files, such as a configuration file.

You can install custom templates from a NuGet package on any NuGet feed, by referencing a NuGet *.nupkg* file directly, or by specifying a file system directory that contains the template. The template engine offers features that allow you to replace values, include and exclude files, and execute custom processing operations when your template is used.

The template engine is open source, and the online code repository is at dotnet/templating on GitHub. Visit the dotnet/dotnet-template-samples repo for samples of templates. More templates, including templates from third parties, are found at Available templates for dotnet new on GitHub. For more information about creating and using custom templates, see How to create your own templates for dotnet new and the dotnet/templating GitHub repo Wiki.

To follow a walkthrough and create a template, see the Create a custom template for dotnet new tutorial.

.NET default templates

When you install the .NET Core SDK, you receive over a dozen built-in templates for creating projects and files, including console apps, class libraries, unit test projects, ASP.NET Core apps (including Angular and React projects), and configuration files. To list the built-in templates, run the `dotnet new` command with the `-l|--list` option:

```
dotnet new --list
```

Configuration

A template is composed of the following parts:

- Source files and folders.
- A configuration file (*template.json*).

Source files and folders

The source files and folders include whatever files and folders you want the template engine to use when the `dotnet new <TEMPLATE>` command is run. The template engine is designed to use *runnable projects* as source code to produce projects. This has several benefits:

- The template engine doesn't require you to inject special tokens into your project's source code.
- The code files aren't special files or modified in any way to work with the template engine. So, the tools you normally use when working with projects also work with template content.
- You build, run, and debug your template projects just like you do for any of your other projects.
- You can quickly create a template from an existing project just by adding a *./.template.config/template.json* configuration file to the project.

Files and folders stored in the template aren't limited to formal .NET project types. Source files and folders may consist of any content that you wish to create when the template is used, even if the template engine produces just

one file as its output.

Files generated by the template can be modified based on logic and settings you've provided in the *template.json* configuration file. The user can override these settings by passing options to the `dotnet new <TEMPLATE>` command. A common example of custom logic is providing a name for a class or variable in the code file that's deployed by a template.

template.json

The *template.json* file is placed in a *.template.config* folder in the root directory of the template. The file provides configuration information to the template engine. The minimum configuration requires the members shown in the following table, which is sufficient to create a functional template.

MEMBER	TYPE	DESCRIPTION
`$schema`	URI	The JSON schema for the *template.json* file. Editors that support JSON schemas enable JSON-editing features when the schema is specified. For example, Visual Studio Code requires this member to enable IntelliSense. Use a value of `http://json.schemastore.org/template`.
`author`	string	The author of the template.
`classifications`	array(string)	Zero or more characteristics of the template that a user might use to find the template when searching for it. The classifications also appear in the *Tags* column when it appears in a list of templates produced by using the `dotnet new -l|--list` command.
`identity`	string	A unique name for this template.
`name`	string	The name for the template that users should see.
`shortName`	string	A default shorthand name for selecting the template that applies to environments where the template name is specified by the user, not selected via a GUI. For example, the short name is useful when using templates from a command prompt with CLI commands.

The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Example

For example, here is a template folder that contains two content files: *console.cs* and *readme.txt*. Take notice that there is the required folder named *.template.config* that contains the *template.json* file.

```
└──mytemplate
    │   console.cs
    │   readme.txt
    │
    └──.template.config
           template.json
```

The *template.json* file looks like the following:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Travis Chau",
  "classifications": [ "Common", "Console" ],
  "identity": "AdatumCorporation.ConsoleTemplate.CSharp",
  "name": "Adatum Corporation Console Application",
  "shortName": "adatumconsole"
}
```

The *mytemplate* folder is an installable template pack. Once the pack is installed, the `shortName` can be used with the `dotnet new` command. For example, `dotnet new adatumconsole` would output the `console.cs` and `readme.txt` files to the current folder.

Packing a template into a NuGet package (nupkg file)

A custom template is packed with the dotnet pack command and a *.csproj* file. Alternatively, NuGet can be used with the nuget pack command along with a *.nuspec* file. However, NuGet requires the .NET Framework on Windows and Mono on Linux and MacOS.

The *.csproj* file is slightly different from a traditional code-project *.csproj* file. Note the following settings:

1. The `<PackageType>` setting is added and set to `Template`.
2. The `<PackageVersion>` setting is added and set to a valid NuGet version number.
3. The `<PackageId>` setting is added and set to a unique identifier. This identifier is used to uninstall the template pack and is used by NuGet feeds to register your template pack.
4. Generic metadata settings should be set: `<Title>`, `<Authors>`, `<Description>`, and `<PackageTags>`.
5. The `<TargetFramework>` setting must be set, even though the binary produced by the template process isn't used. In the example below it's set to `netstandard2.0`.

A template pack, in the form of a *.nupkg* NuGet package, requires that all templates be stored in the *content* folder within the package. There are a few more settings to add to a *.csproj* file to ensure that the generated *.nupkg* can be installed as a template pack:

1. The `<IncludeContentInPack>` setting is set to `true` to include any file the project sets as **content** in the NuGet package.
2. The `<IncludeBuildOutput>` setting is set to `false` to exclude all binaries generated by the compiler from the NuGet package.
3. The `<ContentTargetFolders>` setting is set to `content`. This makes sure that the files set as **content** are stored in the *content* folder in the NuGet package. This folder in the NuGet package is parsed by the dotnet template system.

An easy way to exclude all code files from being compiled by your template project is by using the `<Compile Remove="**\*" />` item in your project file, inside an `<ItemGroup>` element.

An easy way to structure your template pack is to put all templates in individual folders, and then each template folder inside of a *templates* folder that is located in the same directory as your *.csproj* file. This way, you can use a

single project item to include all files and folders in the *templates* as **content**. Inside of an `<ItemGroup>` element, create a `<Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />` item.

Here is an example *.csproj* file that follows all of the guidelines above. It packs the *templates* child folder to the *content* package folder and excludes any code file from being compiled.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>
    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

The example below demonstrates the file and folder structure of using a *.csproj* to create a template pack. The *MyDotnetTemplates.csproj* file and *templates* folder are both located at the root of a directory named *project_folder*. The *templates* folder contains two templates, *mytemplate1* and *mytemplate2*. Each template has content files and a *.template.config* folder with a *template.json* config file.

```
project_folder
|   MyDotnetTemplates.csproj
|
└──templates
    ├──mytemplate1
    |   |   console.cs
    |   |   readme.txt
    |   |
    |   └──.template.config
    |          template.json
    |
    └──mytemplate2
        |   otherfile.cs
        |
        └──.template.config
               template.json
```

Installing a template

Use the dotnet new -i|--install command to install a package.

To install a template from a NuGet package stored at nuget.org

Use the NuGet package identifier to install a template package.

```
dotnet new -i <NUGET_PACKAGE_ID>
```

To install a template from a local nupkg file

Provide the path to a *.nupkg* NuGet package file.

```
dotnet new -i <PATH_TO_NUPKG_FILE>
```

To install a template from a file system directory

Templates can be installed from a template folder, such as the *mytemplate1* folder from the example above. Specify the folder path of the *.template.config* folder. The path to the template directory does not need to be absolute. However, an absolute path is required to uninstall a template that is installed from a folder.

```
dotnet new -i <FILE_SYSTEM_DIRECTORY>
```

Get a list of installed templates

The uninstall command, without any other parameters, will list all installed templates.

```
dotnet new -u
```

That command returns something similar to the following output:

```
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)
  Microsoft.DotNet.Common.ProjectTemplates.3.0
    Templates:
      Class library (classlib) C#
      Class library (classlib) F#
      Class library (classlib) VB
      Console Application (console) C#
      Console Application (console) F#
      Console Application (console) VB
...
```

The first level of items after `Currently installed items:` are the identifiers used in uninstalling a template. And in the example above, `Microsoft.DotNet.Common.ItemTemplates` and `Microsoft.DotNet.Common.ProjectTemplates.3.0` are listed. If the template was installed by using a file system path, this identifier will the folder path of the *.template.config* folder.

Uninstalling a template

Use the dotnet new -u|--uninstall command to uninstall a package.

If the package was installed by either a NuGet feed or by a *.nupkg* file directly, provide the identifier.

```
dotnet new -u <NUGET_PACKAGE_ID>
```

If the package was installed by specifying a path to the *.template.config* folder, use that **absolute** path to uninstall the package. You can see the absolute path of the template in the output provided by the `dotnet new -u` command. For more information, see the Get a list of installed templates section above.

```
dotnet new -u <ABSOLUTE_FILE_SYSTEM_DIRECTORY>
```

Create a project using a custom template

After a template is installed, use the template by executing the `dotnet new <TEMPLATE>` command as you would with any other pre-installed template. You can also specify options to the `dotnet new` command, including template-specific options you configured in the template settings. Supply the template's short name directly to the command:

```
dotnet new <TEMPLATE>
```

See also

- Create a custom template for dotnet new (tutorial)
- dotnet/templating GitHub repo Wiki
- dotnet/dotnet-template-samples GitHub repo
- How to create your own templates for dotnet new
- *template.json* schema at the JSON Schema Store

How to enable TAB completion for .NET Core CLI

3/1/2019 • 2 minutes to read • Edit Online

Starting with .NET Core 2.0 SDK, the .NET Core CLI supports tab completion. This article describes how to configure tab completion for three shells, PowerShell, Bash, and zsh. Other shells may have support for auto completion. Refer to their documentation on how to configure auto completion, the steps should be similar to the steps described in this article.

This topic applies to: ✓ .NET Core 2.0 SDK and later versions

Once setup, tab completion for the .NET Core CLI is triggered by typing a `dotnet` command in the shell, and then pressing the TAB key. The current command line is sent to the `dotnet complete` command, and the results are processed by your shell. You can test the results without enabling tab completion by sending something directly to the `dotnet complete` command. For example:

```
> dotnet complete "dotnet a"
add
clean
--diagnostics
migrate
pack
```

If that command doesn't work, make sure that .NET Core 2.0 SDK or above is installed. If it's installed, but that command still doesn't work, make sure that the `dotnet` command resolves to a version of .NET Core 2.0 SDK and above. Use the `dotnet --version` command to see what version of `dotnet` your current path is resolving to. For more information, see Select the .NET Core version to use.

Examples

Here are some examples of what tab completion provides:

INPUT	BECOMES	BECAUSE
`dotnet a⇥`	`dotnet add`	`add` is the first subcommand, alphabetically.
`dotnet add p⇥`	`dotnet add --help`	Tab completion matches substrings and `--help` comes first alphabetically.
`dotnet add p⇥⇥`	`dotnet add package`	Pressing tab a second time brings up the next suggestion.
`dotnet add package Microsoft⇥`	`dotnet add package Microsoft.ApplicationInsights.Web`	Results are returned alphabetically.
`dotnet remove reference ⇥`	`dotnet remove reference ..\..\src\OmniSharp.DotNet\OmniSharp.DotNet.csproj`	Tab completion is project file aware.

PowerShell

To add tab completion to **PowerShell** for the .NET Core CLI, create or edit the profile stored in the variable `$PROFILE`. For more information, see How to create your profile and Profiles and execution policy.

Add the following code to your profile:

```
# PowerShell parameter completion shim for the dotnet CLI
Register-ArgumentCompleter -Native -CommandName dotnet -ScriptBlock {
     param($commandName, $wordToComplete, $cursorPosition)
         dotnet complete --position $cursorPosition "$wordToComplete" | ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_, $_, 'ParameterValue', $_)
         }
 }
```

Bash

To add tab completion to your **bash** shell for the .NET Core CLI, add the following code to your `.bashrc` file:

```
# bash parameter completion for the dotnet CLI

_dotnet_bash_complete()
{
  local word=${COMP_WORDS[COMP_CWORD]}

  local completions
  completions="$(dotnet complete --position "${COMP_POINT}" "${COMP_LINE}")"

  COMPREPLY=( $(compgen -W "$completions" -- "$word") )
}

complete -f -F _dotnet_bash_complete dotnet
```

Zsh

To add tab completion to your **zsh** shell for the .NET Core CLI, add the following code to your `.zshrc` file:

```
# zsh parameter completion for the dotnet CLI

_dotnet_zsh_complete()
{
  local completions=("$(dotnet complete "$words")")

  reply=( "${(ps:\n:)completions}" )
}

compctl -K _dotnet_zsh_complete dotnet
```

.NET Core SDK telemetry

6/26/2019 • 3 minutes to read • Edit Online

The .NET Core SDK includes a telemetry feature that collects usage information. It's important that the .NET Team understands how the tools are used so they can be improved. For more information, see What we've learned from .NET Core SDK Telemetry.

The collected data is anonymous and published in an aggregated form for use by both Microsoft and the community under the Creative Commons Attribution License.

Scope

The `dotnet` command is used to launch both apps and the .NET Core CLI. The `dotnet` command itself doesn't collect telemetry. The .NET Core CLI commands run by the `dotnet` command collect the telemetry.

Telemetry *isn't enabled* when using the `dotnet` command itself, with no command attached:

- `dotnet`
- `dotnet [path-to-app]`

Telemetry *is enabled* when using the .NET Core CLI commands, such as:

- `dotnet build`
- `dotnet pack`
- `dotnet restore`
- `dotnet run`

How to opt out

The .NET Core SDK telemetry feature is enabled by default. Opt out of the telemetry feature by setting the `DOTNET_CLI_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Data points

The feature collects the following data:

- Timestamp of invocation†
- Command invoked (for example, "build")†
- Three octet IP address used to determine geographical location†
- `ExitCode` of the command
- Test runner (for test projects)
- Operating system and version†
- Whether runtime IDs are present in the runtimes node
- .NET Core SDK version†

†This metric is published.

Starting with .NET Core 2.0 SDK, new data points are collected:

- `dotnet` command arguments and options: only known arguments and options are collected (not arbitrary strings).

- Whether the SDK is running in a container.
- Target frameworks.
- Hashed MAC address: a cryptographically (SHA256) anonymous and unique ID for a machine. This metric isn't published.
- Hashed current working directory.

The feature doesn't collect personal data, such as usernames or email addresses. It doesn't scan your code and doesn't extract sensitive project-level data, such as name, repo, or author. The data is sent securely to Microsoft servers using Microsoft Azure Application Insights technology, held under restricted access, and published under strict security controls from secure Azure Storage systems.

The .NET team wants to know how the tools are used and if they're working well, not what you're building with the tools. If you suspect that the telemetry is collecting sensitive data or that the data is being insecurely or inappropriately handled, file an issue in the dotnet/cli repository for investigation.

Published data

Published data is available quarterly and are listed at .NET Core SDK Usage Data. The columns of a data file are:

- Timestamp
- Occurrences†
- Command
- Geography‡
- OSFamily
- RuntimeID
- OSVersion
- SDKVersion

†The *Occurrences* column displays the aggregate count of that command's use for that row's metrics that day.

‡Typically, the *Geography* column displays the name of a country/region. In some cases, the continent of Antarctica appears in this column, either due to researchers using .NET Core in Antarctica or incorrect location data.

Example

TIMESTAMP	OCCURRENCES	COMMAND	GEOGRAPHY	OSFAMILY	RUNTIMEID	OSVERSION	SDKVERSION
4/16/2017 0:00	8	run	Uganda	Darwin	osx.10.12-x64	10.12	1.0.1

Datasets

- 2016 - Q3
- 2016 - Q4
- 2017 - Q1
- 2017 - Q2
- 2017 - Q3
- 2017 - Q4

Additional datasets are posted using a standard URL format. Replace `<YEAR>` with the year and replace `<QUARTER>` with the quarter of the year (use `1`, `2`, `3`, or `4`). The files are in tab-separated values (*TSV*) format.

```
https://dotnetcli.blob.core.windows.net/usagedata/dotnet-cli-usage-<YEAR>-q<QUARTER>.tsv
```

License

The Microsoft distribution of .NET Core is licensed with the Microsoft Software License Terms: Microsoft .NET Library. For details on data collection and processing, see the section entitled "Data."

.NET NuGet packages use the same license but don't enable telemetry (see Scope).

Disclosure

The .NET Core SDK displays the following text when you first run one of the .NET Core CLI commands (for example, `dotnet restore`). Text may vary slightly depending on the version of the SDK you're running. This "first run" experience is how Microsoft notifies you about data collection.

```
Welcome to .NET Core!
---------------------
Learn more about .NET Core: https://aka.ms/dotnet-docs
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli-docs

Telemetry
---------
The .NET Core tools collect usage data in order to help us improve your experience.
The data is anonymous and doesn't include command-line arguments.
The data is collected by Microsoft and shared with the community.
You can opt-out of telemetry by setting the DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true'
using your favorite shell.

Read more about .NET Core CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry
```

See also

- What we've learned from .NET Core SDK Telemetry
- Telemetry reference source (dotnet/cli repo)
- .NET Core SDK Usage Data

global.json overview

5/15/2019 • 4 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

The *global.json* file allows you to define which .NET Core SDK version is used when you run .NET Core CLI commands. Selecting the .NET Core SDK is independent from specifying the runtime your project targets. The .NET Core SDK version indicates which versions of the .NET Core CLI tools are used. In general, you want to use the latest version of the tools, so no *global.json* file is needed.

For more information about specifying the runtime instead, see Target frameworks.

.NET Core SDK looks for a *global.json* file in the current working directory (which isn't necessarily the same as the project directory) or one of its parent directories.

global.json schema

sdk

Type: Object

Specifies information about the .NET Core SDK to select.

version

Type: String

The version of the .NET Core SDK to use.

Note that this field:

- Doesn't have globbing support, that is, the full version number has to be specified.
- Doesn't support version ranges.

The following example shows the contents of a *global.json* file:

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

global.json and the .NET Core CLI

It's helpful to know which versions are available in order to set one in the *global.json* file. You can find the full list of supported available SDKs at the .NET Downloads site. Starting with .NET Core 2.1 SDK, you can run the following command to verify which SDK versions are already installed on your machine:

```
dotnet --list-sdks
```

To install additional .NET Core SDK versions on your machine, visit the .NET Downloads site.

You can create a new the *global.json* file in the current directory by executing the dotnet new command, similar to the following example:

```
dotnet new globaljson --sdk-version 2.2.100
```

Matching rules

> **NOTE**
> The matching rules are governed by the apphost, which is part of the .NET Core runtime. The latest version of the host is used when you have multiple runtimes installed side-by-side.

Starting with .NET Core 2.0, the following rules apply when determining which version of the SDK to use:

- If no *global.json* file is found or *global.json* doesn't specify an SDK version, the latest installed SDK version is used. Latest SDK version can be either release or pre-release - the highest version number wins.
- If *global.json* does specify an SDK version:
  - If the specified SDK version is found on the machine, that exact version is used.
  - If the specified SDK version can't be found on the machine, the latest installed SDK **patch version** of that version is used. Latest installed SDK **patch version** can be either release or pre-release - the highest version number wins. In .NET Core 2.1 and higher, the **patch versions** lower than the **patch version** specified are ignored in the SDK selection.
  - If the specified SDK version and an appropriate SDK **patch version** can't be found, an error is thrown.

The SDK version is currently composed of the following parts:

`[.NET Core major version].[.NET Core minor version].[xyz][-optional preview name]`

The **feature release** of the .NET Core SDK is represented by the first digit (`x`) in the last portion of the number (`xyz`) for SDK versions 2.1.100 and higher. In general, the .NET Core SDK has a faster release cycle than .NET Core.

The **patch version** is defined by the last two digits (`yz`) in the last portion of the number (`xyz`) for SDK versions 2.1.100 and higher. For example, if you specify `2.1.300` as the SDK version, SDK selection finds up to `2.1.399` but `2.1.400` isn't considered a patch version for `2.1.300`.

.NET Core SDK versions `2.1.100` through `2.1.201` were released during the transition between version number schemes and don't correctly handle the `xyz` notation. We highly recommend if you specify these versions in the *global.json* file, that you ensure the specified versions are on the target machines.

With .NET Core SDK 1.x, if you specified a version and no exact match was found, the latest installed SDK version was used. Latest SDK version can be either release or pre-release - the highest version number wins.

Troubleshooting build warnings

> **WARNING**
> You are working with a preview version of the .NET Core SDK. You can define the SDK version via a global.json file in the current project. More at https://go.microsoft.com/fwlink/?linkid=869452

This warning indicates that your project is being compiled using a preview version of the .NET Core SDK, as explained in the Matching rules section. .NET Core SDK versions have a history and commitment of being high quality. However, if you don't want to use a preview version, add a *global.json* file to your project hierarchy structure to specify which SDK version to use, and use `dotnet --list-sdks` to confirm that the version is installed on your machine. When a new version is released, to use the new version, either remove the *global.json* file or

update it to use the newer version.

> **WARNING**
> Startup project '{startupProject}' targets framework '.NETCoreApp' version '{targetFrameworkVersion}'. This version of the Entity Framework Core .NET Command-line Tools only supports version 2.0 or higher. For information on using older versions of the tools, see https://go.microsoft.com/fwlink/?linkid=871254

Starting with .NET Core 2.1 SDK (version 2.1.300), the `dotnet ef` command comes included in the SDK. This warning indicates that your project targets EF Core 1.0 or 1.1, which isn't compatible with .NET Core 2.1 SDK and later versions. To compile your project, install .NET Core 2.0 SDK (version 2.1.201) and earlier on your machine and define the desired SDK version using the *global.json* file. For more information about the `dotnet ef` command, see EF Core .NET Command-line Tools.

See also

- How project SDKs are resolved

dotnet command

7/19/2019 • 8 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet` - A tool for managing .NET source code and binaries.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet [command] [arguments] [--additional-deps] [--additionalprobingpath] [-d|--diagnostics] [--fx-version]
    [-h|--help] [--info] [--list-runtimes] [--list-sdks] [--roll-forward-on-no-candidate-fx] [-v|--verbosity]
[--version]
```

Description

`dotnet` is a tool for managing .NET source code and binaries. It exposes commands that perform specific tasks, such as `dotnet build` and `dotnet run`. Each command defines its own arguments. Type `--help` after each command to access brief help documentation.

`dotnet` can be used to run applications, by specifying an application DLL, such as `dotnet myapp.dll`. See .NET Core application deployment for to learn about deployment options.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--additional-deps <PATH>`

Path to additional *.deps.json* file.

`--additionalprobingpath <PATH>`

Path containing probing policy and assemblies to probe.

`-d|--diagnostics`

Enables diagnostic output.

`--fx-version <VERSION>`

Version of the .NET Core runtime to use to run the application.

`-h|--help`

Prints out documentation for a given command, such as `dotnet build --help`. `dotnet --help` prints a list of available commands.

`--info`

Prints out detailed information about a .NET Core installation and the machine environment, such as the current operating system, and commit SHA of the .NET Core version.

`--list-runtimes`

Displays the installed .NET Core runtimes.

`--list-sdks`

Displays the installed .NET Core SDKs.

`--roll-forward-on-no-candidate-fx <N>`

Defines behavior when the required shared framework is not available. `N` can be:

- `0` - Disable even minor version roll forward.
- `1` - Roll forward on minor version, but not on major version. This is the default behavior.
- `2` - Roll forward on minor and major versions.

For more information, see Roll forward.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. Not supported in every command; see specific command page to determine if this option is available.

`--version`

Prints out the version of the .NET Core SDK in use.

dotnet commands

General

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

COMMAND	FUNCTION
dotnet build	Builds a .NET Core application.
dotnet build-server	Interacts with servers started by a build.
dotnet clean	Clean build outputs.
dotnet help	Shows more detailed documentation online for the command.
dotnet migrate	Migrates a valid Preview 2 project to a .NET Core SDK 1.0 project.
dotnet msbuild	Provides access to the MSBuild command line.

COMMAND	FUNCTION
dotnet new	Initializes a C# or F# project for a given template.
dotnet pack	Creates a NuGet package of your code.
dotnet publish	Publishes a .NET framework-dependent or self-contained application.
dotnet restore	Restores the dependencies for a given application.
dotnet run	Runs the application from source.
dotnet sln	Options to add, remove, and list projects in a solution file.
dotnet store	Stores assemblies in the runtime package store.
dotnet test	Runs tests using a test runner.

Project references

COMMAND	FUNCTION
dotnet add reference	Adds a project reference.
dotnet list reference	Lists project references.
dotnet remove reference	Removes a project reference.

NuGet packages

COMMAND	FUNCTION
dotnet add package	Adds a NuGet package.
dotnet remove package	Removes a NuGet package.

NuGet commands

COMMAND	FUNCTION
dotnet nuget delete	Deletes or unlists a package from the server.
dotnet nuget locals	Clears or lists local NuGet resources such as http-request cache, temporary cache, or machine-wide global packages folder.
dotnet nuget push	Pushes a package to the server and publishes it.

Global Tools commands

.NET Core Global Tools are available starting with .NET Core SDK 2.1.300:

COMMAND	FUNCTION
dotnet tool install	Installs a Global Tool on your machine.
dotnet tool list	Lists all Global Tools currently installed in the default directory on your machine or in the specified path.
dotnet tool uninstall	Uninstalls a Global Tool from your machine.
dotnet tool update	Updates a Global Tool on your machine.

Additional tools

Starting with .NET Core SDK 2.1.300, a number of tools that were available only on a per project basis using `DotnetCliToolReference` are now available as part of the .NET Core SDK. These tools are listed in the following table:

TOOL	FUNCTION
dev-certs	Creates and manages development certificates.
ef	Entity Framework Core command-line tools.
sql-cache	SQL Server cache command-line tools.
user-secrets	Manages development user secrets.
watch	Starts a file watcher that runs a command when files change.

For more information about each tool, type `dotnet <tool-name> --help`.

Examples

Creates a new .NET Core console application:

```
dotnet new console
```

Restore dependencies for a given application:

```
dotnet restore
```

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Build a project and its dependencies in a given directory:

```
dotnet build
```

Run an application DLL, such as `myapp.dll`:

```
dotnet myapp.dll
```

Environment variables

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`DOTNET_PACKAGES`

The global packages folder. If not set, it defaults to `~/.nuget/packages` on Unix or `%userprofile%\.nuget\packages` on Windows.

`DOTNET_SERVICING`

Specifies the location of the servicing index to use by the shared host when loading the runtime.

`DOTNET_CLI_TELEMETRY_OPTOUT`

Specifies whether data about the .NET Core tools usage is collected and sent to Microsoft. Set to `true` to opt-out of the telemetry feature (values `true`, `1`, or `yes` accepted). Otherwise, set to `false` to opt into the telemetry features (values `false`, `0`, or `no` accepted). If not set, the default is `false` and the telemetry feature is active.

`DOTNET_MULTILEVEL_LOOKUP`

Specifies whether .NET Core runtime, shared framework, or SDK are resolved from the global location. If not set, it defaults to `true`. Set to `false` to not resolve from the global location and have isolated .NET Core installations (values `0` or `false` are accepted). For more information about multi-level lookup, see Multi-level SharedFX Lookup.

`DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX`

Disables minor version roll forward, if set to `0`. For more information, see Roll forward.

dotnet build

7/19/2019 • 4 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet build` - Builds a project and all of its dependencies.

Synopsis

```
dotnet build [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--force] [--interactive] [--no-dependencies]
    [--no-incremental] [--nologo] [--no-restore] [-o|--output] [-r|--runtime] [-v|--verbosity] [--version-suffix]

dotnet build [-h|--help]
```

Description

The `dotnet build` command builds the project and its dependencies into a set of binaries. The binaries include the project's code in Intermediate Language (IL) files with a *.dll* extension and symbol files used for debugging with a *.pdb* extension. A dependencies JSON file (*.deps.json*) is produced that lists the dependencies of the application. A *.runtimeconfig.json* file is produced, which specifies the shared runtime and its version for the application.

If the project has third-party dependencies, such as libraries from NuGet, they're resolved from the NuGet cache and aren't available with the project's built output. With that in mind, the product of `dotnet build` isn't ready to be transferred to another machine to run. This is in contrast to the behavior of the .NET Framework in which building an executable project (an application) produces output that's runnable on any machine where the .NET Framework is installed. To have a similar experience with .NET Core, you need to use the dotnet publish command. For more information, see .NET Core Application Deployment.

Building requires the *project.assets.json* file, which lists the dependencies of your application. The file is created when `dotnet restore` is executed. Without the assets file in place, the tooling can't resolve reference assemblies, which results in errors. With .NET Core 1.x SDK, you needed to explicitly run the `dotnet restore` before running `dotnet build`. Starting with .NET Core 2.0 SDK, `dotnet restore` runs implicitly when you run `dotnet build`. If you want to disable implicit restore when running the build command, you can pass the `--no-restore` option.

> **NOTE**
>
> Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands, such as `dotnet build` and `dotnet run`, that require a restore to occur. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.
>
> This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Whether the project is executable or not is determined by the `<OutputType>` property in the project file. The following example shows a project that produces executable code:

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

To produce a library, omit the `<OutputType>` property. The main difference in built output is that the IL DLL for a library doesn't contain entry points and can't be executed.

MSBuild

`dotnet build` uses MSBuild to build the project, so it supports both parallel and incremental builds. For more information, see Incremental Builds.

In addition to its options, the `dotnet build` command accepts MSBuild options, such as `-p` for setting properties or `-l` to define a logger. For more information about these options, see the MSBuild Command-Line Reference. Or you can also use the dotnet msbuild command.

Running `dotnet build` is equivalent to `dotnet msbuild -restore -target:Build`.

Arguments

`PROJECT | SOLUTION`

The project or solution file to build. If a project or solution file isn't specified, MSBuild searches the current working directory for a file that has a file extension that ends in either *proj* or *sln* and uses that file.

Options

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default value is `Debug`.

- `-f|--framework <FRAMEWORK>`

  Compiles for a specific framework. The framework must be defined in the project file.

- `--force`

  Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file. Available since .NET Core 2.0 SDK.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--no-dependencies`

  Ignores project-to-project (P2P) references and only builds the specified root project.

- `--no-incremental`

  Marks the build as unsafe for incremental build. This flag turns off incremental compilation and forces a clean rebuild of the project's dependency graph.

- `--no-logo`

  Doesn't display the startup banner or the copyright message. Available since .NET Core 3.0 SDK.

- `--no-restore`

  Doesn't execute an implicit restore during build. Available since .NET Core 2.0 SDK.

- `-o|--output <OUTPUT_DIRECTORY>`

  Directory in which to place the built binaries. You also need to define `--framework` when you specify this option. If not specified, the default path is `./bin/<configuration>/<framework>/`.

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  Specifies the target runtime. For a list of Runtime Identifiers (RIDs), see the RID catalog.

- `-v|--verbosity <LEVEL>`

  Sets the MSBuild verbosity level. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. The default is `minimal`.

- `--version-suffix <VERSION_SUFFIX>`

  Sets the value of the `$(VersionSuffix)` property to use when building the project. This only works if the `$(Version)` property isn't set. Then, `$(Version)` is set to the `$(VersionPrefix)` combined with the `$(VersionSuffix)`, separated by a dash.

Examples

- Build a project and its dependencies:

```
dotnet build
```

- Build a project and its dependencies using Release configuration:

```
dotnet build --configuration Release
```

- Build a project and its dependencies for a specific runtime (in this example, Ubuntu 18.04):

```
dotnet build --runtime ubuntu.18.04-x64
```

- Build the project and use the specified NuGet package source during the restore operation (.NET Core 2.0 SDK and later versions):

```
dotnet build --source c:\packages\mypackages
```

- Build the project and set version 1.2.3.4 as a build parameter using the `-p` MSBuild option:

```
dotnet build -p:Version=1.2.3.4
```

dotnet build-server

5/15/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK and later versions

Name

`dotnet build-server` - Interacts with servers started by a build.

Synopsis

```
dotnet build-server shutdown [--msbuild] [--razor] [--vbcscompiler]
dotnet build-server shutdown [-h|--help]
dotnet build-server [-h|--help]
```

Commands

- `shutdown`

  Shuts down build servers that are started from dotnet. By default, all servers are shut down.

Options

- `-h|--help`

  Prints out a short help for the command.

- `--msbuild`

  Shuts down the MSBuild build server.

- `--razor`

  Shuts down the Razor build server.

- `--vbcscompiler`

  Shuts down the VB/C# compiler build server.

dotnet clean

6/27/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet clean` - Cleans the output of a project.

Synopsis

```
dotnet clean [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--interactive]
    [--nologo] [-o|--output] [-r|--runtime] [-v|--verbosity]
dotnet clean [-h|--help]
```

Description

The `dotnet clean` command cleans the output of the previous build. It's implemented as an MSBuild target, so the project is evaluated when the command is run. Only the outputs created during the build are cleaned. Both intermediate (*obj*) and final output (*bin*) folders are cleaned.

Arguments

`PROJECT | SOLUTION`

The MSBuild project or solution to clean. If a project or solution file is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* or *sln*, and uses that file.

Options

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default value is `Debug`. This option is only required when cleaning if you specified it during build time.

- `-f|--framework <FRAMEWORK>`

  The framework that was specified at build time. The framework must be defined in the project file. If you specified the framework at build time, you must specify the framework when cleaning.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--nologo`

  Doesn't display the startup banner or the copyright message. Available since .NET Core 3.0 SDK.

- `-o|--output <OUTPUT_DIRECTORY>`

  The directory that contains the build artifacts to clean. Specify the `-f|--framework <FRAMEWORK>` switch with the output directory switch if you specified the framework when the project was built.

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  Cleans the output folder of the specified runtime. This is used when a self-contained deployment was created. Option available since .NET Core 2.0 SDK.

- `-v|--verbosity <LEVEL>`

  Sets the MSBuild verbosity level. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. The default is `normal`.

Examples

- Clean a default build of the project:

  ```
  dotnet clean
  ```

- Clean a project built using the Release configuration:

  ```
  dotnet clean --configuration Release
  ```

dotnet help reference

5/15/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 2.0 SDK and later versions

Name

`dotnet help` - Shows more detailed documentation online for the specified command.

Synopsis

```
dotnet help <COMMAND_NAME> [-h|--help]
```

Description

The `dotnet help` command opens up the reference page for more detailed information about the specified command at docs.microsoft.com.

Arguments

- `COMMAND_NAME`

  Name of the .NET Core CLI command. For a list of the valid CLI commands, see CLI commands.

Options

- `-h|--help`

  Prints out a short help for the command.

Examples

- Opens the documentation page for the dotnet new command:

  ```
  dotnet help new
  ```

dotnet migrate

7/3/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet migrate` - Migrates a Preview 2 .NET Core project to a .NET Core SDK-style project.

> **NOTE**
> `dotnet migrate` will be removed from the .NET Core 3.0 SDK in the next preview release.

Synopsis

```
dotnet migrate [<SOLUTION_FILE|PROJECT_DIR>] [--format-report-file-json] [-r|--report-file] [-s|--skip-project-references] [--skip-backup] [-t|--template-file] [-v|--sdk-package-version] [-x|--xproj-file]
dotnet migrate [-h|--help]
```

Description

The `dotnet migrate` command migrates a valid Preview 2 *project.json*-based project to a valid .NET Core SDK-style *csproj* project.

By default, the command migrates the root project and any project references that the root project contains. This behavior is disabled using the `--skip-project-references` option at runtime.

Migration can be performed on the following assets:

- A single project by specifying the *project.json* file to migrate.
- All of the directories specified in the *global.json* file by passing in a path to the *global.json* file.
- A *solution.sln* file, where it migrates the projects referenced in the solution.
- On all subdirectories of the given directory recursively.

The `dotnet migrate` command keeps the migrated *project.json* file inside a `backup` directory, which it creates if the directory doesn't exist. This behavior is overridden using the `--skip-backup` option.

By default, the migration operation outputs the state of the migration process to standard output (STDOUT). If you use the `--report-file <REPORT_FILE>` option, the output is saved to the file specify.

The `dotnet migrate` command only supports valid Preview 2 *project.json*-based projects. This means that you cannot use it to migrate DNX or Preview 1 *project.json*-based projects directly to MSBuild/csproj projects. You first need to manually migrate the project to a Preview 2 *project.json*-based project and then use the `dotnet migrate` command to migrate the project.

Arguments

`PROJECT_JSON/GLOBAL_JSON/SOLUTION_FILE/PROJECT_DIR`

The path to one of the following:

- a *project.json* file to migrate.
- a *global.json* file: the folders specified in *global.json* are migrated.
- a *solution.sln* file: the projects referenced in the solution are migrated.
- a directory to migrate: recursively searches for *project.json* files to migrate inside the specified directory.

Defaults to current directory if nothing is specified.

Options

`--format-report-file-json <REPORT_FILE>`

Output migration report file as JSON rather than user messages.

`-h|--help`

Prints out a short help for the command.

`-r|--report-file <REPORT_FILE>`

Output migration report to a file in addition to the console.

`-s|--skip-project-references [Debug|Release]`

Skip migrating project references. By default, project references are migrated recursively.

`--skip-backup`

Skip moving *project.json*, *global.json*, and **.xproj* to a `backup` directory after successful migration.

`-t|--template-file <TEMPLATE_FILE>`

Template csproj file to use for migration. By default, the same template as the one dropped by `dotnet new console` is used.

`-v|--sdk-package-version <VERSION>`

The version of the sdk package that's referenced in the migrated app. The default is the version of the SDK in `dotnet new`.

`-x|--xproj-file <FILE>`

The path to the xproj file to use. Required when there is more than one xproj in a project directory.

Examples

Migrate a project in the current directory and all of its project-to-project dependencies:

```
dotnet migrate
```

Migrate all projects that *global.json* file includes:

```
dotnet migrate path/to/global.json
```

Migrate only the current project and no project-to-project (P2P) dependencies. Also, use a specific SDK version:

```
dotnet migrate -s -v 1.0.0-preview4
```

dotnet msbuild

5/15/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet msbuild` - Builds a project and all of its dependencies.

Synopsis

```
dotnet msbuild <msbuild_arguments> [-h]
```

Description

The `dotnet msbuild` command allows access to a fully functional MSBuild.

The command has the exact same capabilities as the existing MSBuild command-line client for SDK-style project only. The options are all the same. For more information about the available options, see the MSBuild Command-Line Reference.

The dotnet build command is equivalent to `dotnet msbuild -restore -target:Build`. `dotnet build` is more commonly used for building projects, but `dotnet msbuild` gives you more control. For example, if you have a specific target you want to run (without running the build target), you probably want to use `dotnet msbuild`.

Examples

- Build a project and its dependencies:

  ```
  dotnet msbuild
  ```

- Build a project and its dependencies using Release configuration:

  ```
  dotnet msbuild -p:Configuration=Release
  ```

- Run the publish target and publish for the `osx.10.11-x64` RID:

  ```
  dotnet msbuild -t:Publish -p:RuntimeIdentifiers=osx.10.11-x64
  ```

- See the whole project with all targets included by the SDK:

  ```
  dotnet msbuild -pp
  ```

dotnet new

8/20/2019 • 23 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet new` - Creates a new project, configuration file, or solution based on the specified template.

Synopsis

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet new <TEMPLATE> [--dry-run] [--force] [-i|--install] [-lang|--language] [-n|--name] [--nuget-
source] [-o|--output] [-u|--uninstall] [Template options]
dotnet new <TEMPLATE> [-l|--list] [--type]
dotnet new [-h|--help]
```

Description

The `dotnet new` command provides a convenient way to initialize a valid .NET Core project.

The command calls the template engine to create the artifacts on disk based on the specified template and options.

Arguments

`TEMPLATE`

The template to instantiate when the command is invoked. Each template might have specific options you can pass. For more information, see Template options.

If the `TEMPLATE` value isn't an exact match on text in the **Templates** or **Short Name** column, a substring match is performed on those two columns.

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

The command contains a default list of templates. Use `dotnet new -l` to obtain a list of the available templates. The following table shows the templates that come pre-installed with the .NET Core SDK 2.2.100. The default language for the template is shown inside the brackets.

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
Console Application	`console`	[C#], F#, VB	Common/Console

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
Class library	`classlib`	[C#], F#, VB	Common/Library
Unit Test Project	`mstest`	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	`nunit`	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	`nunit-test`	[C#], F#, VB	Test/NUnit
xUnit Test Project	`xunit`	[C#], F#, VB	Test/xUnit
Razor Page	`page`	[C#]	Web/ASP.NET
MVC ViewImports	`viewimports`	[C#]	Web/ASP.NET
MVC ViewStart	`viewstart`	[C#]	Web/ASP.NET
ASP.NET Core Empty	`web`	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	`mvc`	[C#], F#	Web/MVC
ASP.NET Core Web App	`webapp`, `razor`	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	`angular`	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	`react`	[C#]	Web/MVC/SPA
ASP.NET Core with React.js and Redux	`reactredux`	[C#]	Web/MVC/SPA
Razor Class Library	`razorclasslib`	[C#]	Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	`webapi`	[C#], F#	Web/WebAPI
global.json file	`globaljson`		Config
NuGet Config	`nugetconfig`		Config
Web Config	`webconfig`		Config
Solution File	`sln`		Solution

Options

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

- .NET Core 1.x

`--dry-run`

Displays a summary of what would happen if the given command were run if it would result in a template creation.

`--force`

Forces content to be generated even if it would change existing files. This is required when the output directory already contains a project.

`-h|--help`

Prints out help for the command. It can be invoked for the `dotnet new` command itself or for any template, such as `dotnet new mvc --help`.

`-i|--install <PATH|NUGET_ID>`

Installs a source or template pack from the `PATH` or `NUGET_ID` provided. If you want to install a prerelease version of a template package, you need to specify the version in the format of `<package-name>::<package-version>`. By default, `dotnet new` passes * for the version, which represents the last stable package version. See an example at the Examples section.

For information on creating custom templates, see Custom templates for dotnet new.

`-l|--list`

Lists templates containing the specified name. If invoked for the `dotnet new` command, it lists the possible templates available for the given directory. For example if the directory already contains a project, it doesn't list all project templates.

`-lang|--language {C#|F#|VB}`

The language of the template to create. The language accepted varies by the template (see defaults in the arguments section). Not valid for some templates.

> **NOTE**
> Some shells interpret `#` as a special character. In those cases, you need to enclose the language parameter value, such as `dotnet new console -lang "F#"`.

`-n|--name <OUTPUT_NAME>`

The name for the created output. If no name is specified, the name of the current directory is used.

`--nuget-source`

Specifies a NuGet source to use during install.

`-o|--output <OUTPUT_DIRECTORY>`

Location to place the generated output. The default is the current directory.

`--type`

Filters templates based on available types. Predefined values are "project", "item", or "other".

`-u|--uninstall <PATH|NUGET_ID>`

Uninstalls a source or template pack at the `PATH` or `NUGET_ID` provided. When excluding the

`<PATH|NUGET_ID>` value, all currently installed template packs and their associated templates are displayed.

> **NOTE**
> To uninstall a template using a `PATH`, you need to fully qualify the path. For example, *C:/Users/<USER>/Documents/Templates/GarciaSoftware.ConsoleTemplate.CSharp* will work, but *./GarciaSoftware.ConsoleTemplate.CSharp* from the containing folder will not. Additionally, do not include a final terminating directory slash on your template path.

Template options

Each project template may have additional options available. The core templates have the following additional options:

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

console

`--langVersion <VERSION_NUMBER>` - Sets the `LangVersion` property in the created project file. For example, use `--langVersion 7.3` to use C# 7.3. Not supported for F#.

`--no-restore` - Doesn't execute an implicit restore during project creation.

angular, react, reactredux

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-restore` - Doesn't execute an implicit restore during project creation.

`--no-https` - Project doesn't require HTTPS. This option only applies if `IndividualAuth` or `OrganizationalAuth` are not being used.

razorclasslib

`--no-restore` - Doesn't execute an implicit restore during project creation.

classlib

`-f|--framework <FRAMEWORK>` - Specifies the framework to target. Values: `netcoreapp2.2` to create a .NET Core Class Library or `netstandard2.0` to create a .NET Standard Class Library. The default value is `netstandard2.0`.

`--langVersion <VERSION_NUMBER>` - Sets the `LangVersion` property in the created project file. For example, use `--langVersion 7.3` to use C# 7.3. Not supported for F#.

`--no-restore` - Doesn't execute an implicit restore during project creation.

mstest, xunit

`-p|--enable-pack` - Enables packaging for the project using dotnet pack.

`--no-restore` - Doesn't execute an implicit restore during project creation.

nunit

`-f|--framework <FRAMEWORK>` - Specifies the framework to target. The default value is `netcoreapp2.1`.

`-p|--enable-pack` - Enables packaging for the project using dotnet pack.

`--no-restore` - Doesn't execute an implicit restore during project creation.

page

`-na|--namespace <NAMESPACE_NAME>` - Namespace for the generated code. The default value is `MyApp.Namespace`.

`-np|--no-pagemodel` - Creates the page without a PageModel.

viewimports

`-na|--namespace <NAMESPACE_NAME>` - Namespace for the generated code. The default value is `MyApp.Namespace`.

web

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-restore` - Doesn't execute an implicit restore during project creation.

`--no-https` - Project doesn't require HTTPS. This option only applies if `IndividualAuth` or `OrganizationalAuth` are not being used.

mvc, webapp

`-au|--auth <AUTHENTICATION_TYPE>` - The type of authentication to use. The possible values are:

- `None` - No authentication (Default).
- `Individual` - Individual authentication.
- `IndividualB2C` - Individual authentication with Azure AD B2C.
- `SingleOrg` - Organizational authentication for a single tenant.
- `MultiOrg` - Organizational authentication for multiple tenants.
- `Windows` - Windows authentication.

`--aad-b2c-instance <INSTANCE>` - The Azure Active Directory B2C instance to connect to. Use with `IndividualB2C` authentication. The default value is `https://login.microsoftonline.com/tfp/`.

`-ssp|--susi-policy-id <ID>` - The sign-in and sign-up policy ID for this project. Use with `IndividualB2C` authentication.

`-rp|--reset-password-policy-id <ID>` - The reset password policy ID for this project. Use with `IndividualB2C` authentication.

`-ep|--edit-profile-policy-id <ID>` - The edit profile policy ID for this project. Use with `IndividualB2C` authentication.

`--aad-instance <INSTANCE>` - The Azure Active Directory instance to connect to. Use with `SingleOrg` or `MultiOrg` authentication. The default value is `https://login.microsoftonline.com/`.

`--client-id <ID>` - The Client ID for this project. Use with `IndividualB2C`, `SingleOrg`, or `MultiOrg` authentication. The default value is `11111111-1111-1111-11111111111111111`.

`--domain <DOMAIN>` - The domain for the directory tenant. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `qualified.domain.name`.

`--tenant-id <ID>` - The TenantId ID of the directory to connect to. Use with `SingleOrg` authentication.

The default value is `22222222-2222-2222-2222-222222222222`.

`--callback-path <PATH>` - The request path within the application's base path of the redirect URI. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `/signin-oidc`.

`-r|--org-read-access` - Allows this application read-access to the directory. Only applies to `SingleOrg` or `MultiOrg` authentication.

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-https` - Project doesn't require HTTPS. `app.UseHsts` and `app.UseHttpsRedirection` aren't added to `Startup.Configure`. This option only applies if `Individual`, `IndividualB2C`, `SingleOrg`, or `MultiOrg` aren't being used.

`-uld|--use-local-db` - Specifies LocalDB should be used instead of SQLite. Only applies to `Individual` or `IndividualB2C` authentication.

`--no-restore` - Doesn't execute an implicit restore during project creation.

webapi

`-au|--auth <AUTHENTICATION_TYPE>` - The type of authentication to use. The possible values are:

- `None` - No authentication (Default).
- `IndividualB2C` - Individual authentication with Azure AD B2C.
- `SingleOrg` - Organizational authentication for a single tenant.
- `Windows` - Windows authentication.

`--aad-b2c-instance <INSTANCE>` - The Azure Active Directory B2C instance to connect to. Use with `IndividualB2C` authentication. The default value is `https://login.microsoftonline.com/tfp/`.

`-ssp|--susi-policy-id <ID>` - The sign-in and sign-up policy ID for this project. Use with `IndividualB2C` authentication.

`--aad-instance <INSTANCE>` - The Azure Active Directory instance to connect to. Use with `SingleOrg` authentication. The default value is `https://login.microsoftonline.com/`.

`--client-id <ID>` - The Client ID for this project. Use with `IndividualB2C` or `SingleOrg` authentication. The default value is `11111111-1111-1111-11111111111111111`.

`--domain <DOMAIN>` - The domain for the directory tenant. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `qualified.domain.name`.

`--tenant-id <ID>` - The TenantId ID of the directory to connect to. Use with `SingleOrg` authentication. The default value is `22222222-2222-2222-2222-222222222222`.

`-r|--org-read-access` - Allows this application read-access to the directory. Only applies to `SingleOrg` or `MultiOrg` authentication.

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-https` - Project doesn't require HTTPS. `app.UseHsts` and `app.UseHttpsRedirection` aren't added to `Startup.Configure`. This option only applies if `Individual`, `IndividualB2C`, `SingleOrg`, or `MultiOrg` aren't being used.

`-uld|--use-local-db` - Specifies LocalDB should be used instead of SQLite. Only applies to `Individual` or `IndividualB2C` authentication.

`--no-restore` - Doesn't execute an implicit restore during project creation.

globaljson

`--sdk-version <VERSION_NUMBER>` - Specifies the version of the .NET Core SDK to use in the *global.json* file.

Examples

Create a C# console application project by specifying the template name:

```
dotnet new "Console Application"
```

Create an F# console application project in the current directory:

```
dotnet new console -lang F#
```

Create a .NET Standard class library project in the specified directory (available only with .NET Core SDK 2.0 or later versions):

```
dotnet new classlib -lang VB -o MyLibrary
```

Create a new ASP.NET Core C# MVC project in the current directory with no authentication:

```
dotnet new mvc -au None
```

Create a new xUnit project:

```
dotnet new xunit
```

List all templates available for MVC:

```
dotnet new mvc -l
```

List all templates matching the *we* substring. No exact match is found, so substring matching runs against both the short name and name columns.

```
dotnet new we -l
```

Attempt to invoke the template matching *ng*. If a single match can't be determined, list the templates that are partial matches.

```
dotnet new ng
```

Install version 2.0 of the Single Page Application templates for ASP.NET Core (command option available for .NET Core SDK 1.1 and later versions only):

```
dotnet new -i Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

Create a *global.json* in the current directory setting the SDK version to 2.0.0 (available only with .NET Core SDK 2.0 or later versions):

```
dotnet new globaljson --sdk-version 2.0.0
```

See also

- Custom templates for dotnet new
- Create a custom template for dotnet new
- dotnet/dotnet-template-samples GitHub repo
- Available templates for dotnet new

dotnet nuget delete

7/3/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget delete` - Deletes or unlists a package from the server.

Synopsis

```
dotnet nuget delete [<PACKAGE_NAME> <PACKAGE_VERSION>] [--force-english-output] [--interactive] [-k|--api-key]
[--no-service-endpoint]
    [--non-interactive] [-s|--source]
dotnet nuget delete [-h|--help]
```

Description

The `dotnet nuget delete` command deletes or unlists a package from the server. For nuget.org, the action is to unlist the package.

Arguments

- `PACKAGE_NAME`

  Name/ID of the package to delete.

- `PACKAGE_VERSION`

  Version of the package to delete.

Options

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to block and requires manual action for operations like authentication. Option available since .NET Core 2.2 SDK.

- `-k|--api-key <API_KEY>`

  The API key for the server.

- `--no-service-endpoint`

  Doesn't append "api/v2/package" to the source URL. Option available since .NET Core 2.1 SDK.

- `--non-interactive`

  Doesn't prompt for user input or confirmations.

- `-s|--source <SOURCE>`

  Specifies the server URL. Supported URLs for nuget.org include `https://www.nuget.org`, `https://www.nuget.org/api/v3`, and `https://www.nuget.org/api/v2/package`. For private feeds, replace the host name (for example, `%hostname%/api/v3`).

Examples

- Deletes version 1.0 of package `Microsoft.AspNetCore.Mvc`:

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0
```

- Deletes version 1.0 of package `Microsoft.AspNetCore.Mvc`, not prompting user for credentials or other input:

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0 --non-interactive
```

dotnet nuget locals

7/3/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget locals` - Clears or lists local NuGet resources.

Synopsis

```
dotnet nuget locals <CACHE_LOCATION> [(-c|--clear)|(-l|--list)] [--force-english-output]
dotnet nuget locals [-h|--help]
```

Description

The `dotnet nuget locals` command clears or lists local NuGet resources in the http-request cache, temporary cache, or machine-wide global packages folder.

Arguments

- `CACHE_LOCATION`

  The cache location to list or clear. It accepts one of the following values:

  - `all` - Indicates that the specified operation is applied to all cache types: http-request cache, global packages cache, and the temporary cache.
  - `http-cache` - Indicates that the specified operation is applied only to the http-request cache. The other cache locations aren't affected.
  - `global-packages` - Indicates that the specified operation is applied only to the global packages cache. The other cache locations aren't affected.
  - `temp` - Indicates that the specified operation is applied only to the temporary cache. The other cache locations aren't affected.

Options

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

  Prints out a short help for the command.

- `-c|--clear`

  The clear option executes a clear operation on the specified cache type. The contents of the cache directories are deleted recursively. The executing user/group must have permission to the files in the cache directories. If not, an error is displayed indicating the files/folders that weren't cleared.

- `-l|--list`

The list option is used to display the location of the specified cache type.

Examples

- Displays the paths of all the local cache directories (http-cache directory, global-packages cache directory, and temporary cache directory):

```
dotnet nuget locals -l all
```

- Displays the path for the local http-cache directory:

```
dotnet nuget locals --list http-cache
```

- Clears all files from all local cache directories (http-cache directory, global-packages cache directory, and temporary cache directory):

```
dotnet nuget locals --clear all
```

- Clears all files in local global-packages cache directory:

```
dotnet nuget locals -c global-packages
```

- Clears all files in local temporary cache directory:

```
dotnet nuget locals -c temp
```

Troubleshooting

For information on common problems and errors while using the `dotnet nuget locals` command, see Managing the NuGet cache.

dotnet nuget push

7/3/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget push` - Pushes a package to the server and publishes it.

Synopsis

```
dotnet nuget push [<ROOT>] [-d|--disable-buffering] [--force-english-output] [--interactive] [-k|--api-key] [-
n|--no-symbols]
    [--no-service-endpoint] [-s|--source] [-sk|--symbol-api-key] [-ss|--symbol-source] [-t|--timeout]
dotnet nuget push [-h|--help]
```

Description

The `dotnet nuget push` command pushes a package to the server and publishes it. The push command uses server and credential details found in the system's NuGet config file or chain of config files. For more information on config files, see Configuring NuGet Behavior. NuGet's default configuration is obtained by loading *%AppData%\NuGet\NuGet.config* (Windows) or *$HOME/.local/share* (Linux/macOS), then loading any *nuget.config* or *.nuget\nuget.config* starting from the root of drive and ending in the current directory.

Arguments

- `ROOT`

  Specifies the file path to the package to be pushed.

Options

- `-d|--disable-buffering`

  Disables buffering when pushing to an HTTP(S) server to reduce memory usage.

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

Prints out a short help for the command.

- `--interactive`

  Allows the command to block and requires manual action for operations like authentication. Option available since .NET Core 2.2 SDK.

- `-k|--api-key <API_KEY>`

  The API key for the server.

- `-n|--no-symbols`

  Doesn't push symbols (even if present).

- `--no-service-endpoint`

  Doesn't append "api/v2/package" to the source URL. Option available since .NET Core 2.1 SDK.

- `-s|--source <SOURCE>`

  Specifies the server URL. This option is required unless `DefaultPushSource` config value is set in the NuGet config file.

- `-sk|--symbol-api-key <API_KEY>`

  The API key for the symbol server.

- `-ss|--symbol-source <SOURCE>`

  Specifies the symbol server URL.

- `-t|--timeout <TIMEOUT>`

  Specifies the timeout for pushing to a server in seconds. Defaults to 300 seconds (5 minutes). Specifying 0 (zero seconds) applies the default value.

Examples

- Pushes *foo.nupkg* to the default push source, specifying an API key:

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a
```

- Push *foo.nupkg* to the custom push source `https://customsource`, specifying an API key:

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s https://customsource/
```

- Pushes *foo.nupkg* to the default push source:

```
dotnet nuget push foo.nupkg
```

- Pushes *foo.symbols.nupkg* to the default symbols source:

```
dotnet nuget push foo.symbols.nupkg
```

- Pushes *foo.nupkg* to the default push source, specifying a 360-second timeout:

```
dotnet nuget push foo.nupkg --timeout 360
```

- Pushes all *.nupkg* files in the current directory to the default push source:

```
dotnet nuget push *.nupkg
```

NOTE

If this command doesn't work, it might be due to a bug that existed in older versions of the SDK (.NET Core 2.1 SDK and earlier versions). To fix this, upgrade your SDK version or run the following command instead:

```
dotnet nuget push **/*.nupkg
```

dotnet pack

5/18/2019 • 4 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet pack` - Packs the code into a NuGet package.

Synopsis

- .NET Core 2.x
- .NET Core 1.x

```
dotnet pack [<PROJECT>] [-c|--configuration] [--force] [--include-source] [--include-symbols] [--no-build]
[--no-dependencies]
    [--no-restore] [-o|--output] [--runtime] [-s|--serviceable] [-v|--verbosity] [--version-suffix]
dotnet pack [-h|--help]
```

Description

The `dotnet pack` command builds the project and creates NuGet packages. The result of this command is a NuGet package. If the `--include-symbols` option is present, another package containing the debug symbols is created.

NuGet dependencies of the packed project are added to the *.nuspec* file, so they're properly resolved when the package is installed. Project-to-project references aren't packaged inside the project. Currently, you must have a package per project if you have project-to-project dependencies.

By default, `dotnet pack` builds the project first. If you wish to avoid this behavior, pass the `--no-build` option. This option is often useful in Continuous Integration (CI) build scenarios where you know the code was previously built.

You can provide MSBuild properties to the `dotnet pack` command for the packing process. For more information, see NuGet metadata properties and the MSBuild Command-Line Reference. The Examples section shows how to use the MSBuild -p switch for a couple of different scenarios.

Web projects aren't packable by default. To override the default behavior, add the following property to your *.csproj* file:

```
<PropertyGroup>
   <IsPackable>true</IsPackable>
</PropertyGroup>
```

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands, such as `dotnet build` and `dotnet run`, that require a restore to occur. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Arguments

- `PROJECT`

  The project to pack. It's either a path to a csproj file or to a directory. If not specified, it defaults to the current directory.

Options

- .NET Core 2.x
- .NET Core 1.x

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default value is `Debug`.

- `--force`

  Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

- `-h|--help`

  Prints out a short help for the command.

- `--include-source`

  Includes the source files in the NuGet package. The sources files are included in the `src` folder within the `nupkg`.

- `--include-symbols`

  Generates the symbols `nupkg`.

- `--no-build`

  Doesn't build the project before packing. It also implicitly sets the `--no-restore` flag.

- `--no-dependencies`

  Ignores project-to-project references and only restores the root project.

- `--no-restore`

  Doesn't execute an implicit restore when running the command.

- `-o|--output <OUTPUT_DIRECTORY>`

  Places the built packages in the directory specified.

- `--runtime <RUNTIME_IDENTIFIER>`

  Specifies the target runtime to restore packages for. For a list of Runtime Identifiers (RIDs), see the RID catalog.

- `-s|--serviceable`

  Sets the serviceable flag in the package. For more information, see .NET Blog: .NET 4.5.1 Supports Microsoft Security Updates for .NET NuGet Libraries.

- `--version-suffix <VERSION_SUFFIX>`

  Defines the value for the `$(VersionSuffix)` MSBuild property in the project.

- `-v|--verbosity <LEVEL>`

  Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

- Pack the project in the current directory:

```
dotnet pack
```

- Pack the `app1` project:

```
dotnet pack ~/projects/app1/project.csproj
```

- Pack the project in the current directory and place the resulting packages into the `nupkgs` folder:

```
dotnet pack --output nupkgs
```

- Pack the project in the current directory into the `nupkgs` folder and skip the build step:

```
dotnet pack --no-build --output nupkgs
```

- With the project's version suffix configured as `<VersionSuffix>$(VersionSuffix)</VersionSuffix>` in the *.csproj* file, pack the current project and update the resulting package version with the given suffix:

```
dotnet pack --version-suffix "ci-1234"
```

- Set the package version to `2.1.0` with the `PackageVersion` MSBuild property:

```
dotnet pack -p:PackageVersion=2.1.0
```

- Pack the project for a specific target framework:

```
dotnet pack -p:TargetFrameworks=net45
```

- Pack the project and use a specific runtime (Windows 10) for the restore operation (.NET Core SDK 2.0 and later versions):

```
dotnet pack --runtime win10-x64
```

- Pack the project using a .nuspec file:

```
dotnet pack ~/projects/app1/project.csproj /p:NuspecFile=~/projects/app1/project.nuspec
/p:NuspecBasePath=~/projects/app1/nuget
```

dotnet publish

7/19/2019 • 6 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet publish` - Packs the application and its dependencies into a folder for deployment to a hosting system.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet publish [<PROJECT>] [-c|--configuration] [-f|--framework] [--force] [--manifest] [--no-build] [--
no-dependencies]
    [--no-restore] [-o|--output] [-r|--runtime] [--self-contained] [-v|--verbosity] [--version-suffix]
dotnet publish [-h|--help]
```

Description

`dotnet publish` compiles the application, reads through its dependencies specified in the project file, and publishes the resulting set of files to a directory. The output includes the following assets:

- Intermediate Language (IL) code in an assembly with a *dll* extension.
- *.deps.json* file that includes all of the dependencies of the project.
- *.runtimeconfig.json* file that specifies the shared runtime that the application expects, as well as other configuration options for the runtime (for example, garbage collection type).
- The application's dependencies, which are copied from the NuGet cache into the output folder.

The `dotnet publish` command's output is ready for deployment to a hosting system (for example, a server, PC, Mac, laptop) for execution. It's the only officially supported way to prepare the application for deployment. Depending on the type of deployment that the project specifies, the hosting system may or may not have the .NET Core shared runtime installed on it. For more information, see .NET Core Application Deployment. For the directory structure of a published application, see Directory structure.

> **NOTE**
>
> Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands, such as `dotnet build` and `dotnet run`, that require a restore to occur. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.
>
> This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Arguments

`PROJECT`

The project to publish. It's either the path and filename of a C#, F#, or Visual Basic project file, or the path to a directory that contains a C#, F#, or Visual Basic project file. If not specified, it defaults to the current directory.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-f|--framework <FRAMEWORK>`

Publishes the application for the specified target framework. You must specify the target framework in the project file.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--manifest <PATH_TO_MANIFEST_FILE>`

Specifies one or several target manifests to use to trim the set of packages published with the app. The manifest file is part of the output of the `dotnet store` command. To specify multiple manifests, add a `--manifest` option for each manifest. This option is available starting with .NET Core 2.0 SDK.

`--no-build`

Doesn't build the project before publishing. It also implicitly sets the `--no-restore` flag.

`--no-dependencies`

Ignores project-to-project references and only restores the root project.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-o|--output <OUTPUT_DIRECTORY>`

Specifies the path for the output directory. If not specified, it defaults to *./bin/[configuration]/[framework]/publish/* for a framework-dependent deployment or *./bin/[configuration]/[framework]/[runtime]/publish/* for a self-contained deployment. If the path is relative, the output directory generated is relative to the project file location, not to the current working directory.

`--self-contained`

Publishes the .NET Core runtime with your application so the runtime doesn't need to be installed on the target machine. If a runtime identifier is specified, its default value is `true`. For more information about the different deployment types, see .NET Core application deployment.

`-r|--runtime <RUNTIME_IDENTIFIER>`

Publishes the application for a given runtime. This is used when creating a self-contained deployment (SCD). For a list of Runtime Identifiers (RIDs), see the RID catalog. Default is to publish a framework-dependent deployment (FDD).

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`--version-suffix <VERSION_SUFFIX>`

Defines the version suffix to replace the asterisk (`*`) in the version field of the project file.

Examples

Publish the project in the current directory:

```
dotnet publish
```

Publish the application using the specified project file:

```
dotnet publish ~/projects/app1/app1.csproj
```

Publish the project in the current directory using the `netcoreapp1.1` framework:

```
dotnet publish --framework netcoreapp1.1
```

Publish the current application using the `netcoreapp1.1` framework and the runtime for `OS X 10.10` (you must list this RID in the project file).

```
dotnet publish --framework netcoreapp1.1 --runtime osx.10.11-x64
```

Publish the current application but don't restore project-to-project (P2P) references, just the root project during the restore operation (.NET Core SDK 2.0 and later versions):

```
dotnet publish --no-dependencies
```

See also

- Target frameworks
- Runtime IDentifier (RID) catalog

dotnet restore

7/30/2019 • 5 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet restore` - Restores the dependencies and tools of a project.

Synopsis

- .NET Core 2.x
- .NET Core 1.x

```
dotnet restore [<ROOT>] [--configfile] [--disable-parallel] [--force] [--ignore-failed-sources] [--no-cache]
    [--no-dependencies] [--packages] [-r|--runtime] [-s|--source] [-v|--verbosity] [--interactive]
dotnet restore [-h|--help]
```

Description

The `dotnet restore` command uses NuGet to restore dependencies as well as project-specific tools that are specified in the project file. By default, the restoration of dependencies and tools are executed in parallel.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

To restore the dependencies, NuGet needs the feeds where the packages are located. Feeds are usually provided via the *nuget.config* configuration file. A default configuration file is provided when the CLI tools are installed. You specify additional feeds by creating your own *nuget.config* file in the project directory. You also specify additional feeds per invocation at a command prompt.

For dependencies, you specify where the restored packages are placed during the restore operation using the `--packages` argument. If not specified, the default NuGet package cache is used, which is found in the `.nuget/packages` directory in the user's home directory on all operating systems. For example, */home/user1* on Linux or *C:\Users\user1* on Windows.

For project-specific tooling, `dotnet restore` first restores the package in which the tool is packed, and then proceeds to restore the tool's dependencies as specified in its project file.

nuget.config differences

The behavior of the `dotnet restore` command is affected by the settings in the *nuget.config* file, if present. For example, setting the `globalPackagesFolder` in *nuget.config* places the restored NuGet packages in the specified folder. This is an alternative to specifying the `--packages` option on the

`dotnet restore` command. For more information, see the nuget.config reference.

There are three specific settings that `dotnet restore` ignores:

- bindingRedirects

  Binding redirects don't work with `<PackageReference>` elements and .NET Core only supports `<PackageReference>` elements for NuGet packages.

- solution

  This setting is Visual Studio specific and doesn't apply to .NET Core. .NET Core doesn't use a `packages.config` file and instead uses `<PackageReference>` elements for NuGet packages.

- trustedSigners

  This setting isn't applicable as NuGet doesn't yet support cross-platform verification of trusted packages.

Implicit `dotnet restore`

Starting with .NET Core 2.0, `dotnet restore` is run implicitly if necessary when you issue the following commands:

- `dotnet new`
- `dotnet build`
- `dotnet build-server`
- `dotnet run`
- `dotnet test`
- `dotnet publish`
- `dotnet pack`

In most cases, you no longer need to explicitly use the `dotnet restore` command.

Sometimes, it might be inconvenient to run `dotnet restore` implicitly. For example, some automated systems, such as build systems, need to call `dotnet restore` explicitly to control when the restore occurs so that they can control network usage. To prevent `dotnet restore` from running implicitly, you can use the `--no-restore` flag with any of these commands to disable implicit restore.

Arguments

`ROOT`

Optional path to the project file to restore.

Options

- .NET Core 2.x
- .NET Core 1.x

`--configfile <FILE>`

The NuGet configuration file (*nuget.config*) to use for the restore operation.

`--disable-parallel`

Disables restoring multiple projects in parallel.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--ignore-failed-sources`

Only warn about failed sources if there are packages meeting the version requirement.

`--no-cache`

Specifies to not cache packages and HTTP requests.

`--no-dependencies`

When restoring a project with project-to-project (P2P) references, restores the root project and not the references.

`--packages <PACKAGES_DIRECTORY>`

Specifies the directory for restored packages.

`-r|--runtime <RUNTIME_IDENTIFIER>`

Specifies a runtime for the package restore. This is used to restore packages for runtimes not explicitly listed in the `<RuntimeIdentifiers>` tag in the *.csproj* file. For a list of Runtime Identifiers (RIDs), see the RID catalog. Provide multiple RIDs by specifying this option multiple times.

`-s|--source <SOURCE>`

Specifies a NuGet package source to use during the restore operation. This setting overrides all of the sources specified in the *nuget.config* files. Multiple sources can be provided by specifying this option multiple times.

`--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`--interactive`

Allows the command to stop and wait for user input or action (for example to complete authentication). Since .NET Core 2.1.400.

Examples

Restore dependencies and tools for the project in the current directory:

```
dotnet restore
```

Restore dependencies and tools for the `app1` project found in the given path:

```
dotnet restore ~/projects/app1/app1.csproj
```

Restore the dependencies and tools for the project in the current directory using the file path provided as the source:

```
dotnet restore -s c:\packages\mypackages
```

Restore the dependencies and tools for the project in the current directory using the two file paths provided as sources:

```
dotnet restore -s c:\packages\mypackages -s c:\packages\myotherpackages
```

Restore dependencies and tools for the project in the current directory and shows only minimal output:

```
dotnet restore --verbosity minimal
```

dotnet run

5/15/2019 • 5 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet run` - Runs source code without any explicit compile or launch commands.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet run [-c|--configuration] [-f|--framework] [--force] [--launch-profile] [--no-build] [--no-
dependencies]
    [--no-launch-profile] [--no-restore] [-p|--project] [--runtime] [-v|--verbosity] [[--] [application
arguments]]
dotnet run [-h|--help]
```

Description

The `dotnet run` command provides a convenient option to run your application from the source code with one command. It's useful for fast iterative development from the command line. The command depends on the `dotnet build` command to build the code. Any requirements for the build, such as that the project must be restored first, apply to `dotnet run` as well.

Output files are written into the default location, which is `bin/<configuration>/<target>`. For example if you have a `netcoreapp2.1` application and you run `dotnet run`, the output is placed in `bin/Debug/netcoreapp2.1`. Files are overwritten as needed. Temporary files are placed in the `obj` directory.

If the project specifies multiple frameworks, executing `dotnet run` results in an error unless the `-f|--framework <FRAMEWORK>` option is used to specify the framework.

The `dotnet run` command is used in the context of projects, not built assemblies. If you're trying to run a framework-dependent application DLL instead, you must use dotnet without a command. For example, to run `myapp.dll`, use:

```
dotnet myapp.dll
```

For more information on the `dotnet` driver, see the .NET Core Command Line Tools (CLI) topic.

To run the application, the `dotnet run` command resolves the dependencies of the application that are outside of the shared runtime from the NuGet cache. Because it uses cached dependencies, it's not recommended to use `dotnet run` to run applications in production. Instead, create a deployment using the `dotnet publish` command and deploy the published output.

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands, such as `dotnet build` and `dotnet run`, that require a restore to occur. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--`

Delimits arguments to `dotnet run` from arguments for the application being run. All arguments after this delimiter are passed to the application run.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-f|--framework <FRAMEWORK>`

Builds and runs the app using the specified framework. The framework must be specified in the project file.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--launch-profile <NAME>`

The name of the launch profile (if any) to use when launching the application. Launch profiles are defined in the *launchSettings.json* file and are typically called `Development`, `Staging`, and `Production`. For more information, see Working with multiple environments.

`--no-build`

Doesn't build the project before running. It also implicit sets the `--no-restore` flag.

`--no-dependencies`

When restoring a project with project-to-project (P2P) references, restores the root project and not the references.

`--no-launch-profile`

Doesn't try to use *launchSettings.json* to configure the application.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-p|--project <PATH>`

Specifies the path of the project file to run (folder name or full path). If not specified, it defaults to the current directory.

`--runtime <RUNTIME_IDENTIFIER>`

Specifies the target runtime to restore packages for. For a list of Runtime Identifiers (RIDs), see the RID catalog.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

Run the project in the current directory:

```
dotnet run
```

Run the specified project:

```
dotnet run --project ./projects/proj1/proj1.csproj
```

Run the project in the current directory (the `--help` argument in this example is passed to the application, since the blank `--` option is used):

```
dotnet run --configuration Release -- --help
```

Restore dependencies and tools for the project in the current directory only showing minimal output and then run the project: (.NET Core SDK 2.0 and later versions):

```
dotnet run --verbosity m
```

dotnet sln

12/10/2018 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet sln` - Modifies a .NET Core solution file.

Synopsis

```
dotnet sln [<SOLUTION_NAME>] add <PROJECT> <PROJECT> ...
dotnet sln [<SOLUTION_NAME>] add <GLOBBING_PATTERN>
dotnet sln [<SOLUTION_NAME>] remove <PROJECT> <PROJECT> ...
dotnet sln [<SOLUTION_NAME>] remove <GLOBBING_PATTERN>
dotnet sln [<SOLUTION_NAME>] list
dotnet sln [-h|--help]
```

Description

The `dotnet sln` command provides a convenient way to add, remove, and list projects in a solution file.

To use the `dotnet sln` command, the solution file must already exist. If you need to create one, use the dotnet new command, like in the following example:

```
dotnet new sln
```

Commands

`add <PROJECT> ...`

`add <GLOBBING_PATTERN>`

Adds a project or multiple projects to the solution file. Globbing patterns are supported on Unix/Linux based terminals.

`remove <PROJECT> ...`

`remove <GLOBBING_PATTERN>`

Removes a project or multiple projects from the solution file. Globbing patterns are supported on Unix/Linux based terminals.

`list`

Lists all projects in a solution file.

Arguments

`SOLUTION_NAME`

Solution file to use. If not specified, the command searches the current directory for one. If there are multiple

solution files in the directory, one must be specified.

Options

`-h|--help`

Prints out a short help for the command.

Examples

Add a C# project to a solution:

```
dotnet sln todo.sln add todo-app/todo-app.csproj
```

Remove a C# project from a solution:

```
dotnet sln todo.sln remove todo-app/todo-app.csproj
```

Add multiple C# projects to a solution:

```
dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj
```

Remove multiple C# projects from a solution:

```
dotnet sln todo.sln remove todo-app/todo-app.csproj back-end/back-end.csproj
```

Add multiple C# projects to a solution using a globbing pattern:

```
dotnet sln todo.sln add **/*.csproj
```

Remove multiple C# projects from a solution using a globbing pattern:

```
dotnet sln todo.sln remove **/*.csproj
```

> **NOTE**
> Globbing is not a CLI feature but rather a feature of the command shell. To successfully expand the files, you must use a shell that supports globbing. For more information about globbing, see Wikipedia.

dotnet store

1/23/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 2.0 SDK and later versions

Name

`dotnet store` - Stores the specified assemblies in the runtime package store.

Synopsis

```
dotnet store -m|--manifest -f|--framework -r|--runtime [--framework-version] [-h|--help] [--output] [--skip-optimization] [--skip-symbols] [-v|--verbosity] [--working-dir]
```

Description

`dotnet store` stores the specified assemblies in the runtime package store. By default, assemblies are optimized for the target runtime and framework. For more information, see the runtime package store topic.

Required options

`-f|--framework <FRAMEWORK>`

Specifies the target framework.

`-m|--manifest <PATH_TO_MANIFEST_FILE>`

The *package store manifest file* is an XML file that contains the list of packages to store. The format of the manifest file is compatible with the SDK-style project format. So, a project file that references the desired packages can be used with the `-m|--manifest` option to store assemblies in the runtime package store. To specify multiple manifest files, repeat the option and path for each file. For example: `--manifest packages1.csproj --manifest packages2.csproj`.

`-r|--runtime <RUNTIME_IDENTIFIER>`

The runtime identifier to target.

Optional options

`--framework-version <FRAMEWORK_VERSION>`

Specifies the .NET Core SDK version. This option enables you to select a specific framework version beyond the framework specified by the `-f|--framework` option.

`-h|--help`

Shows help information.

`-o|--output <OUTPUT_DIRECTORY>`

Specifies the path to the runtime package store. If not specified, it defaults to the *store* subdirectory of the user profile .NET Core installation directory.

`--skip-optimization`

Skips the optimization phase.

`--skip-symbols`

Skips symbol generation. Currently, you can only generate symbols on Windows and Linux.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`-w|--working-dir <INTERMEDIATE_WORKING_DIRECTORY>`

The working directory used by the command. If not specified, it uses the *obj* subdirectory of the current directory.

Examples

Store the packages specified in the *packages.csproj* project file for .NET Core 2.0.0:

```
dotnet store --manifest packages.csproj --framework-version 2.0.0
```

Store the packages specified in the *packages.csproj* without optimization:

```
dotnet store --manifest packages.csproj --skip-optimization
```

See also

- Runtime package store

dotnet test

6/27/2019 • 6 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet test` - .NET test driver used to execute unit tests.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet test [<PROJECT>] [-a|--test-adapter-path] [--blame] [-c|--configuration] [--collect] [-d|--diag] [-
f|--framework] [--filter]
    [-l|--logger] [--no-build] [--no-restore] [-o|--output] [-r|--results-directory] [-s|--settings] [-t|--
list-tests]
    [-v|--verbosity] [-- <RunSettings arguments>]

dotnet test [-h|--help]
```

Description

The `dotnet test` command is used to execute unit tests in a given project. The `dotnet test` command launches the test runner console application specified for a project. The test runner executes the tests defined for a unit test framework (for example, MSTest, NUnit, or xUnit) and reports the success or failure of each test. If all tests are successful, the test runner returns 0 as an exit code; otherwise if any test fails, it returns 1. The test runner and the unit test library are packaged as NuGet packages and are restored as ordinary dependencies for the project.

Test projects specify the test runner using an ordinary `<PackageReference>` element, as seen in the following sample project file:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.2.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>

</Project>
```

Arguments

`PROJECT`

Path to the test project. If not specified, it defaults to current directory.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`-a|--test-adapter-path <PATH_TO_ADAPTER>`

Use the custom test adapters from the specified path in the test run.

`--blame`

Runs the tests in blame mode. This option is helpful in isolating the problematic tests causing test host to crash. It creates an output file in the current directory as *Sequence.xml* that captures the order of tests execution before the crash.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`, but your project's configuration could override this default SDK setting.

`--collect <DATA_COLLECTOR_FRIENDLY_NAME>`

Enables data collector for the test run. For more information, see Monitor and analyze test run.

`-d|--diag <PATH_TO_DIAGNOSTICS_FILE>`

Enables diagnostic mode for the test platform and write diagnostic messages to the specified file.

`-f|--framework <FRAMEWORK>`

Looks for test binaries for a specific framework.

`--filter <EXPRESSION>`

Filters out tests in the current project using the given expression. For more information, see the Filter option details section. For more information and examples on how to use selective unit test filtering, see Running selective unit tests.

`-h|--help`

Prints out a short help for the command.

`-l|--logger <LoggerUri/FriendlyName>`

Specifies a logger for test results.

`--no-build`

Doesn't build the test project before running it. It also implicit sets the `--no-restore` flag.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-o|--output <OUTPUT_DIRECTORY>`

Directory in which to find the binaries to run.

`-r|--results-directory <PATH>`

The directory where the test results are going to be placed. If the specified directory doesn't exist, it's created.

`-s|--settings <SETTINGS_FILE>`

The `.runsettings` file to use for running the tests. Configure unit tests by using a `.runsettings` file.

`-t|--list-tests`

List all of the discovered tests in the current project.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`RunSettings arguments`

Arguments passed as RunSettings configurations for the test. Arguments are specified as `[name]=[value]` pairs after "-- " (note the space after --). A space is used to separate multiple `[name]=[value]` pairs.

Example: `dotnet test -- MSTest.DeploymentEnabled=false MSTest.MapInconclusiveToFailed=True`

For more information about RunSettings, see vstest.console.exe: Passing RunSettings args.

Examples

Run the tests in the project in the current directory:

```
dotnet test
```

Run the tests in the `test1` project:

```
dotnet test ~/projects/test1/test1.csproj
```

Run the tests in the project in the current directory and generate a test results file in the trx format:

```
dotnet test --logger:trx
```

Filter option details

`--filter <EXPRESSION>`

`<Expression>` has the format `<property><operator><value>[|&<Expression>]`.

`<property>` is an attribute of the `Test Case`. The following are the properties supported by popular unit test frameworks:

TEST FRAMEWORK	SUPPORTED PROPERTIES
MSTest	• FullyQualifiedName • Name • ClassName • Priority • TestCategory
xUnit	• FullyQualifiedName • DisplayName • Traits

The `<operator>` describes the relationship between the property and the value:

OPERATOR	FUNCTION
`=`	Exact match
`!=`	Not exact match
`~`	Contains

`<value>` is a string. All the lookups are case insensitive.

An expression without an `<operator>` is automatically considered as a `contains` on `FullyQualifiedName` property (for example, `dotnet test --filter xyz` is same as `dotnet test --filter FullyQualifiedName~xyz`).

Expressions can be joined with conditional operators:

OPERATOR	FUNCTION
`|`	OR
`&`	AND

You can enclose expressions in parenthesis when using conditional operators (for example, `(Name~TestMethod1) | (Name~TestMethod2)`).

For more information and examples on how to use selective unit test filtering, see Running selective unit tests.

See also

- Frameworks and Targets
- .NET Core Runtime IDentifier (RID) catalog

dotnet tool install

1/23/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool install` - Installs the specified .NET Core Global Tool on your machine.

Synopsis

```
dotnet tool install <PACKAGE_NAME> <-g|--global> [--add-source] [--configfile] [--framework] [-v|--verbosity]
[--version]
dotnet tool install <PACKAGE_NAME> <--tool-path> [--add-source] [--configfile] [--framework] [-v|--verbosity]
[--version]
dotnet tool install <-h|--help>
```

Description

The `dotnet tool install` command provides a way for you to install .NET Core Global Tools on your machine. To use the command, you either have to specify that you want a user-wide installation using the `--global` option or you specify a path to install it using the `--tool-path` option.

Global Tools are installed in the following directories by default when you specify the `-g` (or `--global`) option:

OS	PATH
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to install.

Options

`--add-source <SOURCE>`

Adds an additional NuGet package source to use during installation.

`--configfile <FILE>`

The NuGet configuration (*nuget.config*) file to use.

`--framework <FRAMEWORK>`

Specifies the target framework to install the tool for. By default, the .NET Core SDK tries to choose the most appropriate target framework.

`-g|--global`

Specifies that the installation is user wide. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where to install the Global Tool. PATH can be absolute or relative. If PATH doesn't exist, the command tries to create it. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`--version <VERSION_NUMBER>`

The version of the tool to install. By default, the latest stable package version is installed. Use this option to install preview or older versions of the tool.

Examples

Installs the dotnetsay Global Tool in the default location:

```
dotnet tool install -g dotnetsay
```

Installs the dotnetsay Global Tool on a specific Windows folder:

```
dotnet tool install dotnetsay --tool-path c:\global-tools
```

Installs the dotnetsay Global Tool on a specific Linux/macOS folder:

```
dotnet tool install dotnetsay --tool-path ~/bin
```

Installs version 2.0.0 of the dotnetsay Global Tool:

```
dotnet tool install -g dotnetsay --version 2.0.0
```

See also

- .NET Core Global Tools

dotnet tool list

1/23/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool list` - Lists all .NET Core Global Tools currently installed in the default directory on your machine or in the specified path.

Synopsis

```
dotnet tool list <-g|--global>
dotnet tool list <--tool-path>
dotnet tool list <-h|--help>
```

Description

The `dotnet tool list` command provides a way for you to list all .NET Core Global Tools installed user-wide on your machine (current user profile) or in the specified path. The command lists the package name, version installed, and the Global Tool command. To use the list command, you either have to specify that you want to see all user-wide tools using the `--global` option or specify a custom path using the `--tool-path` option.

Options

`-g|--global`

Lists user-wide Global Tools. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies a custom location where to find Global Tools. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

Examples

Lists all Global Tools installed user-wide on your machine (current user profile):

`dotnet tool list -g`

Lists the Global Tools from a specific Windows folder:

`dotnet tool list --tool-path c:\global-tools`

Lists the Global Tools from a specific Linux/macOS folder:

`dotnet tool list --tool-path ~/bin`

See also

- .NET Core Global Tools

dotnet tool uninstall

1/23/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool uninstall` - Uninstalls the specified .NET Core Global Tool from your machine.

Synopsis

```
dotnet tool uninstall <PACKAGE_NAME> <-g|--global>
dotnet tool uninstall <PACKAGE_NAME> <--tool-path>
dotnet tool uninstall <-h|--help>
```

Description

The `dotnet tool uninstall` command provides a way for you to uninstall .NET Core Global Tools from your machine. To use the command, you either have to specify that you want to remove a user-wide tool using the `--global` option or specify a path to where the tool is installed using the `--tool-path` option.

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to uninstall. You can find the package name using the dotnet tool list command.

Options

`-g|--global`

Specifies that the tool to be removed is from a user-wide installation. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where to uninstall the Global Tool. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

Examples

Uninstalls the dotnetsay Global Tool:

```
dotnet tool uninstall -g dotnetsay
```

Uninstalls the dotnetsay Global Tool from a specific Windows folder:

```
dotnet tool uninstall dotnetsay --tool-path c:\global-tools
```

Uninstalls the dotnetsay Global Tool from a specific Linux/macOS folder:

```
dotnet tool uninstall dotnetsay --tool-path ~/bin
```

See also

- .NET Core Global Tools

dotnet tool update

1/23/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool update` - Updates the specified .NET Core Global Tool on your machine.

Synopsis

```
dotnet tool update <PACKAGE_NAME> <-g|--global> [--configfile] [--framework] [-v|--verbosity]
dotnet tool update <PACKAGE_NAME> <--tool-path> [--configfile] [--framework] [-v|--verbosity]
dotnet tool update <-h|--help>
```

Description

The `dotnet tool update` command provides a way for you to update .NET Core Global Tools on your machine to the latest stable version of the package. The command uninstalls and reinstalls a tool, effectively updating it. To use the command, you either have to specify that you want to update a tool from a user-wide installation using the `--global` option or specify a path to where the tool is installed using the `--tool-path` option.

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to update. You can find the package name using the dotnet tool list command.

Options

`--add-source <SOURCE>`

Adds an additional NuGet package source to use during installation.

`--configfile <FILE>`

The NuGet configuration (*nuget.config*) file to use.

`--framework <FRAMEWORK>`

Specifies the target framework to update the tool for.

`-g|--global`

Specifies that the update is for a user-wide tool. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where the Global Tool is installed. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

Updates the dotnetsay Global Tool:

```
dotnet tool update -g dotnetsay
```

Updates the dotnetsay Global Tool located on a specific Windows folder:

```
dotnet tool update dotnetsay --tool-path c:\global-tools
```

Updates the dotnetsay Global Tool located on a specific Linux/macOS folder:

```
dotnet tool update dotnetsay --tool-path ~/bin
```

See also

- .NET Core Global Tools

dotnet vstest

5/29/2019 • 6 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet-vstest` - Runs tests from the specified files.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet vstest [<TEST_FILE_NAMES>] [--Settings|/Settings] [--Tests|/Tests] [--TestAdapterPath|/TestAdapterPath]
    [--Platform|/Platform] [--Framework|/Framework] [--Parallel|/Parallel] [--TestCaseFilter|/TestCaseFilter]
[--logger|/logger]
    [-lt|--ListTests|/lt|/ListTests] [--ParentProcessId|/ParentProcessId] [--Port|/Port] [--Diag|/Diag] [--
Blame|/Blame] [--InIsolation|/InIsolation]
    [[--] <args>...]] [-?|--Help|/?|/Help]
```

Description

The `dotnet-vstest` command runs the `VSTest.Console` command-line application to run automated unit tests.

Arguments

`TEST_FILE_NAMES`

Run tests from the specified assemblies. Separate multiple test assembly names with spaces.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--Settings|/Settings:<Settings File>`

Settings to use when running tests.

`--Tests|/Tests:<Test Names>`

Run tests with names that match the provided values. Separate multiple values with commas.

`--TestAdapterPath|/TestAdapterPath`

Use custom test adapters from a given path (if any) in the test run.

`--Platform|/Platform:<Platform type>`

Target platform architecture used for test execution. Valid values are `x86`, `x64`, and `ARM`.

`--Framework|/Framework:<Framework Version>`

Target .NET Framework version used for test execution. Examples of valid values are `.NETFramework,Version=v4.6` or `.NETCoreApp,Version=v1.0`. Other supported values are `Framework40`, `Framework45`, `FrameworkCore10`, and `FrameworkUap10`.

`--Parallel|/Parallel`

Execute tests in parallel. By default, all available cores on the machine are available for use. Specify an explicit number of cores by setting the MaxCpuCount property under the RunConfiguration node in the runsettings file.

`--TestCaseFilter|/TestCaseFilter:<Expression>`

Run tests that match the given expression. `<Expression>` is of the format `<property>Operator<value>[|&<Expression>]`, where Operator is one of `=`, `!=`, or `~`. Operator `~` has 'contains' semantics and is applicable for string properties like `DisplayName`. Parenthesis `()` are used to group sub-expressions.

`-?|--Help|/?|/Help`

Prints out a short help for the command.

`--logger|/logger:<Logger Uri/FriendlyName>`

Specify a logger for test results.

- To publish test results to Team Foundation Server, use the `TfsPublisher` logger provider:

  ```
  /logger:TfsPublisher;
      Collection=<team project collection url>;
      BuildName=<build name>;
      TeamProject=<team project name>
      [;Platform=<Defaults to "Any CPU">]
      [;Flavor=<Defaults to "Debug">]
      [;RunTitle=<title>]
  ```

- To log results to a Visual Studio Test Results File (TRX), use the `trx` logger provider. This switch creates a file in the test results directory with given log file name. If `LogFileName` isn't provided, a unique file name is created to hold the test results.

  ```
  /logger:trx [;LogFileName=<Defaults to unique file name>]
  ```

`-lt|--ListTests|/lt|/ListTests:<File Name>`

Lists all discovered tests from the given test container.

`--ParentProcessId|/ParentProcessId:<ParentProcessId>`

Process ID of the parent process responsible for launching the current process.

`--Port|/Port:<Port>`

Specifies the port for the socket connection and receiving the event messages.

`--Diag|/Diag:<Path to log file>`

Enables verbose logs for the test platform. Logs are written to the provided file.

`--Blame|/Blame`

Runs the tests in blame mode. This option is helpful in isolating the problematic tests causing test host to crash. It creates an output file in the current directory as *Sequence.xml* that captures the order of tests execution before the crash.

`--InIsolation|/InIsolation`

Runs the tests in an isolated process. This makes *vstest.console.exe* process less likely to be stopped on an error in the tests, but tests may run slower.

`@<file>`

Reads response file for more options.

`args`

Specifies extra arguments to pass to the adapter. Arguments are specified as name-value pairs of the form `<n>=<v>`, where `<n>` is the argument name and `<v>` is the argument value. Use a space to separate multiple arguments.

Examples

Run tests in `mytestproject.dll`:

```
dotnet vstest mytestproject.dll
```

Run tests in `mytestproject.dll`, exporting to custom folder with custom name:

```
dotnet vstest mytestproject.dll --logger:"trx;LogFileName=custom_file_name.trx" --
ResultsDirectory:custom/file/path
```

Run tests in `mytestproject.dll` and `myothertestproject.exe`:

```
dotnet vstest mytestproject.dll myothertestproject.exe
```

Run `TestMethod1` tests:

```
dotnet vstest /Tests:TestMethod1
```

Run `TestMethod1` and `TestMethod2` tests:

```
dotnet vstest /Tests:TestMethod1,TestMethod2
```

dotnet-install scripts reference

1/29/2019 • 4 minutes to read • Edit Online

Name

`dotnet-install.ps1` | `dotnet-install.sh` - Script used to install the .NET Core CLI tools and the shared runtime.

Synopsis

Windows:

```
dotnet-install.ps1 [-Channel] [-Version] [-InstallDir] [-Architecture] [-SharedRuntime] [-Runtime] [-DryRun]
[-NoPath] [-Verbose] [-AzureFeed] [-UncachedFeed] [-NoCdn] [-FeedCredential] [-ProxyAddress] [-
ProxyUseDefaultCredentials] [-SkipNonVersionedFiles] [-Help]
```

macOS/Linux:

```
dotnet-install.sh [--channel] [--version] [--install-dir] [--architecture] [--runtime] [--dry-run] [--no-path]
[--verbose] [--azure-feed] [--uncached-feed] [--no-cdn] [--feed-credential] [--runtime-id] [--skip-non-
versioned-files] [--help]
```

Description

The `dotnet-install` scripts are used to perform a non-admin installation of the .NET Core SDK, which includes the .NET Core CLI tools and the shared runtime.

We recommend that you use the stable version that is hosted on .NET Core main website. The direct paths to the scripts are:

- https://dot.net/v1/dotnet-install.sh (bash, UNIX)
- https://dot.net/v1/dotnet-install.ps1 (Powershell, Windows)

The main usefulness of these scripts is in automation scenarios and non-admin installations. There are two scripts: one is a PowerShell script that works on Windows, and the other is a bash script that works on Linux/macOS. Both scripts have the same behavior. The bash script also reads PowerShell switches, so you can use PowerShell switches with the script on Linux/macOS systems.

The installation scripts download the ZIP/tarball file from the CLI build drops and proceed to install it in either the default location or in a location specified by `-InstallDir|--install-dir`. By default, the installation scripts download the SDK and install it. If you wish to only obtain the shared runtime, specify the `--runtime` argument.

By default, the script adds the install location to the $PATH for the current session. Override this default behavior by specifying the `--no-path` argument.

Before running the script, install the required dependencies.

You can install a specific version using the `--version` argument. The version must be specified as a three-part version (for example, 1.0.0-13232). If not provided, it uses the `latest` version.

Options

- `-Channel <CHANNEL>`

  Specifies the source channel for the installation. The possible values are:

- `Current` - Most current release.
- `LTS` - Long-Term Support channel (most current supported release).
- Two-part version in X.Y format representing a specific release (for example, `2.0` or `1.0`).
- Branch name. For example, `release/2.0.0`, `release/2.0.0-preview2`, or `master` (for nightly releases).

The default value is `LTS`. For more information on .NET support channels, see the .NET Support Policy page.

- `-Version <VERSION>`

  Represents a specific build version. The possible values are:

  - `latest` - Latest build on the channel (used with the `-Channel` option).
  - `coherent` - Latest coherent build on the channel; uses the latest stable package combination (used with Branch name `-Channel` options).
  - Three-part version in X.Y.Z format representing a specific build version; supersedes the `-Channel` option. For example: `2.0.0-preview2-006120`.

  If not specified, `-Version` defaults to `latest`.

- `-InstallDir <DIRECTORY>`

  Specifies the installation path. The directory is created if it doesn't exist. The default value is *%LocalAppData%\Microsoft\dotnet*. Binaries are placed directly in this directory.

- `-Architecture <ARCHITECTURE>`

  Architecture of the .NET Core binaries to install. Possible values are `<auto>`, `amd64`, `x64`, `x86`, `arm64`, and `arm`. The default value is `<auto>`, which represents the currently running OS architecture.

- `-SharedRuntime`

  > **NOTE**
  > This parameter is obsolete and may be removed in a future version of the script. The recommended alternative is the `Runtime` option.

  Installs just the shared runtime bits, not the entire SDK. This is equivalent to specifying `-Runtime dotnet`.

- `-Runtime <RUNTIME>`

  Installs just the shared runtime, not the entire SDK. The possible values are:

  - `dotnet` - the `Microsoft.NETCore.App` shared runtime.
  - `aspnetcore` - the `Microsoft.AspNetCore.App` shared runtime.

- `-DryRun`

  If set, the script won't perform the installation. Instead, it displays what command line to use to consistently install the currently requested version of the .NET Core CLI. For example, if you specify version `latest`, it displays a link with the specific version so that this command can be used deterministically in a build script. It also displays the binary's location if you prefer to install or download it yourself.

- `-NoPath`

  If set, the installation folder isn't exported to the path for the current session. By default, the script modifies the PATH, which makes the CLI tools available immediately after install.

- `-Verbose`

  Displays diagnostics information.

- `-AzureFeed`

  Specifies the URL for the Azure feed to the installer. We recommended that you don't change this value. The default value is `https://dotnetcli.azureedge.net/dotnet`.

- `-UncachedFeed`

  Allows changing the URL for the uncached feed used by this installer. We recommended that you don't change this value.

- `-NoCdn`

  Disables downloading from the Azure Content Delivery Network (CDN) and uses the uncached feed directly.

- `-FeedCredential`

  Used as a query string to append to the Azure feed. It allows changing the URL to use non-public blob storage accounts.

- `-ProxyAddress`

  If set, the installer uses the proxy when making web requests. (Only valid for Windows)

- `ProxyUseDefaultCredentials`

  If set, the installer uses the credentials of the current user when using proxy address. (Only valid for Windows)

- `-SkipNonVersionedFiles`

  Skips installing non-versioned files, such as *dotnet.exe*, if they already exist.

- `-Help`

  Prints out help for the script.

Examples

- Install the latest long-term supported (LTS) version to the default location:

  Windows:

  ```
  ./dotnet-install.ps1 -Channel LTS
  ```

  macOS/Linux:

  ```
  ./dotnet-install.sh --channel LTS
  ```

- Install the latest version from 2.0 channel to the specified location:

  Windows:

  ```
  ./dotnet-install.ps1 -Channel 2.0 -InstallDir C:\cli
  ```

macOS/Linux:

```
./dotnet-install.sh --channel 2.0 --install-dir ~/cli
```

- Install the 1.1.0 version of the shared runtime:

  Windows:

  ```
  ./dotnet-install.ps1 -Runtime dotnet -Version 1.1.0
  ```

  macOS/Linux:

  ```
  ./dotnet-install.sh --runtime dotnet --version 1.1.0
  ```

- Obtain script and install the 2.1.2 version behind a corporate proxy (Windows only):

  ```
  Invoke-WebRequest 'https://dot.net/v1/dotnet-install.ps1' -Proxy $env:HTTP_PROXY -
  ProxyUseDefaultCredentials -OutFile 'dotnet-install.ps1';
  ./dotnet-install.ps1 -InstallDir '~/.dotnet' -Version '2.1.2' -ProxyAddress $env:HTTP_PROXY -
  ProxyUseDefaultCredentials;
  ```

- Obtain script and install .NET Core CLI one-liner examples:

  Windows:

  ```
  @powershell -NoProfile -ExecutionPolicy unrestricted -Command "
  [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; &
  ([scriptblock]::Create((Invoke-WebRequest -useb 'https://dot.net/v1/dotnet-install.ps1'))) <additional
  install-script args>"
  ```

  macOS/Linux:

  ```
  curl -sSL https://dot.net/v1/dotnet-install.sh | bash /dev/stdin <additional install-script args>
  ```

See also

- .NET Core releases
- .NET Core Runtime and SDK download archive

dotnet-add reference

6/27/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet add reference` - Adds project-to-project (P2P) references.

Synopsis

```
dotnet add [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help] [--interactive]
```

Description

The `dotnet add reference` command provides a convenient option to add project references to a project. After running the command, the `<ProjectReference>` elements are added to the project file.

```
<ItemGroup>
  <ProjectReference Include="app.csproj" />
  <ProjectReference Include="..\lib2\lib2.csproj" />
  <ProjectReference Include="..\lib1\lib1.csproj" />
</ItemGroup>
```

Arguments

- `PROJECT`

  Specifies the project file. If not specified, the command searches the current directory for one.

- `PROJECT_REFERENCES`

  Project-to-project (P2P) references to add. Specify one or more projects. Glob patterns are supported on Unix/Linux-based systems.

Options

- `-h|--help`

  Prints out a short help for the command.

- `-f|--framework <FRAMEWORK>`

  Adds project references only when targeting a specific framework.

- `--interactive`

  Allows the command to stop and wait for user input or action (for example, to complete authentication). Available since .NET Core 3.0 SDK.

Examples

- Add a project reference:

```
dotnet add app/app.csproj reference lib/lib.csproj
```

- Add multiple project references to the project in the current directory:

```
dotnet add reference lib1/lib1.csproj lib2/lib2.csproj
```

- Add multiple project references using a globbing pattern on Linux/Unix:

```
dotnet add app/app.csproj reference **/*.csproj
```

dotnet list reference

6/27/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet list reference` - Lists project-to-project references.

Synopsis

```
dotnet list [<PROJECT>|<SOLUTION>] reference [-h|--help]
```

Description

The `dotnet list reference` command provides a convenient option to list project references for a given project or solution.

Arguments

- `PROJECT | SOLUTION`

  Specifies the project or solution file to use for listing references. If not specified, the command searches the current directory for a project file.

Options

- `-h|--help`

  Prints out a short help for the command.

Examples

- List the project references for the specified project:

  ```
  dotnet list app/app.csproj reference
  ```

- List the project references for the project in the current directory:

  ```
  dotnet list reference
  ```

dotnet remove reference

12/10/2018 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet remove reference` - Removes project-to-project references.

Synopsis

```
dotnet remove [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help]
```

Description

The `dotnet remove reference` command provides a convenient option to remove project references from a project.

Arguments

`PROJECT`

Target project file. If not specified, the command searches the current directory for one.

`PROJECT_REFERENCES`

Project-to-project (P2P) references to remove. You can specify one or multiple projects. Glob patterns are supported on Unix/Linux based terminals.

Options

`-h|--help`

Prints out a short help for the command.

`-f|--framework <FRAMEWORK>`

Removes the reference only when targeting a specific framework.

Examples

Remove a project reference from the specified project:

```
dotnet remove app/app.csproj reference lib/lib.csproj
```

Remove multiple project references from the project in the current directory:

```
dotnet remove reference lib1/lib1.csproj lib2/lib2.csproj
```

Remove multiple project references using a glob pattern on Unix/Linux:

```
dotnet remove app/app.csproj reference **/*.csproj
```

dotnet add package

7/5/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet add package` - Adds a package reference to a project file.

Synopsis

```
dotnet add [<PROJECT>] package <PACKAGE_NAME> [-h|--help] [-f|--framework] [--interactive] [-n|--no-restore]
[--package-directory] [-s|--source] [-v|--version]
```

Description

The `dotnet add package` command provides a convenient option to add a package reference to a project file. After running the command, there's a compatibility check to ensure the package is compatible with the frameworks in the project. If the check passes, a `<PackageReference>` element is added to the project file and dotnet restore is run.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

For example, adding `Newtonsoft.Json` to *ToDo.csproj* produces output similar to the following example:

```
  Writing C:\Users\mairaw\AppData\Local\Temp\tmp95A8.tmp
info : Adding PackageReference for package 'Newtonsoft.Json' into project 'C:\projects\ToDo\ToDo.csproj'.
log  : Restoring packages for C:\Temp\projects\consoleproj\consoleproj.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 79ms
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg 232ms
log  : Installing Newtonsoft.Json 12.0.1.
info : Package 'Newtonsoft.Json' is compatible with all the specified frameworks in project
'C:\projects\ToDo\ToDo.csproj'.
info : PackageReference for package 'Newtonsoft.Json' version '12.0.1' added to file
'C:\projects\ToDo\ToDo.csproj'.
```

The *ToDo.csproj* file now contains a `<PackageReference>` element for the referenced package.

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
```

Arguments

- `PROJECT`

  Specifies the project file. If not specified, the command searches the current directory for one.

- `PACKAGE_NAME`

  The package reference to add.

Options

- `-f|--framework <FRAMEWORK>`

  Adds a package reference only when targeting a specific framework.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action (for example, to complete authentication). Available since .NET Core 2.1 SDK, version 2.1.400 or later.

- `-n|--no-restore`

  Adds a package reference without performing a restore preview and compatibility check.

- `--package-directory <PACKAGE_DIRECTORY>`

  The directory where to restore the packages. The default package restore location is `%userprofile%\.nuget\packages` on Windows and `~/.nuget/packages` on macOS and Linux. For more information, see Managing the global packages, cache, and temp folders in NuGet.

- `-s|--source <SOURCE>`

  The NuGet package source to use during the restore operation.

- `-v|--version <VERSION>`

  Version of the package. See NuGet package versioning.

Examples

- Add `Newtonsoft.Json` NuGet package to a project:

```
dotnet add package Newtonsoft.Json
```

- Add a specific version of a package to a project:

```
dotnet add ToDo.csproj package Microsoft.Azure.DocumentDB.Core -v 1.0.0
```

- Add a package using a specific NuGet source:

```
dotnet add package Microsoft.AspNetCore.StaticFiles -s https://dotnet.myget.org/F/dotnet-
core/api/v3/index.json
```

See also

- Managing the global packages, cache, and temp folders in NuGet
- NuGet package versioning

dotnet list package

6/27/2019 • 3 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 2.2 SDK and later versions

Name

`dotnet list package` - Lists the package references for a project or solution.

Synopsis

```
dotnet list [<PROJECT>|<SOLUTION>] package [--config] [--framework] [--highest-minor] [--highest-patch]
   [--include-prerelease] [--include-transitive] [--interactive] [--outdated] [--source]
dotnet list package [-h|--help]
```

Description

The `dotnet list package` command provides a convenient option to list all NuGet package references for a specific project or a solution. You first need to build the project in order to have the assets needed for this command to process. The following example shows the output of the `dotnet list package` command for the SentimentAnalysis project:

```
Project 'SentimentAnalysis' has the following package references
   [netcoreapp2.1]:
   Top-level Package               Requested   Resolved
   > Microsoft.ML                  0.11.0      0.11.0
   > Microsoft.NETCore.App   (A)   [2.1.0, )   2.1.0

(A) : Auto-referenced package.
```

The **Requested** column refers to the package version specified in the project file and can be a range. The **Resolved** column lists the version that the project is currently using and is always a single value. The packages displaying an `(A)` right next to their names represent implicit package references that are inferred from your project settings (`Sdk` type, `<TargetFramework>` or `<TargetFrameworks>` property, etc.)

Use the `--outdated` option to find out if there are newer versions available of the packages you're using in your projects. By default, `--outdated` lists the latest stable packages unless the resolved version is also a prerelease version. To include prerelease versions when listing newer versions, also specify the `--include-prerelease` option. The following examples shows the output of the `dotnet list package --outdated --include-prerelease` command for the same project as the previous example:

```
The following sources were used:
   https://api.nuget.org/v3/index.json

Project `SentimentAnalysis` has the following updates to its packages
   [netcoreapp2.1]:
   Top-level Package      Requested   Resolved   Latest
   > Microsoft.ML         0.11.0      0.11.0     1.0.0-preview
```

If you need to find out whether your project has transitive dependencies, use the `--include-transitive` option.

Transitive dependencies occur when you add a package to your project that in turn relies on another package. The following example shows the output from running the `dotnet list package --include-transitive` command for the HelloPlugin project, which displays top-level packages and the packages they depend on:

```
Project 'HelloPlugin' has the following package references
   [netcoreapp3.0]:
   Top-level Package                      Requested                   Resolved
   > Microsoft.NETCore.Platforms    (A)   [3.0.0-preview3.19128.7, )   3.0.0-preview3.19128.7
   > Microsoft.WindowsDesktop.App   (A)   [3.0.0-preview3-27504-2, )   3.0.0-preview3-27504-2

   Transitive Package              Resolved
   > Microsoft.NETCore.Targets     2.0.0
   > PluginBase                    1.0.0

(A) : Auto-referenced package.
```

Arguments

`PROJECT | SOLUTION`

The project or solution file to operate on. If not specified, the command searches the current directory for one. If more than one solution or project is found, an error is thrown.

Options

- `--config <SOURCE>`

  The NuGet sources to use when searching for newer packages. Requires the `--outdated` option.

- `--framework <FRAMEWORK>`

  Displays only the packages applicable for the specified target framework. To specify multiple frameworks, repeat the option multiple times. For example: `--framework netcoreapp2.2 --framework netstandard2.0`.

- `-h|--help`

  Prints out a short help for the command.

- `--highest-minor`

  Considers only the packages with a matching major version number when searching for newer packages. Requires the `--outdated` option.

- `--highest-patch`

  Considers only the packages with a matching major and minor version numbers when searching for newer packages. Requires the `--outdated` option.

- `--include-prerelease`

  Considers packages with prerelease versions when searching for newer packages. Requires the `--outdated` option.

- `--include-transitive`

  Lists transitive packages, in addition to the top-level packages. When specifying this option, you get a list of packages that the top-level packages depend on.

- `--interactive`

Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--outdated`

  Lists packages that have newer versions available.

- `-s|--source <SOURCE>`

  The NuGet sources to use when searching for newer packages. Requires the `--outdated` option.

Examples

- List package references of a specific project:

```
dotnet list SentimentAnalysis.csproj package
```

- List package references that have newer versions available, including prerelease versions:

```
dotnet list package --outdated --include-prerelease
```

- List package references for a specific target framework:

```
dotnet list package --framework netcoreapp3.0
```

dotnet remove package

5/15/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet remove package` - Removes package reference from a project file.

Synopsis

```
dotnet remove [<PROJECT>] package <PACKAGE_NAME> [-h|--help]
```

Description

The `dotnet remove package` command provides a convenient option to remove a NuGet package reference from a project.

Arguments

`PROJECT`

Specifies the project file. If not specified, the command searches the current directory for one.

`PACKAGE_NAME`

The package reference to remove.

Options

`-h|--help`

Prints out a short help for the command.

Examples

Removes `Newtonsoft.Json` NuGet package from a project in the current directory:

```
dotnet remove package Newtonsoft.Json
```

.NET Core additional tools overview

5/15/2019 • 2 minutes to read • Edit Online

This section compiles a list of tools that support and extend the .NET Core functionality, in addition to the .NET Core command-line interface (CLI) tools.

WCF Web Service Reference tool

The WCF (Windows Communication Foundation) Web Service Reference is a Visual Studio connected service provider that made its debut in Visual Studio 2017 version 15.5. This tool retrieves metadata from a web service in the current solution, on a network location, or from a WSDL file, and generates a source file compatible with .NET Core, defining a WCF proxy class with methods that you can use to access the web service operations.

WCF dotnet-svcutil tool

The WCF (Windows Communication Foundation) dotnet-svcutil tool is a .NET Core CLI tool that retrieves metadata from a web service on a network location or from a WSDL file, and generates a source file compatible with .NET Core, defining a WCF proxy class with methods that you can use to access the web service operations. The **dotnet-svcutil** tool is an alternative option to the **WCF Web Service Reference** Visual Studio connected service provider which first shipped with Visual Studio 2017 v15.5. The **dotnet-svcutil** tool as a .NET Core CLI tool, is available cross-platform on Linux, macOS, and Windows.

WCF dotnet-svcutil.xmlserializer tool

On the .NET Framework, you can pre-generate a serialization assembly using the svcutil tool. The dotnet-svcutil.xmlserializer NuGet package provides similar functionality on .NET Core. It pre-generates C# serialization code for the types in the client application that are used by the WCF Service Contract and that can be serialized by the XmlSerializer. This improves the startup performance of XML serialization when serializing or deserializing objects of those types.

XML Serializer Generator

Like the Xml Serializer Generator (sgen.exe) for the .NET Framework, the Microsoft.XmlSerializer.Generator NuGet package is the solution for .NET Core and .NET Standard libraries. It creates an XML serialization assembly for types contained in an assembly to improve the startup performance of XML serialization when serializing or de-serializing objects of those types using XmlSerializer.

Use the WCF Web Service Reference Provider Tool

4/24/2019 • 3 minutes to read • Edit Online

Over the years, many Visual Studio developers have enjoyed the productivity that the **Add Service Reference** tool provided when their .NET Framework projects needed to access web services. The **WCF Web Service Reference** tool is a Visual Studio connected service extension that provides an experience like the Add Service Reference functionality for .NET Core and ASP.NET Core projects. This tool retrieves metadata from a web service in the current solution, on a network location, or from a WSDL file, and generates a .NET Core compatible source file containing Windows Communication Foundation (WCF) client proxy code that you can use to access the web service.

> **IMPORTANT**
> You should only reference services from a trusted source. Adding references from an untrusted source may compromise security.

Prerequisites

- Visual Studio 2017 15.5 or later versions

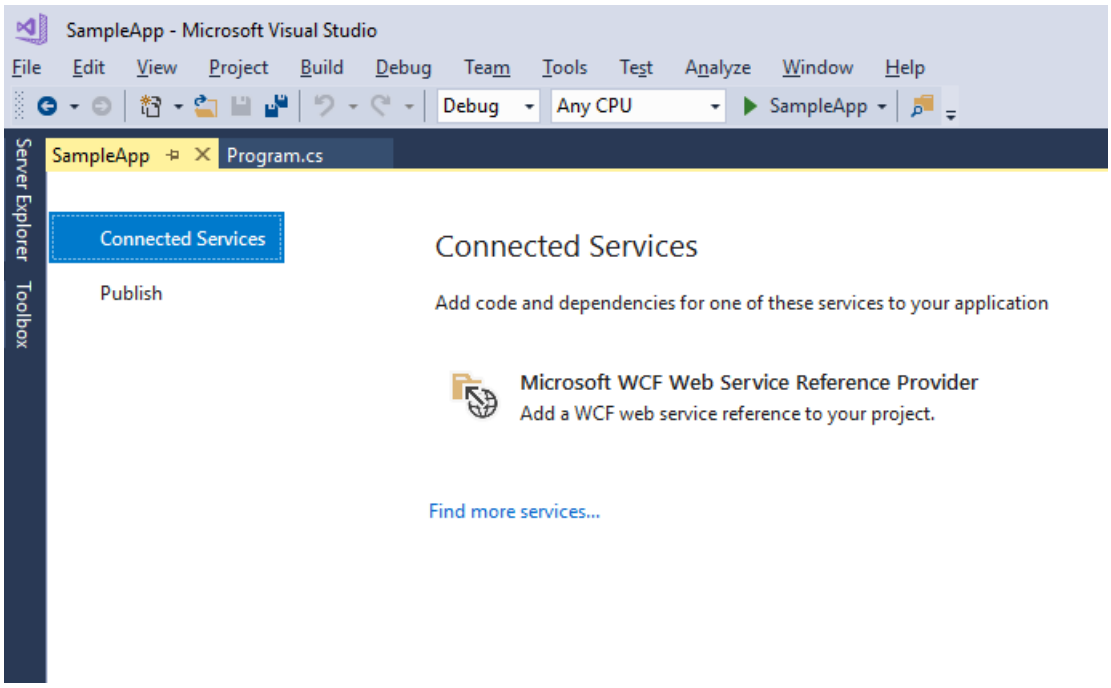
How to use the extension

> **NOTE**
> The **WCF Web Service Reference** option is applicable to projects created using the following project templates:
>
> - **Visual C#** > **.NET Core**
> - **Visual C#** > **.NET Standard**
> - **Visual C#** > **Web** > **ASP.NET Core Web Application**

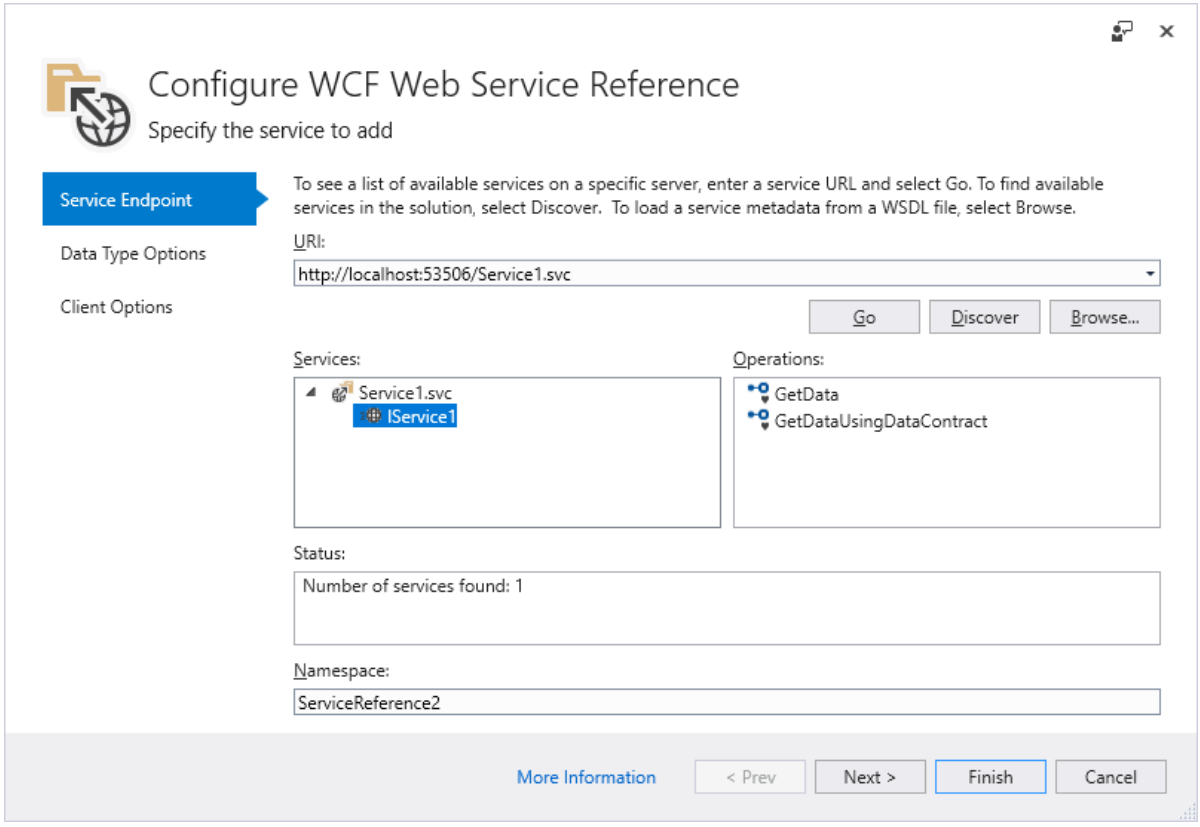
Using the **ASP.NET Core Web Application** project template as an example, this article walks you through adding a WCF service reference to the project:

1. In Solution Explorer, double-click the **Connected Services** node of the project (for a .NET Core or .NET Standard project this option is available when you right-click on the **Dependencies** node of the project in Solution Explorer).

   The **Connected Services** page appears as shown in the following image:



2. On the **Connected Services** page, click **Microsoft WCF Web Service Reference Provider**. This brings up the **Configure WCF Web Service Reference** wizard:



3. Select a service.

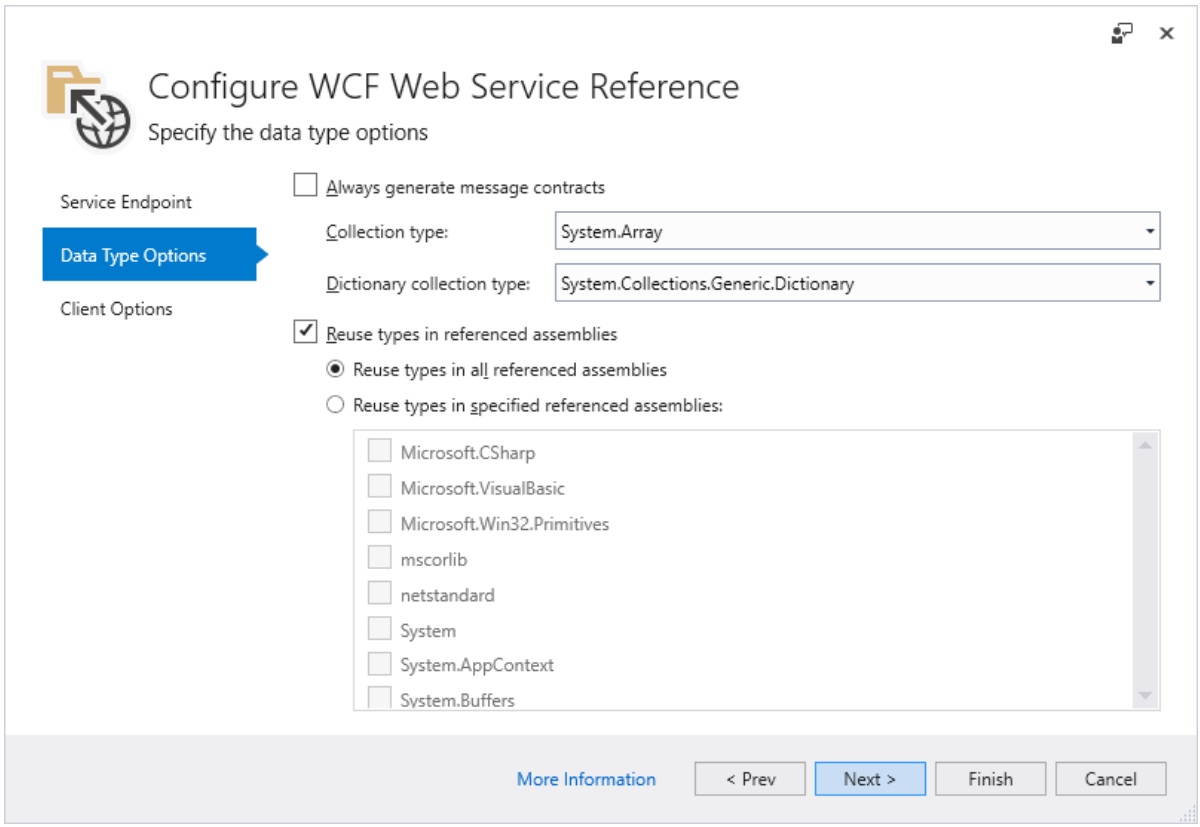
   3a. There are several services search options available within the **Configure WCF Web Service Reference** wizard:

   - To search for services defined in the current solution, click the **Discover** button.
   - To search for services hosted at a specified address, enter a service URL in the **Address** box and click the **Go** button.
   - To select a WSDL file that contains the web service metadata information, click the **Browse** button.

   3b. Select the service from the search results list in the **Services** box. If needed, enter the namespace for the generated code in the corresponding **Namespace** text box.

3c. Click the **Next** button to open the **Data Type Options** and the **Client Options** pages. Alternatively, click the **Finish** button to use the default options.

4. The **Data Type Options** form enables you to refine the generated service reference configuration settings:



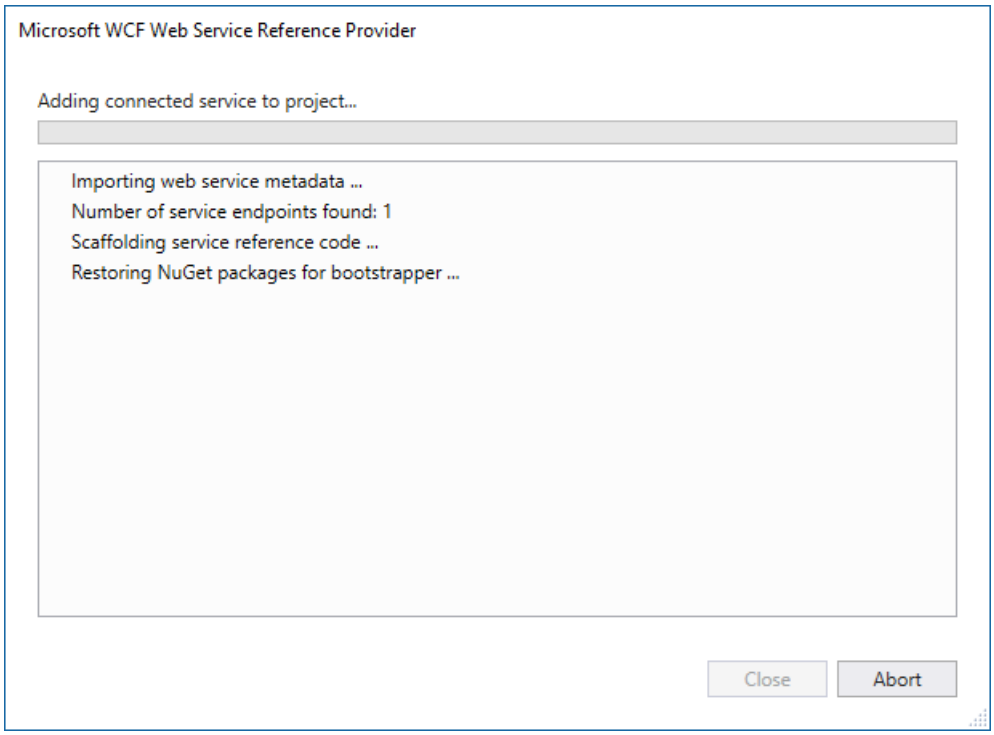
> **NOTE**
>
> The **Reuse types in referenced assemblies** check box option is useful when data types needed for service reference code generation are defined in one of your project's referenced assemblies. It's important to reuse those existing data types to avoid compile-time type clash or runtime issues.

There may be a delay while type information is loaded, depending on the number of project dependencies and other system performance factors. The **Finish** button is disabled during loading unless the **Reuse types in referenced assemblies** check box is unchecked.

5. Click **Finish** when you are done.

While displaying progress, the tool:

- Downloads metadata from the WCF service.
- Generates the service reference code in a file named *reference.cs*, and adds it to your project under the **Connected Services** node.
- Updates the project file (.csproj) with NuGet package references required to compile and run on the target platform.



When these processes complete, you can create an instance of the generated WCF client type and invoke the service operations.

Next steps

Feedback & questions

If you have any questions or feedback, open an issue on GitHub. You can also review any existing questions or issues at the WCF repo on GitHub.

Release notes

- Refer to the Release notes for updated release information, including known issues.

WCF dotnet-svcutil tool for .NET Core

5/6/2019 • 3 minutes to read • Edit Online

The Windows Communication Foundation (WCF) **dotnet-svcutil** tool is a .NET Core CLI tool that retrieves metadata from a web service on a network location or from a WSDL file, and generates a WCF class containing client proxy methods that access the web service operations.

Similar to the **Service Model Metadata - svcutil** tool for .NET Framework projects, the **dotnet-svcutil** is a command-line tool for generating a web service reference compatible with .NET Core and .NET Standard projects.

The **dotnet-svcutil** tool is an alternative option to the **WCF Web Service Reference** Visual Studio connected service provider that first shipped with Visual Studio 2017 v15.5. The **dotnet-svcutil** tool as a .NET Core CLI tool, is available cross-platform on Linux, macOS, and Windows.

> **IMPORTANT**
> You should only reference services from a trusted source. Adding references from an untrusted source may compromise security.

Prerequisites

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

- .NET Core 2.1 SDK or later versions
- Your favorite code editor

Getting started

The following example walks you through the steps required to add a web service reference to a .NET Core web project and invoke the service. You'll create a .NET Core web application named *HelloSvcutil* and add a reference to a web service that implements the following contract:

```
[ServiceContract]
public interface ISayHello
{
    [OperationContract]
    string Hello(string name);
}
```

For this example, let's assume the web service will be hosted at the following address:
`http://contoso.com/SayHello.svc`

From a Windows, macOS, or Linux command window perform the following steps:

1. Create a directory named *HelloSvcutil* for your project and make it your current directory, as in the following example:

   ```
   mkdir HelloSvcutil
   cd HelloSvcutil
   ```

2. Create a new C# web project in that directory using the `dotnet new` command as follows:

```
dotnet new web
```

3. Install the `dotnet-svcutil` NuGet package as a CLI tool:

   - dotnet-svcutil 2.x
   - dotnet-svcutil 1.x

```
dotnet tool install --global dotnet-svcutil
```

4. Run the *dotnet-svcutil* command to generate the web service reference file as follows:

   - dotnet-svcutil 2.x
   - dotnet-svcutil 1.x

```
dotnet-svcutil http://contoso.com/SayHello.svc
```

The generated file is saved as *HelloSvcutil/ServiceReference/Reference.cs*. The *dotnet-svcutil* tool also adds to the project the appropriate WCF packages required by the proxy code as package references.

Using the Service Reference

1. Restore the WCF packages using the `dotnet restore` command as follows:

```
dotnet restore
```

2. Find the name of the client class and operation you want to use. `Reference.cs` will contain a class that inherits from `System.ServiceModel.ClientBase`, with methods that can be used to call operations on the service. In this example, you want to call the *SayHello* service's *Hello* operation. `ServiceReference.SayHelloClient` is the name of the client class, and has a method called `HelloAsync` that can be used to call the operation.

3. Open the `Startup.cs` file in your editor, and add a using statement for the service reference namespace at the top:

```
using ServiceReference;
```

4. Edit the `Configure` method to invoke the web service. You do this by creating an instance of the class that inherits from `ClientBase` and calling the method on the client object:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        var client = new SayHelloClient();
        var response = await client.HelloAsync();
        await context.Response.WriteAsync(response);
    });
}
```

5. Run the application using the `dotnet run` command as follows:

```
dotnet run
```

6. Navigate to the URL listed in the console (for example, `http://localhost:5000`) in your web browser.

You should see the following output: "Hello dotnet-svcutil!"

For a detailed description of the `dotnet-svcutil` tool parameters, invoke the tool passing the help parameter as follows:

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

```
dotnet-svcutil --help
```

Feedback & questions

If you have any questions or feedback, open an issue on GitHub. You can also review any existing questions or issues at the WCF repo on GitHub.

Release notes

- Refer to the Release notes for updated release information, including known issues.

Information

- dotnet-svcutil NuGet Package

Using dotnet-svcutil.xmlserializer on .NET Core

5/15/2019 • 2 minutes to read • Edit Online

The `dotnet-svcutil.xmlserializer` NuGet package can pre-generate a serialization assembly for .NET Core projects. It pre-generates C# serialization code for the types in the client application that are used by the WCF Service Contract and that can be serialized by the XmlSerializer. This improves the startup performance of XML serialization when serializing or deserializing objects of those types.

Prerequisites

- .NET Core 2.1 SDK or later
- Your favorite code editor

You can use the command `dotnet --info` to check which versions of .NET Core SDK and runtime you already have installed.

Getting started

To use `dotnet-svcutil.xmlserializer` in a .NET Core console application:

1. Create a WCF Service named 'MyWCFService' using the default template 'WCF Service Application' in .NET Framework. Add `[XmlSerializerFormat]` attribute on the service method like the following:

   ```
   [ServiceContract]
   public interface IService1
   {
       [XmlSerializerFormat]
       [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
   "http://tempuri.org/IService1/GetDataResponse")]
       string GetData(int value);
   }
   ```

2. Create a .NET Core console application as WCF client application that targets at .NET Core 2.1 or later versions. For example, create an app named 'MyWCFClient' with the following command:

   ```
   dotnet new console --name MyWCFClient
   ```

   To ensure your project is targeting .NET Core 2.1 or later, inspect the `TargetFramework` XML element in your project file:

   ```
   <TargetFramework>netcoreapp2.1</TargetFramework>
   ```

3. Add a package reference to `System.ServiceModel.Http` by running the following command:

   ```
   dotnet add package System.ServiceModel.Http
   ```

4. Add the WCF Client code:

```
using System.ServiceModel;

    class Program
    {
        static void Main(string[] args)
        {
            var myBinding = new BasicHttpBinding();
            var myEndpoint = new EndpointAddress("http://localhost:2561/Service1.svc"); //Fill your
service url here
            var myChannelFactory = new ChannelFactory<IService1>(myBinding, myEndpoint);
            IService1 client = myChannelFactory.CreateChannel();
            string s = client.GetData(1);
            ((ICommunicationObject)client).Close();
        }
    }

[ServiceContract]
public interface IService1
{
    [XmlSerializerFormat]
    [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
"http://tempuri.org/IService1/GetDataResponse")]
    string GetData(int value);
}
```

5. Add a reference to the `dotnet-svcutil.xmlserializer` package by running the following command:

   ```
   dotnet add package dotnet-svcutil.xmlserializer
   ```

   Running the command should add an entry to your project file similar to this:

   ```
   <ItemGroup>
     <DotNetCliToolReference Include="dotnet-svcutil.xmlserializer" Version="1.0.0" />
   </ItemGroup>
   ```

6. Build the application by running `dotnet build`. If everything succeeds, an assembly named *MyWCFClient.XmlSerializers.dll* is generated in the output folder. If the tool failed to generate the assembly, you'll see warnings in the build output.

7. Start the WCF service by, for example, running `http://localhost:2561/Service1.svc` in the browser. Then start the client application, and it will automatically load and use the pre-generated serializers at runtime.

Using Microsoft XML Serializer Generator on .NET Core

5/15/2019 • 2 minutes to read • Edit Online

This tutorial teaches you how to use the Microsoft XML Serializer Generator in a C# .NET Core application. During the course of this tutorial, you learn:

- How to create a .NET Core app
- How to add a reference to the Microsoft.XmlSerializer.Generator package
- How to edit your MyApp.csproj to add dependencies
- How to add a class and an XmlSerializer
- How to build and run the application

Like the Xml Serializer Generator (sgen.exe) for the .NET Framework, the Microsoft.XmlSerializer.Generator NuGet package is the equivalent for .NET Core and .NET Standard projects. It creates an XML serialization assembly for types contained in an assembly to improve the startup performance of XML serialization when serializing or de-serializing objects of those types using XmlSerializer.

Prerequisites

To complete this tutorial:

- .NET Core 2.1 SDK or later
- Your favorite code editor.

> **TIP**
> Need to install a code editor? Try Visual Studio!

Use Microsoft XML Serializer Generator in a .NET Core console application

The following instructions show you how to use XML Serializer Generator in a .NET Core console application.

Create a .NET Core console application

Open a command prompt and create a folder named *MyApp*. Navigate to the folder you created and type the following command:

```
dotnet new console
```

Add a reference to the Microsoft.XmlSerializer.Generator package in the MyApp project

Use the `dotnet add package` command to add the reference in your project.

Type:

```
dotnet add package Microsoft.XmlSerializer.Generator -v 1.0.0
```

Verify changes to MyApp.csproj after adding the package

Open your code editor and let's get started! We're still working from the *MyApp* directory we built the app in.

Open *MyApp.csproj* in your text editor.

After running the `dotnet add package` command, the following lines are added to your *MyApp.csproj* project file:

```
<ItemGroup>
   <PackageReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

Add another ItemGroup section for .NET Core CLI Tool support

Add the following lines after the `ItemGroup` section that we inspected:

```
<ItemGroup>
   <DotNetCliToolReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

Add a class in the application

Open *Program.cs* in your text editor. Add the class named *MyClass* in *Program.cs*.

```
public class MyClass
{
   public int Value;
}
```

Create an `XmlSerializer` for MyClass

Add the following line inside *Main* to create an `XmlSerializer` for MyClass:

```
var serializer = new System.Xml.Serialization.XmlSerializer(typeof(MyClass));
```

Build and run the application

Still within the *MyApp* folder, run the application via `dotnet run` and it automatically loads and uses the pre-generated serializers at runtime.

Type the following command in your console window:

```
dotnet run
```

> **NOTE**
>
> `dotnet run` calls `dotnet build` to ensure that the build targets have been built, and then calls `dotnet <assembly.dll>` to run the target application.

> **IMPORTANT**
>
> The commands and steps shown in this tutorial to run your application are used during development time only. Once you're ready to deploy your app, take a look at the different deployment strategies for .NET Core apps and the `dotnet publish` command.

If everything succeeds, an assembly named *MyApp.XmlSerializers.dll* is generated in the output folder.

Congratulations! You have just:

- Created a .NET Core app.
- Added a reference to the Microsoft.XmlSerializer.Generator package.
- Edited your MyApp.csproj to add dependencies.
- Added a class and an XmlSerializer.
- Built and ran the application.

Related resources

- Introducing XML Serialization
- How to: Serialize Using XmlSerializer (C#)
- How to: Serialize Using XmlSerializer (Visual Basic)

Port your code from .NET Framework to .NET Core

7/9/2019 • 2 minutes to read • Edit Online

If you've got code that runs on the .NET Framework, you may be interested in running your code on .NET Core, too. Here's an overview of the porting process and a list of the tools you may find helpful when porting your code to .NET Core.

Overview of the porting process

This is the process we recommend you take when porting your project to .NET Core. Each step of the process is covered in more detail in further articles.

1. Identify and account for your third-party dependencies.

   This step involves understanding what your third-party dependencies are, how you depend on them, how to check if they also run on .NET Core, and steps you can take if they don't. It also covers how you can migrate your dependencies over to the PackageReference format that is used in .NET Core.

2. Retarget all projects you wish to port to target the .NET Framework 4.7.2 or higher.

   This step ensures that you can use API alternatives for .NET Framework-specific targets when .NET Core doesn't support a particular API.

3. Use the .NET Portability Analyzer to analyze your assemblies and develop a plan to port based on its results.

   The API Portability Analyzer tool analyzes your compiled assemblies and generates a report that shows a high-level portability summary and a breakdown of each API you're using that isn't available on .NET Core. You can use this report alongside an analysis of your codebase to develop a plan for how you'll port your code over.

4. Port your tests code.

   Because porting to .NET Core is such a significant change to your codebase, it's highly recommended to get your tests ported, so that you can run tests as you port your code over. MSTest, xUnit, and NUnit all support .NET Core.

5. Execute your plan for porting!

The following list shows tools you might find helpful to use during the porting process:

- .NET Portability Analyzer - command line tool or Visual Studio Extension, a tool that can generate a report of how portable your code is between .NET Framework and your target .NET Core platform. The report contains an assembly-by-assembly breakdown of the Type and APIs missing on the target .NET Core platform. For more information, see .NET Portability Analyzer. It is recommended to run the .NET Portability Analyzer tool before you start porting, as it will help you identify any gaps in missing APIs.
- .NET API Analyzer - A Roslyn analyzer that discovers .NET Standard API that throws PlatformNotSupportedException on some platforms, detects calls to deprecated APIs, and discovers some other potential compatibility risks for C# APIs on different platforms. For more information, see .NET API analyzer. This analyzer is helpful after you already created your .NET Core project to identify runtime behavior differences on different platforms.
- Reverse Package Search - A useful web service that allows you to search for a type and find packages containing that type.

Additionally, you can attempt to port smaller solutions or individual projects to the .NET Core project file format with the CsprojToVs2017 tool.

> **WARNING**
>
> CsprojToVs2017 is a third-party tool. There is no guarantee that it will work for all of your projects, and it may cause subtle changes in behavior that you depend on. CsprojToVs2017 should be used as a *starting point* that automates the basic things that can be automated. It is not a guaranteed solution to migrating project file formats.

NEXT

.NET Framework technologies unavailable on .NET Core

8/20/2019 • 2 minutes to read • Edit Online

Several technologies available to .NET Framework libraries aren't available for use with .NET Core, such as AppDomains, Remoting, Code Access Security (CAS), and Security Transparency. If your libraries rely on one or more of these technologies, consider the alternative approaches outlined below. For more information on API compatibility, the CoreFX team maintains a List of behavioral changes/compat breaks and deprecated/legacy APIs at GitHub.

Just because an API or technology isn't currently implemented doesn't imply it's intentionally unsupported. You should first search the GitHub repositories for .NET Core to see if a particular issue you encounter is by design, but if you cannot find such an indicator, please file an issue in the dotnet/corefx repository issues at GitHub to ask for specific APIs and technologies. Porting requests in the issues are marked with the `port-to-core` label.

AppDomains

Application domains (AppDomains) isolate apps from one another. AppDomains require runtime support and are generally quite expensive. Creating additional app domains is not supported. We don't plan on adding this capability in future. For code isolation, we recommend separate processes or using containers as an alternative. For the dynamic loading of assemblies, we recommend the new AssemblyLoadContext class.

To make code migration from .NET Framework easier, .NET Core exposes some of the AppDomain API surface. Some of the APIs function normally (for example, AppDomain.UnhandledException), some members do nothing (for example, SetCachePath), and some of them throw PlatformNotSupportedException (for example, CreateDomain). Check the types you use against the `System.AppDomain` reference source in the dotnet/corefx GitHub repository, making sure to select the branch that matches your implemented version.

Remoting

.NET Remoting was identified as a problematic architecture. It's used for cross-AppDomain communication, which is no longer supported. Also, Remoting requires runtime support, which is expensive to maintain. For these reasons, .NET Remoting isn't supported on .NET Core, and we don't plan on adding support for it in the future.

For communication across processes, consider inter-process communication (IPC) mechanisms as an alternative to Remoting, such as the System.IO.Pipes or the MemoryMappedFile class.

Across machines, use a network-based solution as an alternative. Preferably, use a low-overhead plain text protocol, such as HTTP. The Kestrel web server, the web server used by ASP.NET Core, is an option here. Also consider using System.Net.Sockets for network-based, cross-machine scenarios. For more options, see .NET Open Source Developer Projects: Messaging.

Code Access Security (CAS)

Sandboxing, which relies on the runtime or the framework to constrain which resources a managed application or library uses or runs, isn't supported on .NET Framework and therefore is also not supported on .NET Core. There are too many cases in the .NET Framework and the runtime where an elevation of privileges occurs to continue treating CAS as a security boundary. In addition, CAS makes the implementation more complicated and often has correctness-performance implications for applications that don't intend to use it.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts for running processes with the minimum set of privileges.

Security Transparency

Similar to CAS, Security Transparency separates sandboxed code from security critical code in a declarative fashion but is no longer supported as a security boundary. This feature is heavily used by Silverlight.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts for running processes with the least set of privileges.

System.EnterpriseServices

System.EnterpriseServices (COM+) is not supported by .NET Core.

NEXT

Analyze your dependencies to port code to .NET Core

2/8/2019 • 5 minutes to read • Edit Online

To port your code to .NET Core or .NET Standard, you must understand your dependencies. External dependencies are the NuGet packages or DLLs you reference in your project, but that you don't build. Evaluate each dependency and develop a contingency plan for the ones that aren't compatible with .NET Core. Here's how to determine if a dependency is compatible with .NET Core.

Analyze referenced NuGet packages in your projects

If you reference NuGet packages in your project, you need to verify if they're compatible with .NET Core. There are two ways to accomplish that:

- Using the NuGet Package Explorer app
- Using the nuget.org site

After analyzing the packages, if they're not compatible with .NET Core and only target .NET Framework, you can check if the .NET Framework compatibility mode can help with your porting process.

Analyze NuGet packages using NuGet Package Explorer

A NuGet package is itself a set of folders that contain platform-specific assemblies. So you need to check if there's a folder that contains a compatible assembly inside the package.

The easiest way to inspect NuGet Package folders is to use the NuGet Package Explorer tool. After installing it, use the following steps to see the folder names:

1. Open the NuGet Package Explorer.
2. Click **Open package from online feed**.
3. Search for the name of the package.
4. Select the package name from the search results and click **open**.
5. Expand the *lib* folder on the right-hand side and look at folder names.

Look for a folder with any of the following names:

```
netstandard1.0
netstandard1.1
netstandard1.2
netstandard1.3
netstandard1.4
netstandard1.5
netstandard1.6
netstandard2.0
netcoreapp1.0
netcoreapp1.1
netcoreapp2.0
netcoreapp2.1
netcoreapp2.2
portable-net45-win8
portable-win8-wpa8
portable-net451-win81
portable-net45-win8-wpa8-wpa81
```

These values are the Target Framework Monikers (TFMs) that map to versions of the .NET Standard, .NET Core, and traditional Portable Class Library (PCL) profiles that are compatible with .NET Core.

> **IMPORTANT**
>
> When looking at the TFMs that a package supports, note that `netcoreapp*`, while compatible, is for .NET Core projects only and not for .NET Standard projects. A library that only targets `netcoreapp*` and not `netstandard*` can only be consumed by other .NET Core apps.

Analyze NuGet packages using nuget.org

Alternatively, you can see the TFMs that each package supports on nuget.org under the **Dependencies** section of the package page.

Although using the site is an easier method to verify the compatibility, **Dependencies** information is not available on the site for all packages.

.NET Framework compatibility mode

After analyzing the NuGet packages, you might find that they only target the .NET Framework, as most NuGet packages do.

Starting with .NET Standard 2.0, the .NET Framework compatibility mode was introduced. This compatibility mode allows .NET Standard and .NET Core projects to reference .NET Framework libraries. Referencing .NET Framework libraries doesn't work for all projects, such as if the library uses Windows Presentation Foundation (WPF) APIs, but it does unblock many porting scenarios.

When you reference NuGet packages that target the .NET Framework in your project, such as Huitian.PowerCollections, you get a package fallback warning (NU1701) similar to the following example:

```
NU1701: Package 'Huitian.PowerCollections 1.0.0' was restored using '.NETFramework,Version=v4.6.1' instead of the project target framework '.NETStandard,Version=v2.0'. This package may not be fully compatible with your project.
```

That warning is displayed when you add the package and every time you build to make sure you test that package with your project. If your project is working as expected, you can suppress that warning by editing the package properties in Visual Studio or by manually editing the project file in your favorite code editor.

To suppress the warning by editing the project file, find the `PackageReference` entry for the package you want to suppress the warning for and add the `NoWarn` attribute. The `NoWarn` attribute accepts a comma-separated list of all the warning IDs. The following example shows how to suppress the `NU1701` warning for the `Huitian.PowerCollections` package by editing your project file manually:

```
<ItemGroup>
  <PackageReference Include="Huitian.PowerCollections" Version="1.0.0" NoWarn="NU1701" />
</ItemGroup>
```

For more information on how to suppress compiler warnings in Visual Studio, see Suppressing warnings for NuGet packages.

Port your packages to `PackageReference`

.NET Core uses PackageReference to specify package dependencies. If you are using packages.config to specify your packages, you will need to convert over to `PackageReference`.

You can learn more at Migrate from packages.config to PackageReference.

What to do when your NuGet package dependency doesn't run on .NET Core

There are a few things you can do if a NuGet package you depend on doesn't run on .NET Core:

1. If the project is open source and hosted somewhere like GitHub, you can engage the developers directly.
2. You can contact the author directly on nuget.org. Search for the package and click **Contact Owners** on the left-hand side of the package's page.
3. You can search for another package that runs on .NET Core that accomplishes the same task as the package you were using.
4. You can attempt to write the code the package was doing yourself.
5. You could eliminate the dependency on the package by changing the functionality of your app, at least until a compatible version of the package becomes available.

Remember that open-source project maintainers and NuGet package publishers are often volunteers. They contribute because they care about a given domain, do it for free, and often have a different daytime job. So be mindful of that when contacting them to ask for .NET Core support.

If you can't resolve your issue with any of the above, you may have to port to .NET Core at a later date.

The .NET Team would like to know which libraries are the most important to support with .NET Core. You can send an email to dotnet@microsoft.com about the libraries you'd like to use.

Analyze dependencies that aren't NuGet packages

You may have a dependency that isn't a NuGet package, such as a DLL in the file system. The only way to determine the portability of that dependency is to run the .NET Portability Analyzer tool. The tool can analyze assemblies that target the .NET Framework and identify APIs that aren't portable to other .NET platforms such as .NET Core. You can run the tool as a console application or as a Visual Studio extension.

NEXT

Port .NET Framework libraries to .NET Core

8/20/2019 • 6 minutes to read • Edit Online

Learn how to port .NET Framework library code to .NET Core, to run cross-platform and expand the reach of the apps that use it.

Prerequisites

This article assumes that you:

- Are using Visual Studio 2017 or later.
  - .NET Core isn't supported on earlier versions of Visual Studio
- Understand the recommended porting process.
- Have resolved any issues with third-party dependencies.

You should also become familiar with the content of the following topics:

.NET Standard
This topic describes the formal specification of .NET APIs that are intended to be available on all .NET implementations.

Packages, Metapackages and Frameworks
This article discusses how .NET Core defines and uses packages and how packages support code running on multiple .NET implementations.

Developing Libraries with Cross Platform Tools
This topic explains how to write libraries for .NET using cross-platform CLI tools.

Additions to the *csproj* format for .NET Core
This article outlines the changes that were added to the project file as part of the move to *csproj* and MSBuild.

Porting to .NET Core - Analyzing your Third-Party Party Dependencies
This topic discusses the portability of third-party dependencies and what to do when a NuGet package dependency doesn't run on .NET Core.

Retargeting your .NET Framework code to .NET Framework 4.7.2

If your code isn't targeting .NET Framework 4.7.2, we recommended that you retarget to .NET Framework 4.7.2. This ensures the availability of the latest API alternatives for cases where the .NET Standard doesn't support existing APIs.

For each of your projects in Visual Studio you wish to port, do the following:

1. Right-click on the project and select **Properties**.
2. In the **Target Framework** dropdown, select **.NET Framework 4.7.2**.
3. Recompile your projects.

Because your projects now target .NET Framework 4.7.2, use that version of the .NET Framework as your base for porting code.

Determining the portability of your code

The next step is to run the API Portability Analyzer (ApiPort) to generate a portability report for analysis.

Make sure you understand the API Portability Analyzer (ApiPort) and how to generate portability reports for targeting .NET Core. How you do this likely varies based on your needs and personal tastes. What follows are a few different approaches. You may find yourself mixing steps of these approaches depending on how your code is structured.

Dealing primarily with the compiler

This approach may be the best for small projects or projects which don't use many .NET Framework APIs. The approach is simple:

1. Optionally, run ApiPort on your project. If you run ApiPort, gain knowledge from the report on issues you'll need to address.
2. Copy all of your code over into a new .NET Core project.
3. While referring to the portability report (if generated), solve compiler errors until the project fully compiles.

Although this approach is unstructured, the code-focused approach often leads to resolving issues quickly and might be the best approach for smaller projects or libraries. A project that contains only data models might be an ideal candidate for this approach.

Staying on the .NET Framework until portability issues are resolved

This approach might be the best if you prefer to have code that compiles during the entire process. The approach is as follows:

1. Run ApiPort on a project.
2. Address issues by using different APIs that are portable.
3. Take note of any areas where you're prevented from using a direct alternative.
4. Repeat the prior steps for all projects you're porting until you're confident each is ready to be copied over into a new .NET Core project.
5. Copy the code into a new .NET Core project.
6. Work out any issues where you noted that a direct alternative doesn't exist.

This careful approach is more structured than simply working out compiler errors, but it's still relatively code-focused and has the benefit of always having code that compiles. The way you resolve certain issues that couldn't be addressed by just using another API varies greatly. You may find that you need to develop a more comprehensive plan for certain projects, which is covered as the next approach.

Developing a comprehensive plan of attack

This approach might be best for larger and more complex projects, where restructuring code or completely rewriting certain areas of code might be necessary to support .NET Core. The approach is as follows:

1. Run ApiPort on a project.
2. Understand where each non-portable type is used and how that affects overall portability.
   - Understand the nature of those types. Are they small in number but used frequently? Are they large in number but used infrequently? Is their use concentrated, or is it spread throughout your code?
   - Is it easy to isolate code that isn't portable so that you can deal with it more effectively?
   - Do you need to refactor your code?
   - For those types which aren't portable, are there alternative APIs that accomplish the same task? For example if you're using the WebClient class, you might be able to use the HttpClient class instead.
   - Are there different portable APIs available to accomplish a task, even if it's not a drop-in replacement? For example if you're using XmlSchema to parse XML but don't require XML schema discovery, you could use System.Xml.Linq APIs and implement parsing yourself as opposed to relying on an API.
3. If you have assemblies that are difficult to port, is it worth leaving them on .NET Framework for now? Here are some things to consider:
   - You may have some functionality in your library that's incompatible with .NET Core because it relies too

heavily on .NET Framework or Windows-specific functionality. Is it worth leaving that functionality behind for now and releasing a .NET Core version of your library with less features on a temporary basis until resources are available to port the features?
    - Would a refactor help?
4. Is it reasonable to write your own implementation of an unavailable .NET Framework API? You could consider copying, modifying, and using code from the .NET Framework Reference Source. The reference source code is licensed under the MIT License, so you have significant freedom to use the source as a basis for your own code. Just be sure to properly attribute Microsoft in your code.
5. Repeat this process as needed for different projects.

The analysis phase could take some time depending on the size of your codebase. Spending time in this phase to thoroughly understand the scope of changes needed and to develop a plan usually saves you time in the long run, particularly if you have a complex codebase.

Your plan could involve making significant changes to your codebase while still targeting .NET Framework 4.7.2, making this a more structured version of the previous approach. How you go about executing your plan is dependent on your codebase.

Mixing approaches

It's likely that you'll mix the above approaches on a per-project basis. You should do what makes the most sense to you and for your codebase.

Porting your tests

The best way to make sure everything works when you've ported your code is to test your code as you port it to .NET Core. To do this, you'll need to use a testing framework that builds and runs tests for .NET Core. Currently, you have three options:

- xUnit
    - Getting Started
    - Tool to convert an MSTest project to xUnit
- NUnit
    - Getting Started
    - Blog post about migrating from MSTest to NUnit
- MSTest

Recommended approach to porting

Ultimately, the porting effort depends heavily on how your .NET Framework code is structured. A good way to port your code is to begin with the *base* of your library, which are the foundational components of your code. This might be data models or some other foundational classes and methods that everything else uses directly or indirectly.

1. Port the test project that tests the layer of your library that you're currently porting.
2. Copy over the base of your library into a new .NET Core project and select the version of the .NET Standard you wish to support.
3. Make any changes needed to get the code to compile. Much of this may require adding NuGet package dependencies to your *csproj* file.
4. Run the tests and make any needed adjustments.
5. Pick the next layer of code to port over and repeat the prior steps.

If you start with the base of your library and move outward from the base and test each layer as needed, porting is a systematic process where problems are isolated to one layer of code at a time.

NEXT

Organize your project to support both .NET Framework and .NET Core

3/19/2019 • 2 minutes to read • Edit Online

Learn how to create a solution that compiles for both .NET Framework and .NET Core side-by-side. See several options to organize projects to help you achieve this goal. Here are some typical scenarios to consider when you're deciding how to setup your project layout with .NET Core. The list may not cover everything you want; prioritize based on your project's needs.

- **Combine existing projects and .NET Core projects into single projects**

  What this is good for:

  - Simplifying your build process by compiling a single project rather than compiling multiple projects, each targeting a different .NET Framework version or platform.
  - Simplifying source file management for multi-targeted projects because you must manage a single project file. When adding/removing source files, the alternatives require you to manually sync these with your other projects.
  - Easily generating a NuGet package for consumption.
  - Allows you to write code for a specific .NET Framework version in your libraries through the use of compiler directives.

  Unsupported scenarios:

  - Requires developers to use Visual Studio 2017 to open existing projects. To support older versions of Visual Studio, keeping your project files in different folders is a better option.

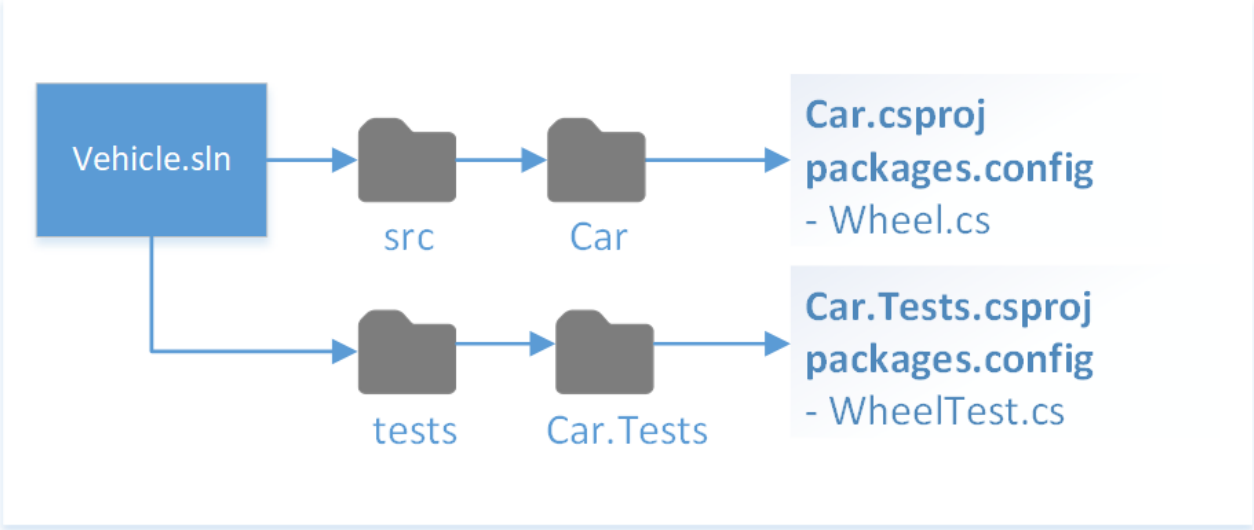
- **Keep existing projects and new .NET Core projects separate**

  What this is good for:

  - Continuing to support development on existing projects without having to upgrade for developers/contributors who may not have Visual Studio 2017.
  - Decreasing the possibility of creating new bugs in existing projects because no code churn is required in those projects.

Example

Consider the repository below:

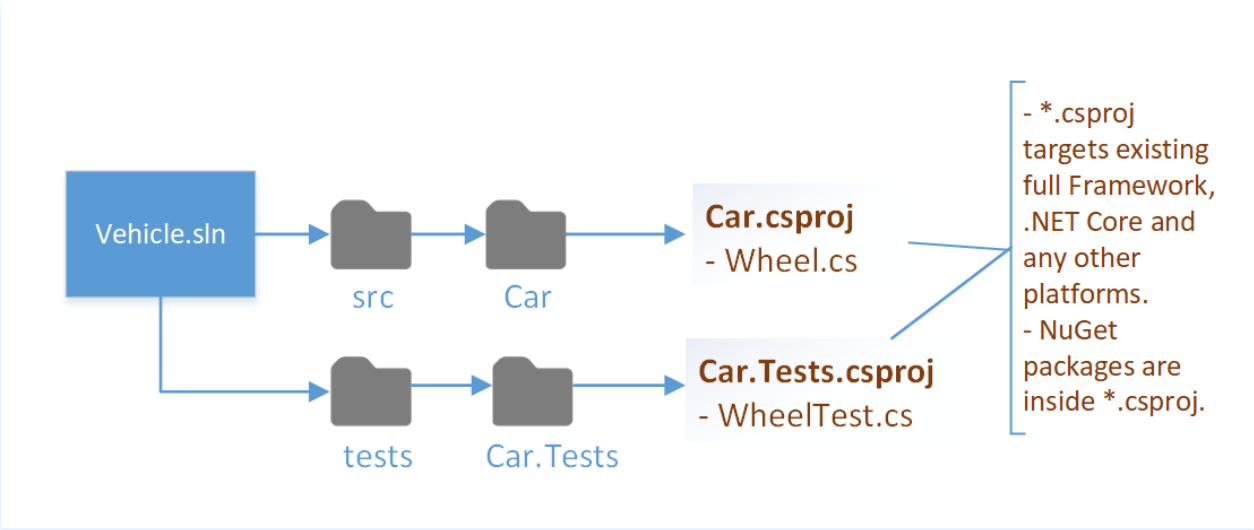


Source Code

The following describes several ways to add support for .NET Core for this repository depending on the constraints and complexity of the existing projects.

Replace existing projects with a multi-targeted .NET Core project

Reorganize the repository so that any existing **.csproj* files are removed and a single **.csproj* file is created that targets multiple frameworks. This is a great option because a single project is able to compile for different frameworks. It also has the power to handle different compilation options and dependencies per targeted framework.



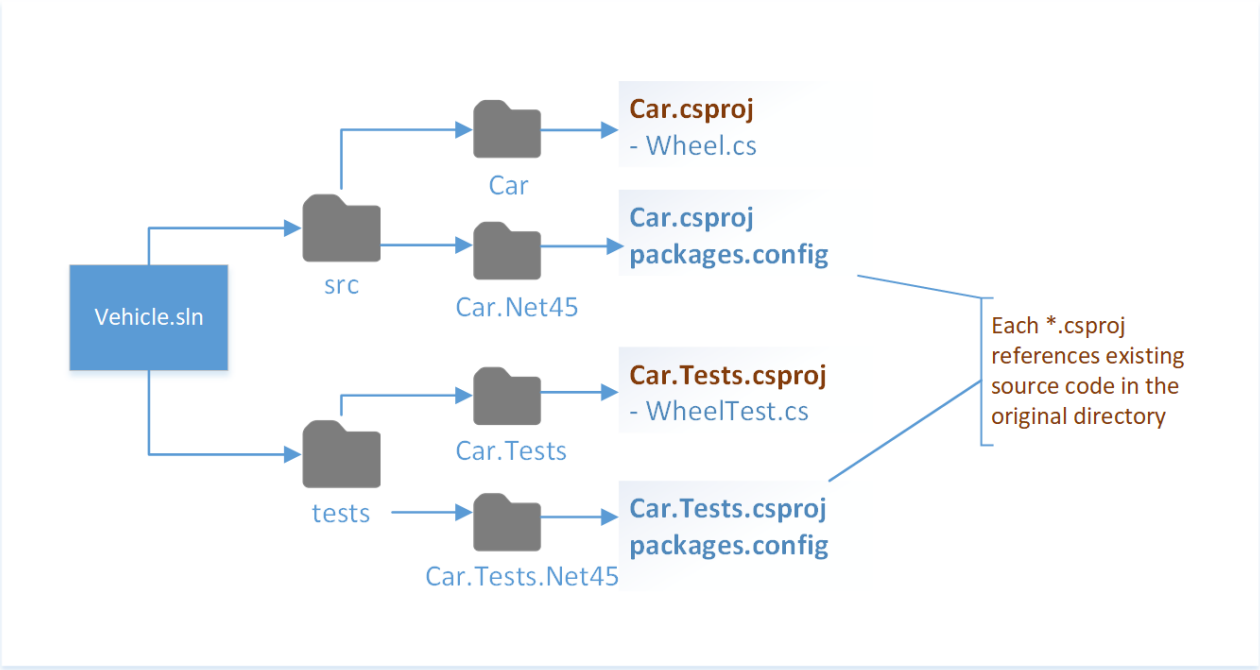
Source Code

Changes to note are:

- Replacement of *packages.config* and **.csproj* with a new .NET Core **.csproj*. NuGet packages are specified with `<PackageReference> ItemGroup`.

Keep existing projects and create a .NET Core project

If there are existing projects that target older frameworks, you may want to leave these projects untouched and use a .NET Core project to target future frameworks.



Source Code

Changes to note are:

- The .NET Core and existing projects are kept in separate folders.
  - Keeping projects in separate folders avoids forcing you to have Visual Studio 2017. You can create a separate solution that only opens the old projects.

See also

Please see the .NET Core porting documentation for more guidance on migrating to .NET Core.

Tools to help with porting to .NET Core

5/15/2019 • 2 minutes to read • Edit Online

You may find the tools listed in this article helpful when porting:

- .NET Portability Analyzer, a toolchain that can generate a report of how portable your code is between .NET Framework and .NET Core: As a command line tool As a Visual Studio Extension
- .NET API analyzer - A Roslyn analyzer that discovers potential compatibility risks for C# APIs on different platforms and detects calls to deprecated APIs.

Additionally, you can attempt to port smaller solutions or individual projects to the .NET Core project file format with the CsprojToVs2017 tool.

> **WARNING**
>
> CsprojToVs2017 is a third-party tool. There is no guarantee that it will work for all of your projects, and it may cause subtle changes in behavior that you depend on. CsprojToVs2017 should be used as a *starting point* that automates the basic things that can be automated. It is not a guaranteed solution to migrating project file formats.

Use the Windows Compatibility Pack to port code to .NET Core

3/9/2019 • 2 minutes to read • Edit Online

Some of the most common issues found when porting existing code to .NET Core are dependencies on APIs and technologies that are only found in the .NET Framework. The *Windows Compatibility Pack* provides many of these technologies, so it's much easier to build .NET Core applications and .NET Standard libraries.

This package is a logical extension of .NET Standard 2.0 that significantly increases API set and existing code compiles with almost no modifications. But in order to keep the promise of .NET Standard ("it is the set of APIs that all .NET implementations provide"), this didn't include technologies that can't work across all platforms, such as registry, Windows Management Instrumentation (WMI), or reflection emit APIs.

The *Windows Compatibility Pack* sits on top of .NET Standard and provides access to technologies that are Windows only. It's especially useful for customers that want to move to .NET Core but plan to stay on Windows as a first step. In that scenario, not being able to use Windows-only technologies is only a migration hurdle with zero architectural benefits.

Package contents

The *Windows Compatibility Pack* is provided via the NuGet Package Microsoft.Windows.Compatibility and can be referenced from projects targeting .NET Core or .NET Standard.

It provides about 20,000 APIs, including Windows-only as well as cross-platform APIs from the following technology areas:

- Code Pages
- CodeDom
- Configuration
- Directory Services
- Drawing
- ODBC
- Permissions
- Ports
- Windows Access Control Lists (ACL)
- Windows Communication Foundation (WCF)
- Windows Cryptography
- Windows EventLog
- Windows Management Instrumentation (WMI)
- Windows Performance Counters
- Windows Registry
- Windows Runtime Caching
- Windows Services

For more information, see the spec of the compatibility pack.

Get started

1. Before porting, make sure to take a look at the Porting Process.
2. When porting existing code to .NET Core or .NET Standard, install the NuGet package Microsoft.Windows.Compatibility.
3. If you want to stay on Windows, you're all set.
4. If you want to run the .NET Core application or .NET Standard library on Linux or macOS, use the API Analyzer to find usage of APIs that won't work cross-platform.
5. Either remove the usages of those APIs, replace them with cross-platform alternatives, or guard them using a platform check, like:

```
private static string GetLoggingPath()
{
    // Verify the code is running on Windows.
    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        using (var key = Registry.CurrentUser.OpenSubKey(@"Software\Fabrikam\AssetManagement"))
        {
            if (key?.GetValue("LoggingDirectoryPath") is string configuredPath)
                return configuredPath;
        }
    }

    // This is either not running on Windows or no logging path was configured,
    // so just use the path for non-roaming user-specific data files.
    var appDataPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
    return Path.Combine(appDataPath, "Fabrikam", "AssetManagement", "Logging");
}
```

For a demo, check out the Channel 9 video of the Windows Compatibility Pack.

How to: Port a Windows Forms desktop app to .NET Core

4/9/2019 • 8 minutes to read • Edit Online

This article describes how to port your Windows Forms-based desktop app from .NET Framework to .NET Core 3.0. The .NET Core 3.0 SDK includes support for Windows Forms applications. Windows Forms is still a Windows-only framework and only runs on Windows. This example uses the .NET Core SDK CLI to create and manage your project.

In this article, various names are used to identify types of files used for migration. When migrating your project, your files will be named differently, so mentally match them to the ones listed below:

FILE	DESCRIPTION
MyApps.sln	The name of the solution file.
MyForms.csproj	The name of the .NET Framework Windows Forms project to port.
MyFormsCore.csproj	The name of the new .NET Core project you create.
MyAppCore.exe	The .NET Core Windows Forms app executable.

Prerequisites

- Visual Studio 2019 for any designer work you want to do.

  Install the following Visual Studio workloads:

  - .NET desktop development
  - .NET cross-platform development
- A working Windows Forms project in a solution that builds and runs without issue.
- Your project must be coded in C#.
- Install the latest .NET Core 3.0 preview.

> **NOTE**
> **Visual Studio 2017** doesn't support .NET Core 3.0 projects. **Visual Studio 2019** supports .NET Core 3.0 projects but doesn't yet support the visual designer for .NET Core 3.0 Windows Forms projects. To use the visual designer, you must have a .NET Windows Forms project in your solution that shares the forms files with the .NET Core project.

Consider

When porting a .NET Framework Windows Forms application, there are a few things you must consider.

1. Check that your application is a good candidate for migration.

   Use the .NET Portability Analyzer to determine if your project will migrate to .NET Core 3.0. If your project has issues with .NET Core 3.0, the analyzer helps you identify those problems.

2. You're using a different version of Windows Forms.

   When .NET Core 3.0 Preview 1 was released, Windows Forms went open-source on GitHub. The code for .NET Core Windows Forms is a fork of the .NET Framework Windows Forms code base. It's possible some differences exist and your app won't port.

3. The Windows Compatibility Pack may help you migrate.

   Some APIs that are available in .NET Framework aren't available in .NET Core 3.0. The Windows Compatibility Pack adds many of these APIs and may help your Windows Forms app become compatible with .NET Core.

4. Update the NuGet packages used by your project.

   It's always a good practice to use the latest versions of NuGet packages before any migration. If your application is referencing any NuGet packages, update them to the latest version. Ensure your application builds successfully. After upgrading, if there are any package errors, downgrade the package to the latest version that doesn't break your code.

5. Visual Studio 2019 doesn't yet support the Forms Designer for .NET Core 3.0

   Currently, you need to keep your existing .NET Framework Windows Forms project file if you want to use the Forms Designer from Visual Studio.

Create a new SDK project

The new .NET Core 3.0 project you create must be in a different directory from your .NET Framework project. If they're both in the same directory, you may run into conflicts with the files that are generated in the **obj** directory. In this example, we'll create a directory named **MyFormsAppCore** in the **SolutionFolder** directory:

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
└──MyFormsAppCore      <--- New folder for core project
```

Next, you need to create the **MyFormsCore.csproj** project in the **MyFormsAppCore** directory. You can create this file manually by using the text editor of choice. Paste in the following XML:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

If you don't want to create the project file manually, you can use Visual Studio or the .NET Core SDK to generate the project. However, you must delete all other files generated by the project template except for the project file. To use the SDK, run the following command from the **SolutionFolder** directory:

```
dotnet new winforms -o MyFormsAppCore -n MyFormsCore
```

After you create the **MyFormsCore.csproj**, your directory structure should look like the following:

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
└──MyFormsAppCore
    └──MyFormsCore.csproj
```

You'll want to add the **MyFormsCore.csproj** project to **MyApps.sln** with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet sln add .\MyFormsAppCore\MyFormsCore.csproj
```

Fix assembly info generation

Windows Forms projects that were created with .NET Framework include an `AssemblyInfo.cs` file, which contains assembly attributes such as the version of the assembly to be generated. SDK-style projects automatically generate this information for you based on the SDK project file. Having both types of "assembly info" creates a conflict. Resolve this problem by disabling automatic generation, which forces the project to use your existing `AssemblyInfo.cs` file.

There are three settings to add to the main `<PropertyGroup>` node.

- **GenerateAssemblyInfo**
  When you set this property to `false`, it won't generate the assembly attributes. This avoids the conflict with the existing `AssemblyInfo.cs` file from the .NET Framework project.
- **AssemblyName**
  The value of this property is the output binary created when you compile. The name doesn't need an extension added to it. For example, using `MyCoreApp` produces `MyCoreApp.exe`.
- **RootNamespace**
  The default namespace used by your project. This should match the default namespace of the .NET Framework project.

Add these three elements to the `<PropertyGroup>` node in the `MyFormsCore.csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreApp</AssemblyName>
    <RootNamespace>WindowsFormsApp1</RootNamespace>
  </PropertyGroup>

</Project>
```

Add source code

Right now, the **MyFormsCore.csproj** project doesn't compile any code. By default, .NET Core projects automatically include all source code in the current directory and any child directories. You must configure the project to include code from the .NET Framework project using a relative path. If your .NET Framework project used **.resx** files for icons and resources for your forms, you'll need to include those too.

Add the following `<ItemGroup>` node to your project. Each statement includes a file glob pattern that includes child directories.

```
<ItemGroup>
  <Compile Include="..\MyFormsApp\**\*.cs" />
  <EmbeddedResource Include="..\MyFormsApp\**\*.resx" />
</ItemGroup>
```

Alternatively, you can create a `<Compile>` or `<EmbeddedResource>` entry for each file in your .NET Framework project.

Add NuGet packages

Add each NuGet package referenced by the .NET Framework project to the .NET Core project.

Most likely your .NET Framework Windows Forms app has a **packages.config** file that contains a list of all of the NuGet packages that are referenced by your project. You can look at this list to determine which NuGet packages to add to the .NET Core project. For example, if the .NET Framework project referenced the `MetroFramework`, `MetroFramework.Design`, and `MetroFramework.Fonts` NuGet packages, add each to the project with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Design
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Fonts
```

The previous commands would add the following NuGet references to the **MyFormsCore.csproj** project:

```
<ItemGroup>
  <PackageReference Include="MetroFramework" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Design" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Fonts" Version="1.2.0.3" />
</ItemGroup>
```

Port control libraries

If you have a Windows Forms Controls library project to port, the directions are the same as porting a .NET Framework Windows Forms app project, except for a few settings. And instead of compiling to an executable, you compile to a library. The difference between the executable project and the library project, besides paths for the file globs that include your source code, is minimal.

Using the previous step's example, lets expand what projects and files we're working with.

FILE	DESCRIPTION
MyApps.sln	The name of the solution file.
MyControls.csproj	The name of the .NET Framework Windows Forms Controls project to port.
MyControlsCore.csproj	The name of the new .NET Core library project you create.
MyCoreControls.dll	The .NET Core Windows Forms Controls library.

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
├──MyFormsAppCore
│   └──MyFormsCore.csproj
│
├──MyFormsControls
│   └──MyControls.csproj
└──MyFormsControlsCore
    └──MyControlsCore.csproj   <--- New project for core controls
```

Consider the differences between the `MyControlsCore.csproj` project and the previously created `MyFormsCore.csproj` project.

```
 <Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

   <PropertyGroup>
-    <OutputType>WinExe</OutputType>
     <TargetFramework>netcoreapp3.0</TargetFramework>
     <UseWindowsForms>true</UseWindowsForms>

     <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
-    <AssemblyName>MyCoreApp</AssemblyName>
-    <RootNamespace>WindowsFormsApp1</RootNamespace>
+    <AssemblyName>MyControlsCore</AssemblyName>
+    <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
   </PropertyGroup>

   <ItemGroup>
-    <Compile Include="..\MyFormsApp\**\*.cs" />
-    <EmbeddedResource Include="..\MyFormsApp\**\*.resx" />
+    <Compile Include="..\MyFormsControls\**\*.cs" />
+    <EmbeddedResource Include="..\MyFormsControls\**\*.resx" />
   </ItemGroup>

 </Project>
```

Here is an example of what the .NET Core Windows Forms Controls library project file would look like:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>

    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreControls</AssemblyName>
    <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="..\MyFormsControls\**\*.cs" />
    <EmbeddedResource Include="..\MyFormsControls\**\*.resx" />
  </ItemGroup>

</Project>
```

As you can see, the `<OutputType>` node was removed, which defaults the compiler to produce a library instead of an executable. The `<AssemblyName>` and `<RootNamespace>` were changed. Specifically the `<RootNamespace>` should match the namespace of the Windows Forms Controls library you are porting. And finally, the `<Compile>` and

`<EmbeddedResource>` nodes were adjusted to point to the folder of the Windows Forms Controls library you are porting.

Next, in the main .NET Core **MyFormsCore.csproj** project add reference to the new .NET Core Windows Forms Control library. Add a reference with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj reference .\MyFormsControlsCore\MyControlsCore.csproj
```

The previous command adds the following to the **MyFormsCore.csproj** project:

```
  <ItemGroup>
    <ProjectReference Include="..\MyFormsControlsCore\MyControlsCore.csproj" />
  </ItemGroup>
```

Problems compiling

If you have problems compiling your projects, you may be using some Windows-only APIs that are available in .NET Framework but not available in .NET Core. You can try adding the Windows Compatibility Pack NuGet package to your project. This package only runs on Windows and adds about 20,000 Windows APIs to .NET Core and .NET Standard projects.

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package Microsoft.Windows.Compatibility
```

The previous command adds the following to the **MyFormsCore.csproj** project:

```
  <ItemGroup>
    <PackageReference Include="Microsoft.Windows.Compatibility" Version="2.0.1" />
  </ItemGroup>
```

Windows Forms Designer

As detailed in this article, Visual Studio 2019 only supports the Forms Designer in .NET Framework projects. By creating a side-by-side .NET Core project, you can test your project with .NET Core while you use the .NET Framework project to design forms. Your solution file includes both the .NET Framework and .NET Core projects. Add and design your forms and controls in the .NET Framework project, and based on the file glob patterns we added to the .NET Core projects, any new or changed files will automatically be included in the .NET Core projects.

Once Visual Studio 2019 supports the Windows Forms Designer, you can copy/paste the content of your .NET Core project file into the .NET Framework project file. Then delete the file glob patterns added with the `<Source>` and `<EmbeddedResource>` items. Fix the paths to any project reference used by your app. This effectively upgrades the .NET Framework project to a .NET Core project.

Next steps

- Read more about the Windows Compatibility Pack.
- Watch a video on porting your .NET Framework Windows Forms project to .NET Core.

How to: Port a WPF desktop app to .NET Core

7/26/2019 • 7 minutes to read • Edit Online

This article describes how to port your Windows Presentation Foundation-based (WPF) desktop app from .NET Framework to .NET Core 3.0. The .NET Core 3.0 SDK includes support for WPF applications. WPF is still a Windows-only framework and only runs on Windows. This example uses the .NET Core SDK CLI to create and manage your project.

In this article, various names are used to identify types of files used for migration. When migrating your project, your files will be named differently, so mentally match them to the ones listed below:

FILE	DESCRIPTION
MyApps.sln	The name of the solution file.
MyWPF.csproj	The name of the .NET Framework WPF project to port.
MyWPFCore.csproj	The name of the new .NET Core project you create.
MyAppCore.exe	The .NET Core WPF app executable.

IMPORTANT

Even though this article uses C# as the target language, the steps are the same for VB.NET, except that VB.NET uses *.vbproj* and *.vb* files instead of *.csproj* and *.cs* files, respectively.

Prerequisites

- Visual Studio 2019 for any designer work you want to do.

  Install the following Visual Studio workloads:

  - .NET desktop development
  - .NET cross-platform development
- A working WPF project in a solution that builds and runs without issue.
- Install the latest .NET Core 3.0 preview.

NOTE

Visual Studio 2017 doesn't support .NET Core 3.0 projects. **Visual Studio 2019** supports .NET Core 3.0 projects but has limited support for the .NET Core WPF visual designer. To use a fully-supported visual designer, you must have a .NET Framework WPF project in your solution that shares its files with the .NET Core project.

Consider

When porting a .NET Framework WPF application, there are a few things you must consider.

1. Check that your application is a good candidate for migration.

   Use the .NET Portability Analyzer to determine if your project will migrate to .NET Core 3.0. If your project

has issues with .NET Core 3.0, the analyzer helps you identify those problems.

2. You're using a different version of WPF.

   When .NET Core 3.0 Preview 1 was released, WPF went open-source on GitHub. The code for .NET Core WPF is a fork of the .NET Framework WPF code base. It's possible some differences exist and your app won't port.

3. The Windows Compatibility Pack may help you migrate.

   Some APIs that are available in .NET Framework aren't available in .NET Core 3.0. The Windows Compatibility Pack adds many of these APIs and may help your WPF app become compatible with .NET Core.

4. Update the NuGet packages used by your project.

   It's always a good practice to use the latest versions of NuGet packages before any migration. If your application is referencing any NuGet packages, update them to the latest version. Ensure your application builds successfully. After upgrading, if there are any package errors, downgrade the package to the latest version that doesn't break your code.

5. Visual Studio 2019 doesn't yet support the WPF Designer for .NET Core 3.0

   Currently, you need to keep your existing .NET Framework WPF project file if you want to use the WPF Designer from Visual Studio.

Create a new SDK project

The new .NET Core 3.0 project you create must be in a different directory from your .NET Framework project. If they're both in the same directory, you may run into conflicts with the files that are generated in the **obj** directory. In this example, you'll create a directory named **MyWPFAppCore** in the **SolutionFolder** directory:

```
SolutionFolder
├───MyApps.sln
├───MyWPFApp
│   └───MyWPF.csproj
└───MyWPFAppCore      <--- New folder for core project
```

Next, you need to create the **MyWPFCore.csproj** project in the **MyWPFAppCore** directory. You can create this file manually by using the text editor of choice. Paste in the following XML:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

</Project>
```

If you don't want to create the project file manually, you can use Visual Studio or the .NET Core SDK to generate the project. However, you must delete all other files generated by the project template except for the project file. To use the SDK, run the following command from the **SolutionFolder** directory:

```
dotnet new wpf -o MyWPFAppCore -n MyWPFCore
```

After you create the **MyWPFCore.csproj**, your directory structure should look like the following:

```
SolutionFolder
├──MyApps.sln
├──MyWPFApp
│   └──MyWPF.csproj
└──MyWPFAppCore
    └──MyWPFCore.csproj
```

You'll want to add the **MyWPFCore.csproj** project to **MyApps.sln** with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet sln add .\MyWPFAppCore\MyWPFCore.csproj
```

Fix assembly info generation

Windows Presentation Foundation projects that were created with .NET Framework include an `AssemblyInfo.cs` file, which contains assembly attributes such as the version of the assembly to be generated. SDK-style projects automatically generate this information for you based on the SDK project file. Having both types of "assembly info" creates a conflict. Resolve this problem by disabling automatic generation, which forces the project to use your existing `AssemblyInfo.cs` file.

There are three settings to add to the main `<PropertyGroup>` node.

- **GenerateAssemblyInfo**
  When you set this property to `false`, it won't generate the assembly attributes. This avoids the conflict with the existing `AssemblyInfo.cs` file from the .NET Framework project.
- **AssemblyName**
  The value of this property is the output binary created when you compile. The name doesn't need an extension added to it. For example, using `MyCoreApp` produces `MyCoreApp.exe`.
- **RootNamespace**
  The default namespace used by your project. This should match the default namespace of the .NET Framework project.

Add these three elements to the `<PropertyGroup>` node in the `MyWPFCore.csproj` file:

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWPF>true</UseWPF>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreApp</AssemblyName>
    <RootNamespace>MyWPF</RootNamespace>
  </PropertyGroup>

</Project>
```

Add source code

Right now, the **MyWPFCore.csproj** project doesn't compile any code. By default, .NET Core projects automatically include all source code in the current directory and any child directories. You must configure the project to include

code from the .NET Framework project using a relative path. If your .NET Framework project used **.resx** files for icons and resources for your windows and controls, you'll need to include those too.

The first `<ItemGroup>` node you need to add to your project includes the **App.xaml** file that represents the startup config and resources your app uses. The **App.xaml** file also has an accompanying **App.xaml.cs** file, but it will be automatically included in a different `<ItemGroup>`.

```
<ItemGroup>
  <ApplicationDefinition Include="..\MyWPFApp\App.xaml">
    <Generator>MSBuild:Compile</Generator>
  </ApplicationDefinition>
</ItemGroup>
```

Next, add the following `<ItemGroup>` node to your project. Each statement includes a file glob pattern that includes child directories. It includes the source code for your project, any settings files, and any resources. The **obj** directory is explicitly excluded.

```
<ItemGroup>
  <Compile Include="..\MyWPFApp\**\*.cs" Exclude="..\MyWPFApp\obj\**" />
  <None Include="..\MyWPFApp\**\*.settings" />
  <EmbeddedResource Include="..\MyWPFApp\**\*.resx" />
</ItemGroup>
```

Next, include another `<ItemGroup>` node that contains a `<Page>` entry for every **xaml** file in your project except the **App.xaml** file. These contain all of the windows, pages, and resources that are in **xaml** format. You cannot use a glob pattern here and must add an entry for every file and indicate the `<Generator>` used.

```
<ItemGroup>
  <Page Include="..\MyWPFApp\MainWindow.xaml">
    <Generator>MSBuild:Compile</Generator>
  </Page>
</ItemGroup>
```

Add NuGet packages

Add each NuGet package referenced by the .NET Framework project to the .NET Core project.

Most likely your .NET Framework WPF app has a **packages.config** file that contains a list of all of the NuGet packages that are referenced by your project. You can look at this list to determine which NuGet packages to add to the .NET Core project. For example, if the .NET Framework project references the `MahApps.Metro` NuGet package, add it to the project with Visual Studio. You can also add the package reference with the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet add .\MyWPFAppCore\MyWPFCore.csproj package MahApps.Metro -v 2.0.0-alpha0262
```

The previous command would add the following NuGet reference to the **MyWPFCore.csproj** project:

```
<ItemGroup>
  <PackageReference Include="MahApps.Metro" Version="2.0.0-alpha0262" />
</ItemGroup>
```

Problems compiling

If you have problems compiling your projects, you may be using some Windows-only APIs that are available in .NET Framework but not available in .NET Core. You can try adding the Windows Compatibility Pack NuGet package to your project. This package only runs on Windows and adds about 20,000 Windows APIs to .NET Core and .NET Standard projects.

```
dotnet add .\MyWPFAppCore\MyWPFCore.csproj package Microsoft.Windows.Compatibility
```

The previous command adds the following to the **MyWPFCore.csproj** project:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Windows.Compatibility" Version="2.0.1" />
</ItemGroup>
```

WPF Designer

As detailed in this article, Visual Studio 2019 only supports the WPF Designer in .NET Framework projects. By creating a side-by-side .NET Core project, you can test your project with .NET Core while you use the .NET Framework project to design forms. Your solution file includes both the .NET Framework and .NET Core projects. Add and design your forms and controls in the .NET Framework project, and based on the file glob patterns we added to the .NET Core projects, any new or changed files will automatically be included in the .NET Core projects.

Once Visual Studio 2019 supports the WPF Designer, you can copy/paste the content of your .NET Core project file into the .NET Framework project file. Then delete the file glob patterns added with the `<Source>` and `<EmbeddedResource>` items. Fix the paths to any project reference used by your app. This effectively upgrades the .NET Framework project to a .NET Core project.

Next steps

- Read more about the Windows Compatibility Pack.
- Watch a video on porting your .NET Framework WPF project to .NET Core.

Dependency loading in .NET Core

8/24/2019 • 2 minutes to read • Edit Online

Every .NET Core application has dependencies. Even the simple `hello world` app has dependencies on portions of the .NET Core class libraries.

Understanding .NET Core default assembly loading logic can help understanding and debugging typical deployment issues.

In some applications, dependencies are dynamically determined at runtime. In these situations, it's critical to understand how managed assemblies and unmanaged dependencies are loaded.

Understanding AssemblyLoadContext

The AssemblyLoadContext API is central to the .NET Core loading design. The Understanding AssemblyLoadContext article provides a conceptual overview to the design.

Loading details

The loading algorithm details are covered briefly in several articles:

- Managed assembly loading algorithm
- Satellite assembly loading algorithm
- Unmanaged (native) library loading algorithm
- Default probing

Create a .NET Core application with plugins

The tutorial Create a .NET Core application with plugins describes how to create a custom AssemblyLoadContext. It uses an AssemblyDependencyResolver to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application.

How to use and debug assembly unloadability in .NET Core

The How to use and debug assembly unloadability in .NET Core article is a step-by-step tutorial. It shows how to load a .NET Core application, execute, and then unload it. The article also provides debugging tips.

Understanding System.Runtime.Loader.AssemblyLoadContext

8/24/2019 • 4 minutes to read • Edit Online

The AssemblyLoadContext class is unique to .NET Core. This article attempts to supplement the AssemblyLoadContext API documentation with conceptual information.

This article is relevant to developers implementing dynamic loading, especially dynamic loading framework developers.

What is the AssemblyLoadContext?

Every .NET Core application implicitly uses the AssemblyLoadContext. It's the runtime's provider for locating and loading dependencies. Whenever a dependency is loaded, an AssemblyLoadContext instance is invoked to locate it.

- It provides a service of locating, loading, and caching managed assemblies and other dependencies.
- To support dynamic code loading and unloading, it creates an isolated context for loading code and its dependencies in their own AssemblyLoadContext instance.

When do you need multiple AssemblyLoadContext instances?

A single AssemblyLoadContext instance is limited to loading exactly one version of an Assembly per simple assembly name, AssemblyName.Name.

This restriction can become a problem when loading code modules dynamically. Each module is independently compiled and may depend on different versions of an Assembly. This problem commonly occurs when different modules depend on different versions of a commonly used library.

To support dynamically loading code, the AssemblyLoadContext API provides for loading conflicting versions of an Assembly in the same application. Each AssemblyLoadContext instance provides a unique dictionary mapping each AssemblyName.Name to a specific Assembly instance.

It also provides a convenient mechanism for grouping dependencies related to a code module for later unload.

What is special about the AssemblyLoadContext.Default instance?

The AssemblyLoadContext.Default instance is automatically populated by the runtime at startup. It uses default probing to locate and find all static dependencies.

It solves the most common dependency loading scenarios.

How does AssemblyLoadContext support dynamic dependencies?

AssemblyLoadContext has various events and virtual functions that can be overridden.

The AssemblyLoadContext.Default instance only supports overriding the events.

The articles Managed assembly loading algorithm, Satellite assembly loading algorithm, and Unmanaged (native) library loading algorithm refer to all the available events and virtual functions. The articles show each event and function's relative position in the loading algorithms. This article doesn't reproduce that information.

This section covers the general principles for the relevant events and functions.

- **Be repeatable**. A query for a specific dependency must always result in the same response. The same loaded dependency instance must be returned. This requirement is fundamental for cache consistency. For managed assemblies in particular, we're creating a Assembly cache. The cache key is a simple assembly name, AssemblyName.Name.
- **Typically don't throw**. It's expected that these functions return `null` rather than throw when unable to find the requested dependency. Throwing will prematurely end the search and be propagate an exception to the caller. Throwing should be restricted to unexpected errors like a corrupted assembly or an out of memory condition.
- **Avoid recursion**. Be aware that these functions and handlers implement the loading rules for locating dependencies. Your implementation shouldn't call APIs that trigger recursion. Your code should typically call **AssemblyLoadContext** load functions that require a specific path or memory reference argument.
- **Load into the correct AssemblyLoadContext**. The choice of where to load dependencies is application-specific. The choice is implemented by these events and functions. When your code calls **AssemblyLoadContext** load-by-path functions call them on the instance where you want the code loaded. Sometime returning `null` and letting the AssemblyLoadContext.Default handle the load may be the simplest option.
- **Be aware of thread races**. Loading can be triggered by multiple threads. The AssemblyLoadContext handles thread races by atomically adding assemblies to its cache. The race loser's instance is discarded. In your implementation logic, don't add extra logic that doesn't handle multiple threads properly.

How are dynamic dependencies isolated?

Each AssemblyLoadContext instance represents a unique scope for Assembly instances and Type definitions.

There's no binary isolation between these dependencies. They're only isolated by not finding each other by name.

In each AssemblyLoadContext:

- AssemblyName.Name may refer to a different Assembly instance.
- Type.GetType may return a different type instance for the same type `name`.

How are dependencies shared?

Dependencies can easily be shared between AssemblyLoadContext instances. The general model is for one AssemblyLoadContext to load a dependency. The other shares the dependency by using a reference to the loaded assembly.

This sharing is required of the runtime assemblies. These assemblies can only be loaded into the AssemblyLoadContext.Default. The same is required for frameworks like `ASP.NET`, `WPF`, or `WinForms`.

It's recommended that shared dependencies should be loaded into AssemblyLoadContext.Default. This sharing is the common design pattern.

Sharing is implemented in the coding of the custom AssemblyLoadContext instance. AssemblyLoadContext has various events and virtual functions that can be overridden. When any of these functions return a reference to an Assembly instance that was loaded in another AssemblyLoadContext instance, the Assembly instance is shared. The standard load algorithm defers to AssemblyLoadContext.Default for loading to simplify the common sharing pattern. See Managed assembly loading algorithm.

Complications

Type conversion issues

When two AssemblyLoadContext instances contain type definitions with the same `name`, they're not the same type. They're the same type if and only if they come from the same Assembly instance.

To complicate matters, exception messages about these mismatched types can be confusing. The types are referred to in the exception messages by their simple type names. The common exception message in this case would be of the form:

```
Object of type 'IsolatedType' cannot be converted to type 'IsolatedType'.
```

Debugging type conversion issues

Given a pair of mismatched types it's important to also know:

- Each type's Type.Assembly
- Each type's AssemblyLoadContext, which can be obtained via the AssemblyLoadContext.GetLoadContext(Assembly) function.

Given two objects `a` and `b`, evaluating the following in the debugger will be helpful:

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

Resolving type conversion issues

There are two design patterns for solving these type conversion issues.

1. Use common shared types. This shared type can either be a primitive runtime type, or it can involve creating a new shared type in a shared assembly. Often the shared type is an interface defined in an application assembly. See also: How are dependencies shared?.

2. Use marshaling techniques to convert from one type to another.

Default probing

8/24/2019 • 2 minutes to read • Edit Online

The AssemblyLoadContext.Default instance is responsible for locating an assembly's dependencies. This article describes the AssemblyLoadContext.Default instance's probing logic.

Host configured probing properties

When the runtime is started, the runtime host provides a set of named probing properties that configure AssemblyLoadContext.Default probe paths.

Each probing property is optional. If present, each property is a string value that contains a delimited list of absolute paths. The delimiter is ';' on Windows and ':' on all other platforms.

PROPERTY NAME	DESCRIPTION
`TRUSTED_PLATFORM_ASSEMBLIES`	List of platform and application assembly file paths.
`PLATFORM_RESOURCE_ROOTS`	List of directory paths to search for satellite resource assemblies.
`NATIVE_DLL_SEARCH_DIRECTORIES`	List of directory paths to search for unmanaged (native) libraries.
`APP_PATHS`	List of directory paths to search for managed assemblies
`APP_NI_PATHS`	List of directory paths to search for native images of managed assemblies.

How are the properties populated?

There are two main scenarios for populating the properties depending on whether the `<myapp>.deps.json` file exists.

- When the `*.deps.json` file is present, it's parsed to populate the probing properties.
- When the `*.deps.json` file isn't present, the application's directory is assumed to contain all the dependencies. The directory's contents are used to populate the probing properties.

Additionally, the `*.deps.json` files for any referenced frameworks are similarly parsed.

Finally the environment variable `ADDITIONAL_DEPS` can be used to add additional dependencies.

How do I see the probing properties from managed code?

Each property is available by calling the AppContext.GetData(String) function with the property name from the table above.

How do I debug the probing properties' construction?

The .NET Core runtime host will output useful trace messages when certain environment variables are enabled:

ENVIRONMENT VARIABLE	DESCRIPTION
`COREHOST_TRACE=1`	Enables tracing.

ENVIRONMENT VARIABLE	DESCRIPTION
`COREHOST_TRACEFILE=<path>`	Traces to a file path instead of the default `stderr`.
`COREHOST_TRACE_VERBOSITY`	Sets the verbosity from 1 (lowest) to 4 (highest).

Managed assembly default probing

When probing to locate a managed assembly, the AssemblyLoadContext.Default looks in order at:

- Files matching the AssemblyName.Name in `TRUSTED_PLATFORM_ASSEMBLIES` (after removing file extensions).
- Native image assembly files in `APP_NI_PATHS` with common file extensions.
- Assembly files in `APP_PATHS` with common file extensions.

Satellite (resource) assembly probing

To find a satellite assembly for a specific culture, construct a set of file paths.

For each path in `PLATFORM_RESOURCE_ROOTS` and then `APP_PATHS`, append the CultureInfo.Name string, a directory separator, the AssemblyName.Name string, and the extension '.dll'.

If any matching file exists, attempt to load and return it.

Unmanaged (native) library probing

When probing to locate an unmanaged library, the `NATIVE_DLL_SEARCH_DIRECTORIES` are searched looking for a matching library,

Managed assembly loading algorithm

8/24/2019 • 2 minutes to read • Edit Online

Managed assemblies are located and loaded with an algorithm involving various stages.

All managed assemblies except satellite assemblies and `WinRT` assemblies use the same algorithm.

When are managed assemblies loaded?

The most common mechanism to trigger a managed assembly load is a static assembly reference. These references are inserted by the compiler whenever code uses a type defined in another assembly. These assemblies are loaded (`load-by-name`) as needed by the runtime.

The direct use of specific APIs will also trigger loads:

API	DESCRIPTION	`ACTIVE` ASSEMBLYLOADCONTEXT
AssemblyLoadContext.LoadFromAssemblyName	`Load-by-name`	The this instance.
AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath	Load from path.	The this instance.
AssemblyLoadContext.LoadFromStream	Load from object.	The this instance.
Assembly.LoadFile	Load from path in a new AssemblyLoadContext instance	The new AssemblyLoadContext instance.
Assembly.LoadFrom	Load from path in the AssemblyLoadContext.Default instance. Adds a Resolving handler to AssemblyLoadContext.Default. The handler will load the assembly's dependencies from its directory.	The AssemblyLoadContext.Default instance.
Assembly.Load(AssemblyName) Assembly.Load(String) Assembly.LoadWithPartialName	`Load-by-name` .	Inferred from caller. Prefer AssemblyLoadContext methods.
Assembly.Load(Byte[]) Assembly.Load(Byte[], Byte[])	Load from object.	Inferred from caller. Prefer AssemblyLoadContext methods.
Type.GetType(String) Type.GetType(String, Boolean) Type.GetType(String, Boolean, Boolean)	`Load-by-name` .	Inferred from caller. Prefer Type.GetType methods with an `assemblyResolver` argument.

API	DESCRIPTION	`ACTIVE` ASSEMBLYLOADCONTEXT
Assembly.GetType	If type `name` describes an assembly qualified generic type, trigger a `Load-by-name`.	Inferred from caller. Prefer Type.GetType when using assembly qualified type names.
Activator.CreateInstance(String, String) Activator.CreateInstance(String, String, Object[]) Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])	`Load-by-name`.	Inferred from caller. Prefer Activator.CreateInstance methods taking a Type argument.

Algorithm

The following algorithm describes how the runtime loads a managed assembly.

1. Determine the `active` AssemblyLoadContext.
   - For a static assembly reference, the `active` AssemblyLoadContext is the instance that loaded the referring assembly.
   - Preferred APIs make the `active` AssemblyLoadContext explicit.
   - Other APIs infer the `active` AssemblyLoadContext. For these APIs, the AssemblyLoadContext.CurrentContextualReflectionContext property is used. If its value is `null`, then the inferred AssemblyLoadContext instance is used.
   - See table above.
2. For the `Load-by-name` methods, the active AssemblyLoadContext loads the assembly. In priority order by:
   - Checking its `cache-by-name`.
   - Calling the AssemblyLoadContext.Load function.
   - Checking the AssemblyLoadContext.Default instances' cache and running managed assembly default probing logic.
   - Raising the AssemblyLoadContext.Resolving event for the active AssemblyLoadContext.
   - Raising the AppDomain.AssemblyResolve event.
3. For the other types of loads, the `active` AssemblyLoadContext loads the assembly. In priority order by:
   - Checking its `cache-by-name`.
   - Loading from the specified path or raw assembly object.
4. In either case, if an assembly is newly loaded, then:
   - The AppDomain.AssemblyLoad event is raised.
   - A reference is added to the assembly's AssemblyLoadContext instance's `cache-by-name`.

5. If the assembly is found, a reference is added as needed to the `active` AssemblyLoadContext instance's `cache-by-name`.

Satellite assembly loading algorithm

8/24/2019 • 2 minutes to read • Edit Online

Satellite assemblies are used to store localized resources customized for language and culture.

Satellite assemblies use a different loading algorithm than general managed assemblies.

When are satellite assemblies loaded?

Satellite assemblies are loaded when loading a localized resource.

The basic API to load localized resources is the System.Resources.ResourceManager class. Ultimately the ResourceManager class will call the GetSatelliteAssembly method for each CultureInfo.Name.

Higher-level APIs may abstract the low-level API.

Algorithm

The .NET Core resource fallback process involves the following steps:

1. Determine the `active` AssemblyLoadContext instance. In all cases, the `active` instance is the executing assembly's AssemblyLoadContext.
2. The `active` instance attempts to load a satellite assembly for the requested culture in priority order by:
   - Checking its cache.
   - Checking the directory of the currently executing assembly for a subdirectory that matches the requested CultureInfo.Name (for example `es-MX`).

     > **NOTE**
     > This feature was not implemented in .NET Core before 3.0.

     > **NOTE**
     > On Linux and macOS, the subdirectory is case-sensitive and must either:
     >
     > - Exactly match case.
     > - Be in lower case.

   - If `active` is the AssemblyLoadContext.Default instance, by running the default satellite (resource) assembly probing logic.
   - Calling the AssemblyLoadContext.Load function.
   - Raising the AssemblyLoadContext.Resolving event.
   - Raising the AppDomain.AssemblyResolve event.
3. If a satellite assembly is loaded:
   - The AppDomain.AssemblyLoad event is raised.
   - The assembly is searched it for the requested resource. If the runtime finds the resource in the assembly,

it uses it. If it doesn't find the resource, it continues the search.

> **NOTE**
> To find a resource within the satellite assembly, the runtime searches for the resource file requested by the ResourceManager for the current CultureInfo.Name. Within the resource file, it searches for the requested resource name. If either is not found, the resource is treated as not found.

4. The runtime next searches the parent culture assemblies through many potential levels, each time repeating steps 2 & 3.

   Each culture has only one parent, which is defined by the CultureInfo.Parent property.

   The search for parent cultures stops when a culture's Parent property is CultureInfo.InvariantCulture.

   For the InvariantCulture, we don't return to steps 2 & 3, but rather continue with step 5.

5. If the resource is still not found, the resource for the default (fallback) culture is used.

   Typically, the resources for the default culture are included in the main application assembly. However, you can specify UltimateResourceFallbackLocation.Satellite for the NeutralResourcesLanguageAttribute.Location property. This value indicates that the ultimate fallback location for resources is a satellite assembly rather than the main assembly.

   > **NOTE**
   > The default culture is the ultimate fallback. Therefore, we recommend that you always include an exhaustive set of resources in the default resource file. This helps prevent exceptions from being thrown. By having an exhaustive set, you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

6. Finally,

   - If the runtime doesn't find a resource file for a default (fallback) culture, a MissingManifestResourceException or MissingSatelliteAssemblyException exception is thrown.
   - If the resource file is found but the requested resource isn't present, the request returns `null`.

Unmanaged (native) library loading algorithm

8/24/2019 • 2 minutes to read • Edit Online

Unmanaged libraries are located and loaded with an algorithm involving various stages.

The following algorithm describes how native libraries are loaded through `PInvoke`.

`PInvoke` load library algorithm

`PInvoke` uses the following algorithm when attempting to load an unmanaged assembly:

1. Determine the `active` AssemblyLoadContext. For an unmanaged load library, the `active` AssemblyLoadContext is the caller of the `PInvoke`.
2. For the `active` AssemblyLoadContext, try to find the assembly in priority order by:
   - Checking its cache.
   - Calling the current System.Runtime.InteropServices.DllImportResolver delegate set by the NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver) function.
   - Calling the AssemblyLoadContext.LoadUnmanagedDll function.
   - Checking the AppDomain instance's cache and running the Unmanaged (native) library probing logic.
   - Raising the AssemblyLoadContext.ResolvingUnmanagedDll event for the `active` AssemblyLoadContext.
3. If the unmanaged library is newly loaded, the AppDomain.AssemblyLoad event is raised.

Create a .NET Core application with plugins

7/19/2019 • 8 minutes to read • Edit Online

This tutorial shows you how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the `System.Runtime.Loader.AssemblyDependencyResolver` type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET Core 3.0 Preview 2 SDK or a newer version.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run `dotnet new console -o AppWithPlugin`.
2. To make building the project easier, create a Visual Studio solution file. Run `dotnet new sln` in the same folder.
3. Run `dotnet sln add AppWithPlugin/AppWithPlugin.csproj` to add the app project to the solution.

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it is still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions do not conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Create a simple plugin with no dependencies

Back in the root folder, do the following:

1. Run `dotnet new classlib -o HelloPlugin` to create a new class library project named `HelloPlugin`.
2. Run `dotnet sln add HelloPlugin/HelloPlugin.csproj` to add the project to the `AppWithPlugin` solution.
3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

In between the `<Project>` tags, add the following elements:

```
<ItemGroup>
<ProjectReference Include="..\PluginBase\PluginBase.csproj">
    <Private>false</Private>
</ProjectReference>
</ItemGroup>
```

The `<Private>false</Private>` element is very important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method will not find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Now that the `HelloPlugin` project is complete, we should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"` as an element of the `pluginPaths` array.

Create a plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJson` plugin projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. The project files themselves do not have any special information for the project references, and (after adding the plugin paths to the `pluginPaths` array) the plugins run perfectly, even if run in the same execution of the AppWithPlugin app. However, these projects do not copy the referenced assemblies to their output directory, so the assemblies need to be present on the user's machine for the plugins to work. There are two ways to work around this problem. The first option is to use the `dotnet publish` command to publish the class library. Alternatively, if you want to be able to use the output of `dotnet build` for your plugin, you can add the `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` property between the `<PropertyGroup>` tags in the plugin's project file. See the `XcopyablePlugin` plugin project for an example.

Other plugin examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository. The completed sample includes a few other examples of `AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

How to reference a plugin interface assembly defined in a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET Core 3.0, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

How to use and debug assembly unloadability in .NET Core

7/19/2019 • 12 minutes to read • Edit Online

Starting with .NET Core 3.0, the ability to load and later unload a set of assemblies is supported. In .NET Framework, custom app domains were used for this purpose, but .NET Core only supports a single default app domain.

.NET Core 3.0 and later versions support unloadability through AssemblyLoadContext. You can load a set of assemblies into a collectible `AssemblyLoadContext`, execute methods in them or just inspect them using reflection, and finally unload the `AssemblyLoadContext`. That unloads the assemblies loaded into that `AssemblyLoadContext`.

There's one noteworthy difference between the unloading using `AssemblyLoadContext` and using AppDomains. With AppDomains, the unloading is forced. At the unload time, all threads running in the target AppDomain are aborted, managed COM objects created in the target AppDomain are destroyed, etc. With `AssemblyLoadContext`, the unload is "cooperative". Calling the AssemblyLoadContext.Unload method just initiates the unloading. The unloading finishes after:

- No threads have methods from the assemblies loaded into the `AssemblyLoadContext` on their call stacks.
- None of the types from the assemblies loaded into the `AssemblyLoadContext`, instances of those types and the assemblies themselves outside of the `AssemblyLoadContext` are referenced by:
  - References outside of the `AssemblyLoadContext`, except of weak references (WeakReference or WeakReference<T>).
  - Strong GC handles (GCHandleType.Normal or GCHandleType.Pinned) from both inside and outside of the `AssemblyLoadContext`.

Using collectible AssemblyLoadContext

This section contains a detailed step-by-step tutorial that shows a simple way to load a .NET Core application into a collectible `AssemblyLoadContext`, execute its entry point, and then unload it. You can find a complete sample at https://github.com/dotnet/samples/tree/master/core/tutorials/Unloading.

Create a collectible AssemblyLoadContext

You need to derive your class from the AssemblyLoadContext and overload its AssemblyLoadContext.Load method. That method resolves references to all assemblies that are dependencies of assemblies loaded into that `AssemblyLoadContext`. The following code is an example of the simplest custom `AssemblyLoadContext`:

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly Load(AssemblyName name)
    {
        return null;
    }
}
```

As you can see, the `Load` method returns `null`. That means that all the dependency assemblies are loaded into the default context, and the new context contains only the assemblies explicitly loaded into it.

If you want to load some or all of the dependencies into the `AssemblyLoadContext` too, you can use the `AssemblyDependencyResolver` in the `Load` method. The `AssemblyDependencyResolver` resolves the assembly names to absolute assembly file paths using the *.deps.json* file contained in the directory of the main assembly loaded into the context and using assembly files in that directory.

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    private AssemblyDependencyResolver _resolver;

    public TestAssemblyLoadContext(string mainAssemblyToLoadPath) : base(isCollectible: true)
    {
        _resolver = new AssemblyDependencyResolver(mainAssemblyToLoadPath);
    }

    protected override Assembly Load(AssemblyName name)
    {
        string assemblyPath = _resolver.ResolveAssemblyToPath(name);
        if (assemblyPath != null)
        {
            return LoadFromAssemblyPath(assemblyPath);
        }

        return null;
    }
}
```

Use a custom collectible AssemblyLoadContext

This section assumes the simpler version of the `TestAssemblyLoadContext` is being used.

You can create an instance of the custom `AssemblyLoadContext` and load an assembly into it as follows:

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

For each of the assemblies referenced by the loaded assembly, the `TestAssemblyLoadContext.Load` method is called so that the `TestAssemblyLoadContext` can decide where to get the assembly from. In our case, it returns `null` to indicate that it should be loaded into the default context from locations that the runtime uses to load assemblies by default.

Now that an assembly was loaded, you can execute a method from it. Run the `Main` method:

```
var args = new object[1] { new string[] {"Hello"}};
int result = (int)a.EntryPoint.Invoke(null, args);
```

After the `Main` method returns, you can initiate unloading by either calling the `Unload` method on the custom `AssemblyLoadContext` or getting rid of the reference you have to the `AssemblyLoadContext`:

```
alc.Unload();
```

This is sufficient to unload the test assembly. Let's actually put all of this into a separate non-inlineable method to ensure that the `TestAssemblyLoadContext`, `Assembly`, and `MethodInfo` (the `Assembly.EntryPoint`) can't be kept alive by stack slot references (real- or JIT-introduced locals). That could keep the `TestAssemblyLoadContext` alive and prevent the unload.

Also, return a weak reference to the `AssemblyLoadContext` so that you can use it later to detect unload completion.

```
[MethodImpl(MethodImplOptions.NoInlining)]
static int ExecuteAndUnload(string assemblyPath, out WeakReference alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] { new string[] {"Hello"}};
    int result = (int)a.EntryPoint.Invoke(null, args);

    alc.Unload();

    return result;
}
```

Now you can run this function to load, execute, and unload the assembly.

```
WeakReference testAlcWeakRef;
int result = ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

However, the unload doesn't complete immediately. As previously mentioned, it relies on the GC to collect all the objects from the test assembly. In many cases, it isn't necessary to wait for the unload completion. However, there are cases where it's useful to know that the unload has finished. For example, you may want to delete the assembly file that was loaded into the custom `AssemblyLoadContext` from disk. In such a case, the following code snippet can be used. It triggers a GC and waits for pending finalizers in a loop until the weak reference to the custom `AssemblyLoadContext` is set to `null`, indicating the target object was collected. Note that in most cases, just one pass through the loop is required. However, for more complex cases where objects created by the code running in the `AssemblyLoadContext` have finalizers, more passes may be needed.

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

The Unloading event

In some cases, it may be necessary for the code loaded into a custom `AssemblyLoadContext` to perform some cleanup when the unloading is initiated. For example, it may need to stop threads, clean up some strong GC handles, etc. The `Unloading` event can be used in such cases. A handler that performs the necessary cleanup can be hooked to this event.

Troubleshoot unloadability issues

Due to the cooperative nature of the unloading, it's easy to forget about references keeping the stuff in a collectible `AssemblyLoadContext` alive and preventing unload. Here is a summary of things (some of them non-obvious) that can hold the references:

- Regular references held from outside of the collectible `AssemblyLoadContext`, stored in a stack slot or a processor register (method locals, either explicitly created by the user code or implicitly by the JIT), a static variable or a strong / pinning GC handle, and transitively pointing to:
  - An assembly loaded into the collectible `AssemblyLoadContext`.
  - A type from such an assembly.
  - An instance of a type from such an assembly.
- Threads running code from an assembly loaded into the collectible `AssemblyLoadContext`.

- Instances of custom non-collectible `AssemblyLoadContext` types created inside of the collectible `AssemblyLoadContext`
- Pending RegisteredWaitHandle instances with callbacks set to methods in the custom `AssemblyLoadContext`

Hints to find stack slot / processor register rooting an object:

- Passing function call results directly to another function may create a root even though there is no user-created local variable.
- If a reference to an object was available at any point in a method, the JIT might have decided to keep the reference in a stack slot / processor register for as long as it wants in the current function.

Debug unloading issues

Debugging issues with unloading can be tedious. You can get into situations where you don't know what can be holding an `AssemblyLoadContext` alive, but the unload fails. The best weapon to help with that is WinDbg (LLDB on Unix) with the SOS plugin. You need to find what's keeping a `LoaderAllocator` belonging to the specific `AssemblyLoadContext` alive. This plugin allows you to look at GC heap objects, their hierarchies, and roots. To load the plugin into the debugger, enter the following command in the debugger command line:

In WinDbg (it seems WinDbg does that automatically when breaking into .NET Core application):

```
.loadby sos coreclr
```

In LLDB:

```
plugin load /path/to/libsosplugin.so
```

Let's try to debug an example program that has problems with unloading. Source code is included below. When you run it under WinDbg, the program breaks into the debugger right after attempting to check for the unload success. You can then start looking for the culprits.

Note that if you debug using LLDB on Unix, the SOS commands in the following examples don't have the `!` in front of them.

```
!dumpheap -type LoaderAllocator
```

This command dumps all objects with a type name containing `LoaderAllocator` that are in the GC heap. Here is an example:

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24 System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

In the "Statistics:" part below, check the `MT` (`MethodTable`) belonging to the `System.Reflection.LoaderAllocator`, which is the object we care about. Then in the list at the beginning, find the entry with `MT` matching that one and get the address of the object itself. In our case, it is "000002b78000ce40"

Now that we know the address of the `LoaderAllocator` object, we can use another command to find its GC roots

```
!gcroot -all 0x000002b78000ce40
```

This command dumps the chain of object references that lead to the `LoaderAllocator` instance. The list starts with the root, which is the entity that keeps our `LoaderAllocator` alive and thus is the core of the problem you're debugging. The root can be a stack slot, a processor register, a GC handle, or a static variable.

Here is an example of the output of the `gcroot` command:

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

Now you need to figure out where the root is located so you can fix it. The easiest case is when the root is a stack slot or a processor register. In that case, the `gcroot` shows you the name of the function whose frame contains the root and the thread executing that function. The difficult case is when the root is a static variable or a GC handle.

In the previous example, the first root is a local of type `System.Reflection.RuntimeMethodInfo` stored in the frame of the function `example.Program.Main(System.String[])` at address `rbp-20` (`rbp` is the processor register `rbp` and -20 is a hexadecimal offset from that register).

The second root is a normal (strong) `GCHandle` that holds a reference to an instance of the `test.Test` class.

The third root is a pinned `GCHandle`. This one is actually a static variable. Unfortunately, there is no way to tell. Statics for reference types are stored in a managed object array in internal runtime structures.

Another case that can prevent unloading of an `AssemblyLoadContext` is when a thread has a frame of a method from an assembly loaded into the `AssemblyLoadContext` on its stack. You can check that by dumping managed call stacks of all threads:

```
~*e !clrstack
```

The command means "apply to all threads the `!clrstack` command". The following is the output of that command for the example. Unfortunately, LLDB on Unix doesn't have any way to apply a command to all threads, so you'll need to resort to manually switching threads and repeating the `clrstack` command. Ignore all threads where the debugger says "Unable to walk the managed stack."

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[]) [E:\unloadability\example\Program.cs @
70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame: 0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc() [E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame: 0000001fc727f7f0]
```

As you can see, the last thread has `test.Program.ThreadProc()`. This is a function from the assembly loaded into the `AssemblyLoadContext`, and so it keeps the `AssemblyLoadContext` alive.

Example source with unloadability issues

The following code is used in the previous debugging example.

Main testing program

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
```

```
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo mi)
        {
            entryPoint = mi;
        }
        MethodInfo entryPoint;
    }

    class Program
    {
        static TestInfo entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference testAlcWeakRef, out MethodInfo
testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a method for an assembly loaded into the
TestAssemblyLoadContext in a static variable
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            int result = (int)a.EntryPoint.Invoke(null, args);
            alc.Unload();

            return result;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a method for an assembly loaded into the
TestAssemblyLoadContext in a local variable
            MethodInfo testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out testAlcWeakRef, out
testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
```

```
            System.Diagnostics.Debugger.Break();

            Console.WriteLine($"Test completed, result={result}, entryPoint: {testEntryPoint} unload success:
{!testAlcWeakRef.IsAlive}");
        }
    }
}
```

Program loaded into the TestAssemblyLoadContext

The following code represents the `test.dll` passed to the `ExecuteAndUnload` method in the main testing program.

```
using System;
using System.Runtime.InteropServices;

namespace test
{
    class Test
    {
        string message = "Hello";
    }

    class Program
    {
        public static void ThreadProc()
        {
            // Issue preventing unlopading #4 - a thread running method inside of the TestAssemblyLoadContext
at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

Build .NET Core from source

1/23/2019 • 3 minutes to read • Edit Online

The ability to build .NET Core from its source code is important in multiple ways: it makes it easier to port .NET Core to new platforms, it enables contributions and fixes to the product, and it enables the creation of custom versions of .NET. This article gives guidance to developers who want to build and distribute their own versions of .NET Core.

Build the CLR from source

The source code for the .NET CoreCLR can be found in the dotnet/coreclr repository on GitHub.

The build currently depends on the following prerequisites:

- Git
- CMake
- Python
- a C++ compiler.

After you've installed these prerequisites, you can build the CLR by invoking the build script (`build.cmd` on Windows, or `build.sh` on Linux and macOS) at the base of the dotnet/coreclr repository.

Installing the components differ depending on the operating system (OS). See the build instructions for your specific OS:

- Windows
- Linux
- macOS
- FreeBSD
- NetBSD

There is no cross-building across OS (only for ARM, which is built on X64).
You have to be on the particular platform to build that platform.

The build has two main `buildTypes`:

- Debug (default)- Compiles the runtime with minimal optimizations and additional runtime checks (asserts). This reduction in optimization level and the additional checks slow runtime execution but are valuable for debugging. This is the recommended setting for development and testing environments.
- Release - Compiles the runtime with full optimizations and without the additional runtime checks. This will yield much faster run time performance but it can take a bit longer to build and can be difficult to debug. Pass `release` to the build script to select this build type.

In addition, by default the build not only creates the runtime executables, but it also builds all the tests. There are quite a few tests, taking a significant amount of time that isn't necessary if you just want to experiment with changes. You can skip the tests builds by adding the `skiptests` argument to the build script, like in the following example (replace `.\build` with `./build.sh` on Unix machines):

```
.\build skiptests
```

The previous example showed how to build the `Debug` flavor, which has development time checks (asserts) enabled

and optimizations disabled. To build the release (full speed) flavor, do the following:

```
.\build release skiptests
```

You can find more build options with build by using the -? or -help qualifier.

Using Your Build

The build places all of its generated files under the `bin` directory at the base of the repository. There is a *bin\Log* directory that contains log files generated during the build (Most useful when the build fails). The actual output is placed in a *bin\Product[platform].[CPU architecture].[build type]* directory, such as *bin\Product\Windows_NT.x64.Release*.

While the 'raw' output of the build is sometimes useful, normally you're only interested in the NuGet packages, which are placed in the `.nuget\pkg` subdirectory of the previous output directory.

There are two basic techniques for using your new runtime:

1. **Use dotnet.exe and NuGet to compose an application**. See Using Your Build for instructions on creating a program that uses your new runtime by using the NuGet packages you just created and the 'dotnet' command-line interface (CLI). This technique is the expected way non-runtime developers are likely to consume your new runtime.

2. **Use corerun.exe to run an application using unpackaged DLLs**. This repository also defines a simple host called corerun.exe that does NOT take any dependency on NuGet. You need to tell the host where to get the required DLLs you actually use, and you have to manually gather them together. This technique is used by all the tests in the dotnet/coreclr repo, and is useful for quick local 'edit-compile-debug' loop such as preliminary unit testing. See Executing .NET Core Apps with CoreRun.exe for details on using this technique.

Build the CLI from source

The source code for the .NET Core CLI can be found in the dotnet/cli repository on GitHub.

In order to build the .NET Core CLI, you need the following installed on your machine.

- Windows & Linux:
  - git on the PATH
- macOS:
  - git on the PATH
  - Xcode
  - OpenSSL

In order to build, run `build.cmd` on Windows, or `build.sh` on Linux and macOS from the root. If you don't want to execute tests, run `build.cmd -t:Compile` or `./build.sh -t:Compile`. To build the CLI in macOS Sierra, you need to set the DOTNET_RUNTIME_ID environment variable by running `export DOTNET_RUNTIME_ID=osx.10.11-x64`.

Using your build

Use the `dotnet` executable from *artifacts/{os}-{arch}/stage2* to try out the newly built CLI. If you want to use the build output when invoking `dotnet` from the current console, you can also add *artifacts/{os}-{arch}/stage2* to the PATH.

See also

- .NET Core Common Language Runtime (CoreCLR)

- .NET Core CLI Developer Guide
- .NET Core distribution packaging

.NET Core distribution packaging

4/28/2019 • 5 minutes to read • Edit Online

As .NET Core becomes available on more and more platforms, it's useful to learn how to package, name, and version it. This way, package maintainers can help ensure a consistent experience no matter where users choose to run .NET. This article is useful for users that are:

- Attempting to build .NET Core from source.
- Wanting to make changes to the .NET Core CLI that could impact the resulting layout or packages produced.

Disk layout

When installed, .NET Core consists of several components that are laid out as follows in the filesystem:

```
.
├── dotnet                      (1)
├── LICENSE.txt                 (8)
├── ThirdPartyNotices.txt       (8)
├── host
│   └── fxr
│       └── <fxr version>       (2)
├── sdk
│   ├── <sdk version>           (3)
│   └── NuGetFallbackFolder     (4)
└── shared
    ├── Microsoft.NETCore.App
    │   └── <runtime version>   (5)
    └── Microsoft.AspNetCore.App
        └── <aspnetcore version> (6)
    └── Microsoft.AspNetCore.All
        └── <aspnetcore version> (7)
/
├─usr/share/man/man1
│       └── dotnet.1.gz         (9)
└─usr/bin
        └── dotnet              (10)
```

- (1) **dotnet** The host (also known as the "muxer") has two distinct roles: activate a runtime to launch an application, and activate an SDK to dispatch commands to it. The host is a native executable (`dotnet.exe`).

While there's a single host, most of the other components are in versioned directories (2,3,5,6). This means multiple versions can be present on the system since they're installed side by side.

- (2) **host/fxr/<fxr version>** contains the framework resolution logic used by the host. The host uses the latest hostfxr that is installed. The hostfxr is responsible for selecting the appropriate runtime when executing a .NET Core application. For example, an application built for .NET Core 2.0.0 uses the 2.0.5 runtime when it's available. Similarly, hostfxr selects the appropriate SDK during development.

- (3) **sdk/<sdk version>** The SDK (also known as "the tooling") is a set of managed tools that are used to write and build .NET Core libraries and applications. The SDK includes the .NET Core Command-line interface (CLI), the managed languages compilers, MSBuild, and associated build tasks and targets, NuGet, new project templates, and so on.

- (4) **sdk/NuGetFallbackFolder** contains a cache of NuGet packages used by an SDK during the restore operation, such as when running `dotnet restore` or `dotnet build /t:Restore` .

The **shared** folder contains frameworks. A shared framework provides a set of libraries at a central location so they can be used by different applications.

- (5) **shared/Microsoft.NETCore.App/<runtime version>** This framework contains the .NET Core runtime and supporting managed libraries.
- (6,7) **shared/Microsoft.AspNetCore.{App,All}/<aspnetcore version>** contains the ASP.NET Core libraries. The libraries under `Microsoft.AspNetCore.App` are developed and supported as part of the .NET Core project. The libraries under `Microsoft.AspNetCore.All` are a superset that also contains third-party libraries.
- (8) **LICENSE.txt,ThirdPartyNotices.txt** are the .NET Core license and licenses of third-party libraries used in .NET Core, respectively.
- (9,10) **dotnet.1.gz, dotnet** `dotnet.1.gz` is the dotnet manual page. `dotnet` is a symlink to the dotnet host(1). These files are installed at well known locations for system integration.

Recommended packages

.NET Core versioning is based on the runtime component `[major].[minor]` version numbers. The SDK version uses the same `[major].[minor]` and has an independent `[patch]` that combines feature and patch semantics for the SDK. For example: SDK version 2.2.302 is the second patch release of the third feature release of the SDK that supports the 2.2 runtime. For more information about how versioning works, see .NET Core versioning overview.

Some of the packages include part of the version number in their name. This allows you to install a specific version. The rest of the version isn't included in the version name. This allows the OS package manager to update the packages (for example, automatically installing security fixes). Supported package managers are Linux specific.

The following table shows the recommended packages:

NAME	EXAMPLE	USE CASE: INSTALL ...	CONTAINS	DEPENDENCIES	VERSION
dotnet-sdk-[major]	dotnet-sdk-2	Latest sdk for runtime major		dotnet-sdk-[major].[latestminor]	<sdk version>
dotnet-sdk-[major].[minor]	dotnet-sdk-2.1	Latest sdk for specific runtime		dotnet-sdk-[major].[minor].[latest sdk feat]xx	<sdk version>
dotnet-sdk-[major].[minor].[sdk feat]xx	dotnet-sdk-2.1.3xx	Specific sdk feature release	(3),(4)	aspnetcore-runtime-[major].[minor]	<sdk version>
aspnetcore-runtime-[major].[minor]	aspnetcore-runtime-2.1	Specific ASP.NET Core runtime	(6),[(7)]	dotnet-runtime-[major].[minor]	<runtime version>
dotnet-runtime-[major].[minor]	dotnet-runtime-2.1	Specific runtime	(5)	host-fxr: <runtime version>+	<runtime version>
dotnet-host-fxr	dotnet-host-fxr	*dependency*	(2)	host:<runtime version>+	<runtime version>

NAME	EXAMPLE	USE CASE: INSTALL ...	CONTAINS	DEPENDENCIES	VERSION
dotnet-host	dotnet-host	*dependency*	(1),(8),(9),(10)		<runtime version>

Most distributions require all artifacts to be built from source. This has some impact on the packages:

- The third-party libraries under `shared/Microsoft.AspNetCore.All` can't be easily built from source. So that folder is omitted from the `aspnetcore-runtime` package.
- The `NuGetFallbackFolder` is populated using binary artifacts from `nuget.org`. It should remain empty.

Multiple `dotnet-sdk` packages may provide the same files for the `NuGetFallbackFolder`. To avoid issues with the package manager, these files should be identical (checksum, modification date, and so on).

Preview versions

Package maintainers may decide to provide preview versions of the shared framework and SDK. Preview releases may be provided using the `dotnet-sdk-[major].[minor].[sdk feat]xx`, `aspnetcore-runtime-[major].[minor]`, or `dotnet-runtime-[major].[minor]` packages. For preview releases, the package version major must be set to zero. This way, the final release is installed as an upgrade of the package.

Patch packages

Since a patch version of a package may cause a breaking change, a package maintainer may want to provide *patch packages*. These packages allow you to install a specific patch version that isn't automatically upgraded. Only use patch packages in rare circumstances as they aren't upgraded with (security) fixes.

The following table shows the recommended packages and **patch packages**:

NAME	EXAMPLE	CONTAINS	DEPENDENCIES
dotnet-sdk-[major]	dotnet-sdk-2		dotnet-sdk-[major].[latest sdk minor]
dotnet-sdk-[major].[minor]	dotnet-sdk-2.1		dotnet-sdk-[major].[minor].[latest sdk feat]xx
dotnet-sdk-[major].[minor].[sdk feat]xx	dotnet-sdk-2.1.3xx		dotnet-sdk-[major].[minor].[latest sdk patch]
dotnet-sdk-[major].[minor].[patch]	dotnet-sdk-2.1.300	(3),(4)	aspnetcore-runtime-[major].[minor].[sdk runtime patch]
aspnetcore-runtime-[major].[minor]	aspnetcore-runtime-2.1		aspnetcore-runtime-[major].[minor].[latest runtime patch]
aspnetcore-runtime-[major].[minor].[patch]	aspnetcore-runtime-2.1.0	(6),[(7)]	dotnet-runtime-[major].[minor].[patch]
dotnet-runtime-[major].[minor]	dotnet-runtime-2.1		dotnet-runtime-[major].[minor].[latest runtime patch]
dotnet-runtime-[major].[minor].[patch]	dotnet-runtime-2.1.0	(5)	host-fxr:<runtime version>+

NAME	EXAMPLE	CONTAINS	DEPENDENCIES
dotnet-host-fxr	dotnet-host-fxr	(2)	host:<runtime version>+
dotnet-host	dotnet-host	(1),(8),(9),(10)	

An alternative to using patch packages is *pinning* the packages to a specific version using the package manager. To avoid affecting other applications/users, such applications can be built and deployed in a container.

Building packages

The dotnet/source-build repository provides instructions on how to build a source tarball of the .NET Core SDK and all its components. The output of the source-build repository matches the layout described in the first section of this article.

project.json and Visual Studio 2015 with .NET Core

1/17/2019 • 2 minutes to read • Edit Online

On March 7, 2017, the .NET Core and ASP.NET Core documentation was updated for the release of Visual Studio 2017. The previous version of the documentation used Visual Studio 2015 and pre-release tooling based on the *project.json* file.

The documentation version from before the March 7 update is available in a PDF file and in a branch in the documentation repository.

PDF documentation

The best source of the earlier documentation is the .NET Core - PDF for project.json and Visual Studio 2015.

Documentation repository branch

You can view the earlier version of the documentation in the repository, but many links won't work and many code snippets are references that aren't expanded.

- .NET Core - project.json branch in the documentation repository

Current version of the documentation

- .NET Core documentation
- ASP.NET Core documentation

.NET Framework Guide

5/15/2019 • 2 minutes to read • Edit Online

NOTE

This .NET Framework content set includes information for .NET Framework versions 4.5 through 4.8. To download the .NET Framework, see Installing the .NET Framework. For a list of new features and changes in the NET Framework, see What's New in the .NET Framework. For a list of supported platforms, see .NET Framework System Requirements. For documentation about older versions of the .NET Framework, see .NET previous versions documentation.

The .NET Framework is a development platform for building apps for web, Windows, Windows Phone, Windows Server, and Microsoft Azure. It consists of the common language runtime (CLR) and the .NET Framework class library, which includes a broad range of functionality and support for many industry standards.

The .NET Framework provides many services, including memory management, type and memory safety, security, networking, and application deployment. It provides easy-to-use data structures and APIs that abstract the lower-level Windows operating system. You can use different programming languages with the .NET Framework, including C#, F#, and Visual Basic.

For a general introduction to the .NET Framework for both users and developers, see Getting Started. For an introduction to the architecture and key features of the .NET Framework, see the overview.

The .NET Framework can be used with Docker and with Windows Containers.

Installation

The .NET Framework comes with Windows, enabling you to run .NET Framework applications. You may need a later version of the .NET Framework than the one that comes with your Windows version. For more information, see Install the .NET Framework on Windows.

See Repair the .NET Framework to learn how to repair your .NET Framework installation if you're experiencing errors when installing the .NET Framework.

For more detailed information on downloading the .NET Framework, see Install the .NET Framework for developers.

In this section

- What's New
  Describes key new features and changes in the latest versions of the .NET Framework. Includes lists of obsolete types and members, and provides a guide for migrating your applications from the previous version of the .NET Framework.
- Get Started
  Provides a comprehensive overview of the .NET Framework and links to additional resources.
- Installation Guide
  Provides resources and guidance about .NET Framework installation and troubleshooting.
- Migration Guide
  Provides resources and a list of changes you need to consider if you're migrating your application to a new version of the .NET Framework.

- Development Guide
  Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.
- Tools
  Describes the tools that help you develop, configure, and deploy applications by using .NET Framework technologies.
- Additional class libraries and APIs
  Provides documentation for private .NET Framework APIs used by Microsoft tools.

See also

- .NET Framework Class Library

What's new in the .NET Framework

8/23/2019 • 86 minutes to read • Edit Online

This article summarizes key new features and improvements in the following versions of the .NET Framework:

- .NET Framework 4.8
- .NET Framework 4.7.2
- .NET Framework 4.7.1
- .NET Framework 4.7
- .NET Framework 4.6.2
- .NET Framework 4.6.1
- .NET 2015 and .NET Framework 4.6
- .NET Framework 4.5.2
- .NET Framework 4.5.1
- .NET Framework 4.5

This article does not provide comprehensive information about each new feature and is subject to change. For general information about the .NET Framework, see Getting Started. For supported platforms, see System Requirements. For download links and installation instructions, see Installation Guide.

> **NOTE**
> The .NET Framework team also releases features out of band with NuGet to expand platform support and to introduce new functionality, such as immutable collections and SIMD-enabled vector types. For more information, see Additional Class Libraries and APIs and The .NET Framework and Out-of-Band Releases. See a complete list of NuGet packages for the .NET Framework.

Introducing .NET Framework 4.8

.NET Framework 4.8 builds on previous versions of the .NET Framework 4.x by adding many new fixes and several new features while remaining a very stable product.

Downloading and installing .NET Framework 4.8

You can download .NET Framework 4.8 from the following locations:

- .NET Framework 4.8 Web Installer
- NET Framework 4.8 Offline Installer

.NET Framework 4.8 can be installed on Windows 10, Windows 8.1, Windows 7 SP1, and the corresponding server platforms starting with Windows Server 2008 R2 SP1. You can install .NET Framework 4.8 by using either the web installer or the offline installer. The recommended way for most users is to use the web installer.

You can target .NET Framework 4.8 in Visual Studio 2012 or later by installing the .NET Framework 4.8 Developer Pack.

What's new in .NET Framework 4.8

.NET Framework 4.8 introduces new features in the following areas:

- Base classes
- Windows Communication Foundation (WCF)
- Windows Presentation Foundation (WPF)
- Common language runtime

Improved accessibility, which allows an application to provide an appropriate experience for users of Assistive Technology, continues to be a major focus of .NET Framework 4.8. For information on accessibility improvements in .NET Framework 4.8, see What's new in accessibility in the .NET Framework.

Base classes

Reduced FIPS impact on Cryptography. In previous versions of the .NET Framework, managed cryptographic provider classes such as SHA256Managed throw a CryptographicException when the system cryptographic libraries are configured in "FIPS mode". These exceptions are thrown because the managed versions of the cryptographic provider classes, unlike the system cryptographic libraries, have not undergone FIPS (Federal Information Processing Standards) 140-2 certification. Because few developers have their development machines in FIPS mode, the exceptions are commonly thrown in production systems.

By default in applications that target .NET Framework 4.8, the following managed cryptography classes no longer throw a CryptographicException in this case:

- MD5Cng
- MD5CryptoServiceProvider
- RC2CryptoServiceProvider

- RijndaelManaged
- RIPEMD160Managed
- SHA256Managed

Instead, these classes redirect cryptographic operations to a system cryptography library. This change effectively removes a potentially confusing difference between developer environments and production environments and makes native components and managed components operate under the same cryptographic policy. Applications that depend on these exceptions can restore the previous behavior by setting the AppContext switch `Switch.System.Security.Cryptography.UseLegacyFipsThrow` to `true`. For more information, see Managed cryptography classes do not throw a CryptographyException in FIPS mode.

Use of updated version of ZLib

Starting with .NET Framework 4.5, the clrcompression.dll assembly uses ZLib, a native external library for data compression, in order to provide an implementation for the deflate algorithm. The .NET Framework 4.8, clrcompression.dll is updated to use ZLib Version 1.2.11, which includes several key improvements and fixes.

Windows Communication Foundation (WCF)

Introduction of ServiceHealthBehavior

Health endpoints are widely used by orchestration tools to manage services based on their health status. Health checks can also be used by monitoring tools to track and provide notifications about the availability and performance of a service.

ServiceHealthBehavior is a WCF service behavior that extends IServiceBehavior. When added to the ServiceDescription.Behaviors collection, a service behavior does the following:

- Returns service health status with HTTP response codes. You can specify in a query string the HTTP status code for a HTTP/GET health probe request.
- Publishes information about service health. Service-specific details, including service state, throttle counts, and capacity can be displayed by using an HTTP/GET request with the `?health` query string. Ease of access to such information is important when troubleshooting a misbehaving WCF service.

There are two ways to expose the health endpoint and publish WCF service health information:

- Through code. For example:

```
ServiceHost host = new ServiceHost(typeof(Service1),
                   new Uri("http://contoso:81/Service1"));
ServiceHealthBehavior healthBehavior =
    host.Description.Behaviors.Find<ServiceHealthBehavior>();
if (healthBehavior == null)
{
   healthBehavior = new ServiceHealthBehavior();
}
host.Description.Behaviors.Add(healthBehavior);
```

```
Dim host As New ServiceHost(GetType(Service1),
            New Uri("http://contoso:81/Service1"))
Dim healthBehavior As ServiceHealthBehavior =
   host.Description.Behaviors.Find(Of ServiceHealthBehavior)()
If healthBehavior Is Nothing Then
   healthBehavior = New ServiceHealthBehavior()
End If
host.Description.Behaviors.Add(healthBehavior)
```

- By using a configuration file. For example:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="DefaultBehavior">
      <serviceHealth httpsGetEnabled="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

A service's health status can be queried by using query parameters such as `OnServiceFailure`, `OnDispatcherFailure`, `OnListenerFailure`, `OnThrottlePercentExceeded`), and an HTTP response code can be specified for each query parameter. If the HTTP response code is omitted for a query parameter, a 503 HTTP response code is used by default. For example:

- OnServiceFailure: `https://contoso:81/Service1?health&OnServiceFailure=450`

  A 450 HTTP response status code is returned when ServiceHost.State is greater than CommunicationState.Opened. Query parameters and examples:

- OnDispatcherFailure: `https://contoso:81/Service1?health&OnDispatcherFailure=455`

  A 455 HTTP response status code is returned when the state of any of the channel dispatchers is greater than CommunicationState.Opened.

- OnListenerFailure: `https://contoso:81/Service1?health&OnListenerFailure=465`

  A 465 HTTP response status code is returned when the state of any of the channel listeners is greater than CommunicationState.Opened.

- OnThrottlePercentExceeded: `https://contoso:81/Service1?health&OnThrottlePercentExceeded= 70:350,95:500`

  Specifies the percentage {1 – 100} that triggers the response and its HTTP response code {200 – 599}. In this example:

  - If the percentage is greater than 95, a 500 HTTP response code is returned.
  - If the percentage or between 70 and 95, 350 is returned.
  - Otherwise, 200 is returned.

The service health status can be displayed either in HTML by specifying a query string like `https://contoso:81/Service1?health` or in XML by specifying a query string like `https://contoso:81/Service1?health&Xml`. A query string like `https://contoso:81/Service1?health&NoContent` returns an empty HTML page.

Windows Presentation Foundation (WPF)

High DPI enhancements

In .NET Framework 4.8, WPF adds support for Per-Monitor V2 DPI Awareness and Mixed-Mode DPI scaling. See High DPI Desktop Application Development on Windows for additional information about high DPI development.

.NET Framework 4.8 improves support for hosted HWNDs and Windows Forms interoperation in High-DPI WPF applications on platforms that support Mixed-Mode DPI scaling (starting with Windows 10 April 2018 Update). When hosted HWNDs or Windows Forms controls are created as Mixed-Mode DPI-scaled windows by calling SetThreadDpiHostingBehavior and SetThreadDpiAwarenessContext, they can be hosted in a Per-Monitor V2 WPF application and are sized and scaled appropriately. Such hosted content is not rendered at the native DPI; instead, the operating system scales the hosted content to the appropriate size. The support for Per-Monitor v2 DPI awareness mode also allows WPF controls to be hosted (i.e., parented) in a native window in a high-DPI application.

To enable support for Mixed-Mode High DPI scaling, you can set the following AppContext switches the application configuration file:

```
<runtime>
   <AppContextSwitchOverrides value = "Switch.System.Windows.DoNotScaleForDpiChanges=false;
Switch.System.Windows.DoNotUsePresentationDpiCapabilityTier2OrGreater=false"/>
</runtime>
```

Common language runtime

The runtime in .NET Framework 4.8 includes the following changes and improvements:

Improvements to the JIT compiler. The Just-in-time (JIT) compiler in .NET Framework 4.8 is based on the JIT compiler in .NET Core 2.1. Many of the optimizations and all of the bug fixes made to the .NET Core 2.1 JIT compiler are included in the .NET Framework 4.8 JIT compiler.

NGEN improvements. The runtime has improved its memory management for Native Image Generator (NGEN) images so that data mapped from NGEN images are not memory-resident. This reduces the surface area available to attacks that attempt to execute arbitrary code by modifying memory that will be executed.

Antimalware scanning for all assemblies. In previous versions of the .NET Framework, the runtime scans all assemblies loaded from disk using either Windows Defender or third-party antimalware software. However, assemblies loaded from other sources, such as by the Assembly.Load(Byte[]) method, are not scanned and can potentially contain undetected malware. Starting with .NET Framework 4.8 running on Windows 10, the runtime triggers a scan by antimalware solutions that implement the Antimalware Scan Interface (AMSI).

What's new in .NET Framework 4.7.2

.NET Framework 4.7.2 includes new features in the following areas:

- Base classes
- ASP.NET
- Networking
- SQL
- WPF
- ClickOnce

A continuing focus in .NET Framework 4.7.2 is improved accessibility, which allows an application to provide an appropriate experience for users of Assistive Technology. For information on accessibility improvements in .NET Framework 4.7.2, see What's new in accessibility in the .NET Framework.

Base classes

.NET Framework 4.7.2 features a large number of cryptographic enhancements, better decompression support for ZIP archives, and additional collection APIs.

New overloads of RSA.Create and DSA.Create

The DSA.Create(DSAParameters) and RSA.Create(RSAParameters) methods let you supply key parameters when instantiating a new DSA or RSA key. They allow you to replace code like the following:

```
// Before .NET Framework 4.7.2
using (RSA rsa = RSA.Create())
{
   rsa.ImportParameters(rsaParameters);
   // Other code to execute using the RSA instance.
}
```

```
' Before .NET Framework 4.7.2
Using rsa = RSA.Create()
   rsa.ImportParameters(rsaParameters)
   ' Other code to execute using the rsa instance.
End Using
```

with code like this:

```
// Starting with .NET Framework 4.7.2
using (RSA rsa = RSA.Create(rsaParameters))
{
   // Other code to execute using the rsa instance.
}
```

```
' Starting with .NET Framework 4.7.2
Using rsa = RSA.Create(rsaParameters)
   ' Other code to execute using the rsa instance.
End Using
```

The DSA.Create(Int32) and RSA.Create(Int32) methods let you generate new DSA or RSA keys with a specific key size. For example:

```
using (DSA dsa = DSA.Create(2048))
{
   // Other code to execute using the dsa instance.
}
```

```
Using dsa = DSA.Create(2048)
   ' Other code to execute using the dsa instance.
End Using
```

Rfc2898DeriveBytes constructors accept a hash algorithm name

The Rfc2898DeriveBytes class has three new constructors with a HashAlgorithmName parameter that identifies the HMAC algorithm to use when deriving keys. Instead of using SHA-1, developers should use a SHA-2-based HMAC like SHA-256, as shown in the following example:

```
private static byte[] DeriveKey(string password, out int iterations, out byte[] salt,
                                out HashAlgorithmName algorithm)
{
   iterations = 100000;
   algorithm = HashAlgorithmName.SHA256;

   const int SaltSize = 32;
   const int DerivedValueSize = 32;

   using (Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(password, SaltSize,
                                                             iterations, algorithm))
   {
      salt = pbkdf2.Salt;
      return pbkdf2.GetBytes(DerivedValueSize);
   }
}
```

```
Private Shared Function DeriveKey(password As String, ByRef iterations As Integer,
                                  ByRef salt AS Byte(), ByRef algorithm As HashAlgorithmName) As Byte()
   iterations = 100000
   algorithm = HashAlgorithmName.SHA256

   Const SaltSize As Integer = 32
   Const  DerivedValueSize As Integer = 32

   Using pbkdf2 = New Rfc2898DeriveBytes(password, SaltSize, iterations, algorithm)
      salt = pbkdf2.Salt
      Return pbkdf2.GetBytes(DerivedValueSize)
   End Using
End Function
```

Support for ephemeral keys

PFX import can optionally load private keys directly from memory, bypassing the hard drive. When the new X509KeyStorageFlags.EphemeralKeySet

flag is specified in an X509Certificate2 constructor or one of the overloads of the X509Certificate2.Import method, the private keys will be loaded as ephemeral keys. This prevents the keys from being visible on the disk. However:

- Since the keys are not persisted to disk, certificates loaded with this flag are not good candidates to add to an X509Store.
- Keys loaded in this manner are almost always loaded via Windows CNG. Therefore, callers must access the private key by calling extension methods, such as cert.GetRSAPrivateKey(). The X509Certificate2.PrivateKey property does not function.
- Since the legacy X509Certificate2.PrivateKey property does not work with certificates, developers should perform rigorous testing before switching to ephemeral keys.

Programmatic creation of PKCS#10 certification signing requests and X.509 public key certificates

Starting with .NET Framework 4.7.2, workloads can generate certificate signing requests (CSRs), which allows certificate request generation to be staged into existing tooling. This is frequently useful in test scenarios.

For more information and code examples, see "Programmatic creation of PKCS#10 certification signing requests and X.509 public key certificates" in the .NET Blog.

New SignerInfo members

Starting with .NET Framework 4.7.2, the SignerInfo class exposes more information about the signature. You can retrieve the value of the System.Security.Cryptography.Pkcs.SignerInfo.SignatureAlgorithm property to determine the signature algorithm used by the signer. SignerInfo.GetSignature can be called to get a copy of the cryptographic signature for this signer.

Leaving a wrapped stream open after CryptoStream is disposed

Starting with .NET Framework 4.7.2, the CryptoStream class has an additional constructor that allows Dispose to not close the wrapped stream. To leave the wrapped stream open after the CryptoStream instance is disposed, call the new CryptoStream constructor as follows:

```
var cStream = new CryptoStream(stream, transform, mode, leaveOpen: true);
```

```
Dim cStream = New CryptoStream(stream, transform, mode, leaveOpen:=true)
```

Decompression changes in DeflateStream

Starting with .NET Framework 4.7.2, the implementation of decompression operations in the DeflateStream class has changed to use native Windows APIs by default. Typically, this results in a substantial performance improvement.

Support for decompression by using Windows APIs is enabled by default for applications that target .NET Framework 4.7.2. Applications that target earlier versions of .NET Framework but are running under .NET Framework 4.7.2 can opt into this behavior by adding the following AppContext switch to the application configuration file:

```
<AppContextSwitchOverrides value="Switch.System.IO.Compression.DoNotUseNativeZipLibraryForDecompression=false" />
```

Additional collection APIs

.NET Framework 4.7.2 adds a number of new APIs to the SortedSet<T> and HashSet<T> types. These include:

- `TryGetValue` methods, which extend the try pattern used in other collection types to these two types. The methods are:
  - public bool HashSet<T>.TryGetValue(T equalValue, out T actualValue)
  - public bool SortedSet<T>.TryGetValue(T equalValue, out T actualValue)
- `Enumerable.To*` extension methods, which convert a collection to a HashSet<T>:
  - public static HashSet<TSource> ToHashSet<TSource>(this IEnumerable<TSource> source)
  - public static HashSet<TSource> ToHashSet<TSource>(this IEnumerable<TSource> source, IEqualityComparer<TSource> comparer)
- New HashSet<T> constructors that let you set the collection's capacity, which yields a performance benefit when you know the size of the HashSet<T> in advance:
  - public HashSet(int capacity)
  - public HashSet(int capacity, IEqualityComparer<T> comparer)

The ConcurrentDictionary<TKey,TValue> class includes new overloads of the AddOrUpdate and GetOrAdd methods to retrieve a value from the dictionary or to add it if it is not found, and to add a value to the dictionary or to update it if it already exists.

```
public TValue AddOrUpdate<TArg>(TKey key, Func<TKey, TArg, TValue> addValueFactory, Func<TKey, TValue, TArg, TValue> updateValueFactory, TArg
factoryArgument)

public TValue GetOrAdd<TArg>(TKey key, Func<TKey, TArg, TValue> valueFactory, TArg factoryArgument)
```

```
Public AddOrUpdate(Of TArg)(key As TKey, addValueFactory As Func(Of TKey, TArg, TValue), updateValueFactory As Func(Of TKey, TValue, TArg, TValue),
factoryArgument As TArg) As TValue

Public GetOrAdd(Of TArg)(key As TKey, valueFactory As Func(Of TKey, TArg, TValue), factoryArgument As TArg) As TValue
```

ASP.NET

Support for dependency injection in Web Forms

Dependency injection (DI) decouples objects and their dependencies so that an object's code no longer needs to be changed just because a dependency has changed. When developing ASP.NET applications that target .NET Framework 4.7.2, you can:

- Use setter-based, interface-based, and constructor-based injection in handlers and modules, Page instances, and user controls of ASP.NET web application projects.
- Use setter-based and interface-based injection in handlers and modules, Page instances, and user controls of ASP.NET web site projects.
- Plug in different dependency injection frameworks.

Support for same-site cookies

SameSite prevents a browser from sending a cookie along with a cross-site request. .NET Framework 4.7.2 adds a HttpCookie.SameSite property whose value is a System.Web.SameSiteMode enumeration member. If its value is SameSiteMode.Strict or SameSiteMode.Lax, ASP.NET adds the `SameSite` attribute to the set-cookie header. SameSite support applies to HttpCookie objects, as well as to FormsAuthentication and System.Web.SessionState cookies.

You can set SameSite for an HttpCookie object as follows:

```
var c = new HttpCookie("secureCookie", "same origin");
c.SameSite = SameSiteMode.Lax;
```

```
Dim c As New HttpCookie("secureCookie", "same origin")
c.SameSite = SameSiteMode.Lax
```

You can also configure SameSite cookies at the application level by modifying the web.config file:

```
<system.web>
   <httpCookies sameSite="Strict" />
</system.web>
```

You can add SameSite for FormsAuthentication and System.Web.SessionState cookies by modifying the web config file:

```
<system.web>
   <authentication mode="Forms">
      <forms cookieSameSite="Lax">
         <!-- ...   -->
      </forms>
   <authentication />
   <sessionSate cookieSameSite="Lax"></sessionState>
</system.web>
```

Networking

Implementation of HttpClientHandler properties

.NET Framework 4.7.1 added eight properties to the System.Net.Http.HttpClientHandler class. However, two threw a PlatformNotSupportedException. .NET Framework 4.7.2 now provides an implementation for these properties. The properties are:

- CheckCertificateRevocationList
- SslProtocols

SQLClient

Support for Azure Active Directory Universal Authentication and Multi-Factor authentication

Growing compliance and security demands require that many customers use multi-factor authentication (MFA). In addition, current best practices discourage including user passwords directly in connection strings. To support these changes, .NET Framework 4.7.2 extends SQLClient connection strings by adding a new value, "Active Directory Interactive", for the existing "Authentication" keyword to support MFA and Azure AD Authentication. The new interactive method supports native and federated Azure AD users as well as Azure AD guest users. When this method is used, the MFA authentication imposed by Azure AD is supported for SQL databases. In addition, the authentication process requests a user password to adhere to security best practices.

In previous versions of the .NET Framework, SQL connectivity supported only the SqlAuthenticationMethod.ActiveDirectoryPassword and SqlAuthenticationMethod.ActiveDirectoryIntegrated options. Both of these are part of the non-interactive ADAL protocol, which does not support MFA. With the new SqlAuthenticationMethod.ActiveDirectoryInteractive option, SQL connectivity supports MFA as well as existing authentication methods (password and integrated authentication), which allows users to enter user passwords interactively without persisting passwords in the connection string.

For more information and an example, see "SQL -- Azure AD Universal and Multi-factor Authentication Support" in the .NET Blog.

Support for Always Encrypted version 2

NET Framework 4.7.2 adds supports for enclave-based Always Encrypted. The original version of Always Encrypted is a client-side encryption technology in which encryption keys never leave the client. In enclave-based Always Encrypted, the client can optionally send the encryption keys to a secure enclave, which is a secure computational entity that can be considered part of SQL Server but that SQL Server code cannot tamper with. To support enclave-based Always Encrypted, .NET Framework 4.7.2 adds the following types and members to the System.Data.SqlClient namespace:

- SqlConnectionStringBuilder.EnclaveAttestationUrl, which specifies the Uri for enclave-based Always Encrypted.
- SqlColumnEncryptionEnclaveProvider, which is an abstract class from which all enclave providers are derived.
- SqlEnclaveSession, which encapsulates the state for a given enclave session.
- SqlEnclaveAttestationParameters, which provides the attestation parameters used by SQL Server to get information required to execute a particular Attestation Protocol.

The application configuration file then specifies a concrete implementation of the abstract System.Data.SqlClient.SqlColumnEncryptionEnclaveProvider class that provides the functionality for the enclave provider. For example:

```
<configuration>
  <configSections>
    <section name="SqlColumnEncryptionEnclaveProviders"
type="System.Data.SqlClient.SqlColumnEncryptionEnclaveProviderConfigurationSection,System.Data,Version=4.0.0.0,Culture=neutral,PublicKeyToken=b77a5
c561934e089"/>
  </configSections>
  <SqlColumnEncryptionEnclaveProviders>
    <providers>
      <add name="Azure" type="Microsoft.SqlServer.Management.AlwaysEncrypted.AzureEnclaveProvider,MyApp"/>
      <add name="HGS" type="Microsoft.SqlServer.Management.AlwaysEncrypted.HGSEnclaveProvider,MyApp" />
    </providers>
  </SqlColumnEncryptionEnclaveProviders >
</configuration>
```

The basic flow of enclave-based Always Encrypted is:

1. The user creates an AlwaysEncrypted connection to SQL Server that supported enclave-based Always Encrypted. The driver contacts the attestation service to ensure that it is connecting to right enclave.
2. Once the enclave has been attested, the driver establishes a secure channel with the secure enclave hosted on SQL Server.
3. The driver shares encryption keys authorized by the client with the secure enclave for the duration of the SQL connection.

Windows Presentation Foundation

Finding ResourceDictionaries by Source

Starting with .NET Framework 4.7.2, a diagnostic assistant can locate the ResourceDictionaries that have been created from a given source Uri. (This feature is for use by diagnostic assistants, not by production applications.) A diagnostic assistant such as Visual Studio's "Edit-and-Continue" facility lets its user edit a ResourceDictionary with the intent that the changes be applied to the running application. One step in achieving this is finding all the ResourceDictionaries that the running application has created from the dictionary that's being edited. For example, an application can declare a ResourceDictionary whose content is copied from a given source URI:

```
<ResourceDictionary Source="MyRD.xaml">
```

A diagnostic assistant that edits the original markup in *MyRD.xaml* can use the new feature to locate the dictionary. The feature is implemented by a new static method, ResourceDictionaryDiagnostics.GetResourceDictionariesForSource. The diagnostic assistant calls the new method using an absolute Uri that identifies the original markup, as illustrated by the following code:

```
IEnumerable<ResourceDictionary> dictionaries = ResourceDictionaryDiagnostics.GetResourceDictionariesForSource(new
Uri("pack://application:,,,/MyApp;component/MyRD.xaml"));
```

```
Dim dictionaries As IEnumerable(Of ResourceDictionary) = ResourceDictionaryDiagnostics.GetResourceDictionariesForSource(New
Uri("pack://application:,,,/MyApp;component/MyRD.xaml"))
```

The method returns an empty enumerable unless VisualDiagnostics is enabled and the `ENABLE_XAML_DIAGNOSTICS_SOURCE_INFO` environment variable is set.

Finding ResourceDictionary owners

Starting with .NET Framework 4.7.2, a diagnostic assistant can locate the owners of a given ResourceDictionary. (The feature is for use by diagnostic assistants and not by production applications.) Whenever a change is made to a ResourceDictionary, WPF automatically finds all DynamicResource references that might be affected by the change.

A diagnostic assistant such as Visual Studio's "Edit-and-Continue" facility may want extend this to handle StaticResource references. The first step in this process is to find the owners of the dictionary; that is, to find all the objects whose `Resources` property refers to the dictionary (either directly, or

indirectly via the ResourceDictionary.MergedDictionaries property). Three new static methods implemented on the System.Windows.Diagnostics.ResourceDictionaryDiagnostics class, one for each of the base types that has a `Resources` property, support this step:

- `public static IEnumerable<FrameworkElement> GetFrameworkElementOwners(ResourceDictionary dictionary);`
- `public static IEnumerable<FrameworkContentElement> GetFrameworkContentElementOwners(ResourceDictionary dictionary);`
- `public static IEnumerable<Application> GetApplicationOwners(ResourceDictionary dictionary);`

These methods return an empty enumerable unless VisualDiagnostics is enabled and the `ENABLE_XAML_DIAGNOSTICS_SOURCE_INFO` environment variable is set.

Finding StaticResource references

A diagnostic assistant can now receive a notification whenever a StaticResource reference is resolved. (The feature is for use by diagnostic assistants, not by production applications.) A diagnostic assistant such as Visual Studio's "Edit-and-Continue" facility may want to update all uses of a resource when its value in a ResourceDictionary changes. WPF does this automatically for DynamicResource references, but it intentionally does not do so for StaticResource references. Starting with .NET Framework 4.7.2, the diagnostic assistant can use these notifications to locate those uses of the static resource.

The notification is implemented by the new ResourceDictionaryDiagnostics.StaticResourceResolved event:

```
public static event EventHandler<StaticResourceResolvedEventArgs> StaticResourceResolved;
```

```
Public Shared Event StaticResourceResolved As EventHandler(Of StaticResourceResolvedEventArgs)
```

This event is raised whenever the runtime resolves a StaticResource reference. The StaticResourceResolvedEventArgs arguments describe the resolution, and indicate the object and property that host the StaticResource reference and the ResourceDictionary and key used for the resolution:

```
public class StaticResourceResolvedEventArgs : EventArgs
{
   public Object TargetObject { get; }

   public Object TargetProperty { get; }

   public ResourceDictionary ResourceDictionary { get; }

   public object ResourceKey { get; }
}
```

```
Public Class StaticResourceResolvedEventArgs : Inherits EventArgs
   Public ReadOnly Property TargetObject As Object
   Public ReadOnly Property TargetProperty As Object
   Public ReadOnly Property ResourceDictionary As ResourceDictionary
   Public ReadOnly Property ResourceKey As Object
End Class
```

The event is not raised (and its `add` accessor is ignored) unless VisualDiagnostics is enabled and the `ENABLE_XAML_DIAGNOSTICS_SOURCE_INFO` environment variable is set.

ClickOnce

HDPI-aware applications for Windows Forms, Windows Presentation Foundation (WPF), and Visual Studio Tools for Office (VSTO) can all be deployed by using ClickOnce. If the following entry is found in the application manifest, deployment will succeed under .NET Framework 4.7.2:

```
<windowsSettings>
   <dpiAware xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">true</dpiAware>
</windowsSettings>
```

For Windows Forms application, the previous workaround of setting DPI awareness in the application configuration file rather than the application manifest is no longer necessary for ClickOnce deployment to succeed.

What's new in .NET Framework 4.7.1

.NET Framework 4.7.1 includes new features in the following areas:

- Base classes
- Common language runtime (CLR)
- Networking
- ASP.NET

In addition, a major focus in .NET Framework 4.7.1 is improved accessibility, which allows an application to provide an appropriate experience for users of Assistive Technology. For information on accessibility improvements in .NET Framework 4.7.1, see What's new in accessibility in the .NET Framework.

Base classes

Support for .NET Standard 2.0

.NET Standard defines a set of APIs that must be available on each .NET implementation that supports that version of the standard. .NET Framework 4.7.1 fully supports .NET Standard 2.0 and adds about 200 APIs that are defined in .NET Standard 2.0 and are missing from .NET Framework 4.6.1, 4.6.2, and 4.7. (Note that these versions of the .NET Framework support .NET Standard 2.0 only if additional .NET Standard support files are also deployed on the target system.) For more information, see "BCL - .NET Standard 2.0 Support" in the .NET Framework 4.7.1 Runtime and Compiler Features blog post.

Support for configuration builders

Configuration builders allow developers to inject and build configuration settings for applications dynamically at run time. Custom configuration builders can be used to modify existing data in a configuration section or to build a configuration section entirely from scratch. Without configuration builders, .config files are static, and their settings are defined some time before an application is launched.

To create a custom configuration builder, you derive your builder from the abstract ConfigurationBuilder class and override its ConfigurationBuilder.ProcessConfigurationSection and ConfigurationBuilder.ProcessRawXml. You also define your builders in your .config file. For more information, see the "Configuration Builders" section in the .NET Framework 4.7.1 ASP.NET and Configuration Features blog post.

Run-time feature detection

The System.Runtime.CompilerServices.RuntimeFeature class provides a mechanism for determine whether a predefined feature is supported on a given .NET implementation at compile time or run time. At compile time, a compiler can check whether a specified field exists to determine whether the feature is supported; if so, it can emit code that takes advantage of that feature. At run time, an application can call the RuntimeFeature.IsSupported method before emitting code at runtime. For more information, see Add helper method to describe features supported by the runtime.

Value tuple types are serializable

Starting with .NET Framework 4.7.1, System.ValueTuple and its associated generic types are marked as Serializable, which allows binary serialization. This should make migrating Tuple types, such as Tuple<T1,T2,T3> and Tuple<T1,T2,T3,T4>, to value tuple types easier. For more information, see "Compiler -- ValueTuple is Serializable" in the .NET Framework 4.7.1 Runtime and Compiler Features blog post.

Support for read-only references

.NET Framework 4.7.1 adds the System.Runtime.CompilerServices.IsReadOnlyAttribute. This attribute is used by language compilers to mark members that have read-only ref return types or parameters. For more information, see "Compiler -- Support for ReadOnlyReferences" in the .NET Framework 4.7.1 Runtime and Compiler Features blog post. For information on ref return values, see Ref return values and ref locals (C# Guide) and Ref return values (Visual Basic).

Common language runtime (CLR)

Garbage collection performance improvements

Changes to garbage collection (GC) in .NET Framework 4.7.1 improve overall performance, especially for Large Object Heap (LOH) allocations. In .NET Framework 4.7.1, separate locks are used for Small Object Heap (SOH) and LOH allocations, which allows LOH allocations to occur when Background GC (BGC) is sweeping the SOH. As a result, applications that make a large number of LOH allocations should see a reduction in allocation lock contention and improved performance. For more information, see the "Runtime -- GC Performance Improvements" section in the .NET Framework 4.7.1 Runtime and Compiler Features blog post.

Networking

SHA-2 support for Message.HashAlgorithm

In .NET Framework 4.7 and earlier versions, the Message.HashAlgorithm property supported values of HashAlgorithm.Md5 and HashAlgorithm.Sha only. Starting with .NET Framework 4.7.1, HashAlgorithm.Sha256, HashAlgorithm.Sha384, and HashAlgorithm.Sha512 are also supported. Whether this value is actually used depends on MSMQ, since the Message instance itself does no hashing but simply passes on values to MSMQ. For more information, see the "SHA-2 support for Message.HashAlgorithm" section in the .NET Framework 4.7.1 ASP.NET and Configuration features blog post.

ASP.NET

Execution steps in ASP.NET applications

ASP.NET processes requests in a predefined pipeline that includes 23 events. ASP.NET executes each event handler as an execution step. In versions of ASP.NET up to .NET Framework 4.7, ASP.NET can't flow the execution context due to switching between native and managed threads. Instead, ASP.NET selectively flows only the HttpContext. Starting with .NET Framework 4.7.1, the HttpApplication.OnExecuteRequestStep(Action<HttpContextBase,Action>) method also allows modules to restore ambient data. This feature is targeted at libraries concerned with tracing, profiling, diagnostics, or transactions, for example, that care about the execution flow of the application. For more information, see the "ASP.NET Execution Step Feature" in the .NET Framework 4.7.1 ASP.NET and Configuration Features blog post.

ASP.NET HttpCookie parsing

.NET Framework 4.7.1 includes a new method, HttpCookie.TryParse, that provides a standardized way to create an HttpCookie object from a string and accurately assign cookie values such as expiration date and path. For more information, see "ASP.NET HttpCookie parsing" in the .NET Framework 4.7.1 ASP.NET and Configuration Features blog post.

SHA-2 hash options for ASP.NET forms authentication credentials

In .NET Framework 4.7 and earlier versions, ASP.NET allowed developers to store user credentials with hashed passwords in configuration files using either MD5 or SHA1. Starting with .NET Framework 4.7.1, ASP.NET also supports new secure SHA-2 hash options such as SHA256, SHA384, and SHA512. SHA1 remains the default, and a non-default hash algorithm can be defined in the web configuration file. For example:

```
<system.web>
    <authentication mode="Forms">
        <forms loginUrl="~/login.aspx">
          <credentials passwordFormat="SHA512">
            <user name="jdoe"
password="6D003E98EA1C7F04ABF8FCB375388907B7F3EE06F278DB966BE960E7CBBD103DF30CA6D61F7E7FD981B2E4E3A64D43C836A4BEDCA165C33B163E6BCDC538A664" />
          </credentials>
        </forms>
    </authentication>
</system.web>
```

What's new in .NET Framework 4.7

.NET Framework 4.7 includes new features in the following areas:

- Base classes
- Networking
- ASP.NET
- Windows Communication Foundation (WCF)
- Windows Forms
- Windows Presentation Foundation (WPF)

For a list of new APIs added to .NET Framework 4.7, see .NET Framework 4.7 API Changes on GitHub. For a list of feature improvements and bug fixes in .NET Framework 4.7, see .NET Framework 4.7 List of Changes on GitHub. For additional information, see Announcing the .NET Framework 4.7 in the .NET blog.

Base classes

.NET Framework 4.7 improves serialization by the DataContractJsonSerializer:

Enhanced functionality with Elliptic Curve Cryptography (ECC)*

In .NET Framework 4.7, `ImportParameters(ECParameters)` methods were added to the ECDsa and ECDiffieHellman classes to allow for an object to represent an already-established key. An `ExportParameters(Boolean)` method was also added for exporting the key using explicit curve parameters.

.NET Framework 4.7 also adds support for additional curves (including the Brainpool curve suite), and has added predefined definitions for ease-of-creation through the new Create and Create factory methods.

You can see an example of .NET Framework 4.7 cryptography improvements on GitHub.

Better support for control characters by the DataContractJsonSerializer

In .NET Framework 4.7, the DataContractJsonSerializer serializes control characters in conformity with the ECMAScript 6 standard. This behavior is enabled by default for applications that target .NET Framework 4.7, and is an opt-in feature for applications that are running under .NET Framework 4.7 but target a previous version of the .NET Framework. For more information, see Retargeting Changes in the .NET Framework 4.7.

Networking

.NET Framework 4.7 adds the following network-related feature:

Default operating system support for TLS protocols*

The TLS stack, which is used by System.Net.Security.SslStream and up-stack components such as HTTP, FTP, and SMTP, allows developers to use the default TLS protocols supported by the operating system. Developers need no longer hard-code a TLS version.

ASP.NET

In .NET Framework 4.7, ASP.NET includes the following new features:

Object Cache Extensibility

Starting with .NET Framework 4.7, ASP.NET adds a new set of APIs that allow developers to replace the default ASP.NET implementations for in-memory object caching and memory monitoring. Developers can now replace any of the following three components if the ASP.NET implementation is not adequate:

- **Object Cache Store**. By using the new cache providers configuration section, developers can plug in new implementations of an object cache for an ASP.NET application by using the new **ICacheStoreProvider** interface.
- **Memory monitoring**. The default memory monitor in ASP.NET notifies applications when they are running close to the configured private bytes limit for the process, or when the machine is low on total available physical RAM. When these limits are near, notifications are fired. For some applications, notifications are fired too close to the configured limits to allow for useful reactions. Developers can now write their own memory monitors to replace the default by using the ApplicationMonitors.MemoryMonitor property.
- **Memory Limit Reactions**. By default, ASP.NET attempts to trim the object cache and periodically call GC.Collect when the private byte process limit is near. For some applications, the frequency of calls to GC.Collect or the amount of cache that is trimmed are inefficient. Developers can now replace or supplement the default behavior by subscribing **IObserver** implementations to the application's memory monitor.

Windows Communication Foundation (WCF)

Windows Communication Foundation (WCF) adds the following features and changes:

Ability to configure the default message security settings to TLS 1.1 or TLS 1.2

Starting with .NET Framework 4.7, WCF allows you to configure TSL 1.1 or TLS 1.2 in addition to SSL 3.0 and TSL 1.0 as the default message security protocol. This is an opt-in setting; to enable it, you must add the following entry to your application configuration file:

```
<runtime>
   <AppContextSwitchOverrides
value="Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols=false;Switch.System.Net.DontEnableSchUseStrongCrypto=false" />
</runtime>
```

Improved reliability of WCF applications and WCF serialization

WCF includes a number of code changes that eliminate race conditions, thereby improving performance and the reliability of serialization options. These include:

- Better support for mixing asynchronous and synchronous code in calls to **SocketConnection.BeginRead** and **SocketConnection.Read**.
- Improved reliability when aborting a connection with **SharedConnectionListener** and **DuplexChannelBinder**.
- Improved reliability of serialization operations when calling the FormatterServices.GetSerializableMembers(Type) method.
- Improved reliability when removing a waiter by calling the **ChannelSynchronizer.RemoveWaiter** method.

Windows Forms

In .NET Framework 4.7, Windows Forms improves support for high DPI monitors.

High DPI support

Starting with applications that target .NET Framework 4.7, the .NET Framework features high DPI and dynamic DPI support for Windows Forms applications. High DPI support improves the layout and appearance of forms and controls on high DPI monitors. Dynamic DPI changes the layout and appearance of forms and controls when the user changes the DPI or display scale factor of a running application.

High DPI support is an opt-in feature that you configure by defining a <System.Windows.Forms.ConfigurationSection> section in your application configuration file. For more information on adding high DPI support and dynamic DPI support to your Windows Forms application, see High DPI Support in Windows Forms.

Windows Presentation Foundation (WPF)

In .NET Framework 4.7, WPF includes the following enhancements:

Support for a touch/stylus stack based on Windows WM_POINTER messages

You now have the option of using a touch/stylus stack based on WM_POINTER messages instead of the Windows Ink Services Platform (WISP). This is an opt-in feature in the .NET Framework. For more information, see Retargeting Changes in the .NET Framework 4.7.

New implementation for WPF printing APIs

WPF's printing APIs in the System.Printing.PrintQueue class call the Windows Print Document Package API instead of the deprecated XPS Print API. For the impact of this change on application compatibility, see Retargeting Changes in the .NET Framework 4.7.

What's new in .NET Framework 4.6.2

The .NET Framework 4.6.2 includes new features in the following areas:

- ASP.NET
- Character categories
- Cryptography
- SqlClient
- Windows Communication Foundation
- Windows Presentation Foundation (WPF)
- Windows Workflow Foundation (WF)
- ClickOnce
- Converting Windows Forms and WPF apps to UWP apps
- Debugging improvements

For a list of new APIs added to .NET Framework 4.6.2, see .NET Framework 4.6.2 API Changes on GitHub. For a list of feature improvements and bug fixes in .NET Framework 4.6.2, see .NET Framework 4.6.2 List of Changes on GitHub. For additional information, see Announcing .NET Framework 4.6.2 in the .NET blog.

ASP.NET

In the .NET Framework 4.6.2, ASP.NET includes the following enhancements:

Improved support for localized error messages in data annotation validators

Data annotation validators enable you to perform validation by adding one or more attributes to a class property. The attribute's ValidationAttribute.ErrorMessage element defines the text of the error message if validation fails. Starting with the .NET Framework 4.6.2, ASP.NET makes it easy to localize error messages. Error messages will be localized if:

1. The ValidationAttribute.ErrorMessage is provided in the validation attribute.
2. The resource file is stored in the App_LocalResources folder.
3. The name of the localized resources file has the form `DataAnnotation.Localization.{` *name* `}.resx`, where *name* is a culture name in the format *languageCode* `-` *country/regionCode* or *languageCode*.
4. The key name of the resource is the string assigned to the ValidationAttribute.ErrorMessage attribute, and its value is the localized error message.

For example, the following data annotation attribute defines the default culture's error message for an invalid rating.

```
public class RatingInfo
{
   [Required(ErrorMessage = "The rating must be between 1 and 10.")]
   [Display(Name = "Your Rating")]
   public int Rating { get; set; }
}
```

```
Public Class RatingInfo
   <Required(ErrorMessage = "The rating must be between 1 and 10.")>
   <Display(Name = "Your Rating")>
   Public Property Rating As Integer = 1
End Class
```

You can then create a resource file, DataAnnotation.Localization.fr.resx, whose key is the error message string and whose value is the localized error message. The file must be found in the `App.LocalResources` folder. For example, the following is the key and its value in a localized French (fr) language error message:

NAME	VALUE
The rating must be between 1 and 10.	La note doit être comprise entre 1 et 10.

In addition, data annotation localization is extensible. Developers can plug in their own string localizer provider by implementing the IStringLocalizerProvider interface to store localization string somewhere other than in a resource file.

Async support with session-state store providers

ASP.NET now allows task-returning methods to be used with session-state store providers, thereby allowing ASP.NET apps to get the scalability benefits of async. To supports asynchronous operations with session state store providers, ASP.NET includes a new interface, System.Web.SessionState.ISessionStateModule, which inherits from IHttpModule and allows developers to implement their own session-state module and async session store providers. The interface is defined as follows:

```
public interface ISessionStateModule : IHttpModule {
    void ReleaseSessionState(HttpContext context);
    Task ReleaseSessionStateAsync(HttpContext context);
}
```

```
Public Interface ISessionStateModule : Inherits IHttpModule
   Sub ReleaseSessionState(context As HttpContext)
   Function ReleaseSessionStateAsync(context As HttpContext) As Task
End Interface
```

In addition, the SessionStateUtility class includes two new methods, IsSessionStateReadOnly and IsSessionStateRequired, that can be used to support asynchronous operations.

Async support for output-cache providers

Starting with the .NET Framework 4.6.2, task-returning methods can be used with output-cache providers to provide the scalability benefits of async. Providers that implement these methods reduce thread-blocking on a web server and improve the scalability of an ASP.NET service.

The following APIs have been added to support asynchronous output-cache providers:

- The System.Web.Caching.OutputCacheProviderAsync class, which inherits from System.Web.Caching.OutputCacheProvider and allows developers to implement an asynchronous output-cache provider.
- The OutputCacheUtility class, which provides helper methods for configuring the output cache.
- 18 new methods in the System.Web.HttpCachePolicy class. These include GetCacheability, GetCacheExtensions, GetETag, GetETagFromFileDependencies, GetMaxAge, GetMaxAge, GetNoStore, GetNoTransforms, GetOmitVaryStar, GetProxyMaxAge, GetRevalidation, GetUtcLastModified, GetVaryByCustom, HasSlidingExpiration, and IsValidUntilExpires.
- 2 new methods in the System.Web.HttpCacheVaryByContentEncodings class: GetContentEncodings and SetContentEncodings.
- 2 new methods in the System.Web.HttpCacheVaryByHeaders class: GetHeaders and SetHeaders.
- 2 new methods in the System.Web.HttpCacheVaryByParams class: GetParams and SetParams.

- In the System.Web.Caching.AggregateCacheDependency class, the GetFileDependencies method.
- In the CacheDependency, the GetFileDependencies method.

Character categories

Characters in the .NET Framework 4.6.2 are classified based on the Unicode Standard, Version 8.0.0. In .NET Framework 4.6 and .NET Framework 4.6.1, characters were classified based on Unicode 6.3 character categories.

Support for Unicode 8.0 is limited to the classification of characters by the CharUnicodeInfo class and to types and methods that rely on it. These include the StringInfo class, the overloaded Char.GetUnicodeCategory method, and the character classes recognized by the .NET Framework regular expression engine. Character and string comparison and sorting is unaffected by this change and continues to rely on the underlying operating system or, on Windows 7 systems, on character data provided by the .NET Framework.

For changes in character categories from Unicode 6.0 to Unicode 7.0, see The Unicode Standard, Version 7.0.0 at The Unicode Consortium website. For changes from Unicode 7.0 to Unicode 8.0, see The Unicode Standard, Version 8.0.0 at The Unicode Consortium website.

Cryptography

Support for X509 certificates containing FIPS 186-3 DSA

The .NET Framework 4.6.2 adds support for DSA (Digital Signature Algorithm) X509 certificates whose keys exceed the FIPS 186-2 1024-bit limit.

In addition to supporting the larger key sizes of FIPS 186-3, the .NET Framework 4.6.2 allows computing signatures with the SHA-2 family of hash algorithms (SHA256, SHA384, and SHA512). FIPS 186-3 support is provided by the new System.Security.Cryptography.DSACng class.

In keeping with recent changes to the RSA class in .NET Framework 4.6 and the ECDsa class in .NET Framework 4.6.1, the DSA abstract base class in .NET Framework 4.6.2 has additional methods to allow callers to use this functionality without casting. You can call the DSACertificateExtensions.GetDSAPrivateKey extension method to sign data, as the following example shows.

```
public static byte[] SignDataDsaSha384(byte[] data, X509Certificate2 cert)
{
    using (DSA dsa = cert.GetDSAPrivateKey())
    {
        return dsa.SignData(data, HashAlgorithmName.SHA384);
    }
}
```

```
Public Shared Function SignDataDsaSha384(data As Byte(), cert As X509Certificate2) As Byte()
    Using DSA As DSA = cert.GetDSAPrivateKey()
        Return DSA.SignData(data, HashAlgorithmName.SHA384)
    End Using
End Function
```

And you can call the DSACertificateExtensions.GetDSAPublicKey extension method to verify signed data, as the following example shows.

```
public static bool VerifyDataDsaSha384(byte[] data, byte[] signature, X509Certificate2 cert)
{
    using (DSA dsa = cert.GetDSAPublicKey())
    {
        return dsa.VerifyData(data, signature, HashAlgorithmName.SHA384);
    }
}
```

```
 Public Shared Function VerifyDataDsaSha384(data As Byte(), signature As Byte(), cert As X509Certificate2) As Boolean
    Using dsa As DSA = cert.GetDSAPublicKey()
        Return dsa.VerifyData(data, signature, HashAlgorithmName.SHA384)
    End Using
End Function
```

Increased clarity for inputs to ECDiffieHellman key derivation routines

.NET Framework 3.5 added support for Elliptic Curve Diffie-Hellman Key Agreement with three different Key Derivation Function (KDF) routines. The inputs to the routines, and the routines themselves, were configured via properties on the ECDiffieHellmanCng object. But since not every routine read every input property, there was ample room for confusion on the past of the developer.

To address this in the .NET Framework 4.6.2, the following three methods have been added to the ECDiffieHellman base class to more clearly represent these KDF routines and their inputs:

ECDIFFIEHELLMAN METHOD	DESCRIPTION
DeriveKeyFromHash(ECDiffieHellmanPublicKey, HashAlgorithmName, Byte[], Byte[])	Derives key material using the formula HASH(secretPrepend || *x* || secretAppend) HASH(secretPrepend OrElse *x* OrElse secretAppend) where *x* is the computed result of the EC Diffie-Hellman algorithm.

ECDIFFIEHELLMAN METHOD	DESCRIPTION
DeriveKeyFromHmac(ECDiffieHellmanPublicKey, HashAlgorithmName, Byte[], Byte[], Byte[])	Derives key material using the formula HMAC(hmacKey, secretPrepend || *x* || secretAppend) HMAC(hmacKey, secretPrepend OrElse *x* OrElse secretAppend) where *x* is the computed result of the EC Diffie-Hellman algorithm.
DeriveKeyTls(ECDiffieHellmanPublicKey, Byte[], Byte[])	Derives key material using the TLS pseudo-random function (PRF) derivation algorithm.

Support for persisted-key symmetric encryption

The Windows cryptography library (CNG) added support for storing persisted symmetric keys and using hardware-stored symmetric keys, and the .NET Framework 4.6.2 made it possible for developers to make use of this feature. Since the notion of key names and key providers is implementation-specific, using this feature requires utilizing the constructor of the concrete implementation types instead of the preferred factory approach (such as calling `Aes.Create`).

Persisted-key symmetric encryption support exists for the AES (AesCng) and 3DES (TripleDESCng) algorithms. For example:

```
public static byte[] EncryptDataWithPersistedKey(byte[] data, byte[] iv)
{
    using (Aes aes = new AesCng("AesDemoKey", CngProvider.MicrosoftSoftwareKeyStorageProvider))
    {
        aes.IV = iv;

        // Using the zero-argument overload is required to make use of the persisted key
        using (ICryptoTransform encryptor = aes.CreateEncryptor())
        {
            if (!encryptor.CanTransformMultipleBlocks)
            {
                throw new InvalidOperationException("This is a sample, this case wasn't handled...");
            }

            return encryptor.TransformFinalBlock(data, 0, data.Length);
        }
    }
}
```

```
Public Shared Function EncryptDataWithPersistedKey(data As Byte(), iv As Byte()) As Byte()
    Using Aes As Aes = New AesCng("AesDemoKey", CngProvider.MicrosoftSoftwareKeyStorageProvider)
        Aes.IV = iv

        ' Using the zero-argument overload Is required to make use of the persisted key
        Using encryptor As ICryptoTransform = Aes.CreateEncryptor()
            If Not encryptor.CanTransformMultipleBlocks Then
                Throw New InvalidOperationException("This is a sample, this case wasn't handled...")
            End If
            Return encryptor.TransformFinalBlock(data, 0, data.Length)
        End Using
    End Using
End Function
```

SignedXml support for SHA-2 hashing

The .NET Framework 4.6.2 adds support to the SignedXml class for RSA-SHA256, RSA-SHA384, and RSA-SHA512 PKCS#1 signature methods, and SHA256, SHA384, and SHA512 reference digest algorithms.

The URI constants are all exposed on SignedXml:

SIGNEDXML FIELD	CONSTANT
XmlDsigSHA256Url	"http://www.w3.org/2001/04/xmlenc#sha256"
XmlDsigRSASHA256Url	"http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"
XmlDsigSHA384Url	"http://www.w3.org/2001/04/xmldsig-more#sha384"
XmlDsigRSASHA384Url	"http://www.w3.org/2001/04/xmldsig-more#rsa-sha384"
XmlDsigSHA512Url	"http://www.w3.org/2001/04/xmlenc#sha512"
XmlDsigRSASHA512Url	"http://www.w3.org/2001/04/xmldsig-more#rsa-sha512"

Any programs that have registered a custom SignatureDescription handler into CryptoConfig to add support for these algorithms will continue to function as they did in the past, but since there are now platform defaults, the CryptoConfig registration is no longer necessary.

SqlClient

.NET Framework Data Provider for SQL Server (System.Data.SqlClient) includes the following new features in the .NET Framework 4.6.2:

Connection pooling and timeouts with Azure SQL databases

When connection pooling is enabled and a timeout or other login error occurs, an exception is cached, and the cached exception is thrown on any subsequent connection attempt for the next 5 seconds to 1 minute. For more details, see SQL Server Connection Pooling (ADO.NET).

This behavior is not desirable when connecting to Azure SQL Databases, since connection attempts can fail with transient errors that are typically recovered quickly. To better optimize the connection retry experience, the connection pool blocking period behavior is removed when connections to Azure SQL Databases fail.

The addition of the new `PoolBlockingPeriod` keyword lets you to select the blocking period best suited for your app. Values include:

Auto

The connection pool blocking period for an application that connects to an Azure SQL Database is disabled, and the connection pool blocking period for an application that connects to any other SQL Server instance is enabled. This is the default value. If the Server endpoint name ends with any of the following, they are considered Azure SQL Databases:

- .database.windows.net
- .database.chinacloudapi.cn
- .database.usgovcloudapi.net
- .database.cloudapi.de

AlwaysBlock

The connection pool blocking period is always enabled.

NeverBlock

The connection pool blocking period is always disabled.

Enhancements for Always Encrypted

SQLClient introduces two enhancements for Always Encrypted:

- To improve performance of parameterized queries against encrypted database columns, encryption metadata for query parameters is now cached. With the SqlConnection.ColumnEncryptionQueryMetadataCacheEnabled property set to `true` (which is the default value), if the same query is called multiple times, the client retrieves parameter metadata from the server only once.
- Column encryption key entries in the key cache are now evicted after a configurable time interval, set using the SqlConnection.ColumnEncryptionKeyCacheTtl property.

Windows Communication Foundation

In the .NET Framework 4.6.2, Windows Communication Foundation has been enhanced in the following areas:

WCF transport security support for certificates stored using CNG

WCF transport security supports certificates stored using the Windows cryptography library (CNG). In the .NET Framework 4.6.2, this support is limited to using certificates with a public key that has an exponent no more than 32 bits in length. When an application targets the .NET Framework 4.6.2, this feature is on by default.

For applications that target the .NET Framework 4.6.1 and earlier but are running on the .NET Framework 4.6.2, this feature can be enabled by adding the following line to the <runtime> section of the app.config or web.config file.

```
<AppContextSwitchOverrides
    value="Switch.System.ServiceModel.DisableCngCertificates=false"
/>
```

This can also be done programmatically with code like the following:

```
private const string DisableCngCertificates = @"Switch.System.ServiceModel.DisableCngCertificates";
AppContext.SetSwitch(disableCngCertificates, false);
```

```
Const DisableCngCertificates As String = "Switch.System.ServiceModel.DisableCngCertificates"
AppContext.SetSwitch(disableCngCertificates, False)
```

Better support for multiple daylight saving time adjustment rules by the DataContractJsonSerializer class

Customers can use an application configuration setting to determine whether the DataContractJsonSerializer class supports multiple adjustment rules for a single time zone. This is an opt-in feature. To enable it, add the following setting to your app.config file:

```
<runtime>
     <AppContextSwitchOverrides value="Switch.System.Runtime.Serialization.DoNotUseTimeZoneInfo=false" />
</runtime>
```

When this feature is enabled, a DataContractJsonSerializer object uses the TimeZoneInfo type instead of the TimeZone type to deserialize date and time data. TimeZoneInfo supports multiple adjustment rules, which makes it possible to work with historic time zone data; TimeZone does not.

For more information on the TimeZoneInfo structure and time zone adjustments, see Time Zone Overview.

NetNamedPipeBinding best match

WCF has a new app setting that can be set on client applications to ensure they always connect to the service listening on the URI that best matches the one that they request. With this app setting set to `false` (the default), it is possible for clients using NetNamedPipeBinding to attempt to connect to a service listening on a URI that is a substring of the requested URI.

For example, a client tries to connect to a service listening at `net.pipe://localhost/Service1`, but a different service on that machine running with administrator privilege is listening at `net.pipe://localhost`. With this app setting set to `false`, the client would attempt to connect to the wrong service. After setting the app setting to `true`, the client will always connect to the best matching service.

> **NOTE**
> Clients using NetNamedPipeBinding find services based on the service's base address (if it exists) rather than the full endpoint address. To ensure this setting always works the service should use a unique base address.

To enable this change, add the following app setting to your client application's App.config or Web.config file:

```
<configuration>
    <appSettings>
        <add key="wcf:useBestMatchNamedPipeUri" value="true" />
    </appSettings>
</configuration>
```

SSL 3.0 is not a default protocol

When using NetTcp with transport security and a credential type of certificate, SSL 3.0 is no longer a default protocol used for negotiating a secure connection. In most cases, there should be no impact to existing apps, because TLS 1.0 is included in the protocol list for NetTcp. All existing clients should be able to negotiate a connection using at least TLS 1.0. If Ssl3 is required, use one of the following configuration mechanisms to add it to the list of negotiated protocols.

- The SslStreamSecurityBindingElement.SslProtocols property
- The TcpTransportSecurity.SslProtocols property
- The <transport> section of the <netTcpBinding> section
- The <sslStreamSecurity> section of the <customBinding> section

Windows Presentation Foundation (WPF)

In the .NET Framework 4.6.2, Windows Presentation Foundation has been enhanced in the following areas:

Group sorting

An application that uses a CollectionView object to group data can now explicitly declare how to sort the groups. Explicit sorting addresses the problem of non-intuitive ordering that occurs when an app dynamically adds or removes groups, or when it changes the value of item properties involved in grouping. It can also improve the performance of the group creation process by moving comparisons of the grouping properties from the sort of the full collection to the sort of the groups.

To support group sorting, the new GroupDescription.SortDescriptions and GroupDescription.CustomSort properties describe how to sort the collection of groups produced by the GroupDescription object. This is analogous to the way the identically named ListCollectionView properties describe how to sort the data items.

Two new static properties of the PropertyGroupDescription class, CompareNameAscending and CompareNameDescending, can be used for the most common cases.

For example, the following XAML groups data by age, sort the age groups in ascending order, and group the items within each age group by last name.

```
<GroupDescriptions>
     <PropertyGroupDescription
         PropertyName="Age"
         CustomSort=
              "{x:Static PropertyGroupDescription.CompareNamesAscending}"/>
     </PropertyGroupDescription>
</GroupDescriptions>

<SortDescriptions>
     <SortDescription PropertyName="LastName"/>
</SortDescriptions>
```

Soft keyboard support

Soft Keyboard support enables focus tracking in a WPF applications by automatically invoking and dismissing the new Soft Keyboard in Windows 10 when the touch input is received by a control that can take textual input.

In previous versions of the .NET Framework, WPF applications cannot opt into the focus tracking without disabling WPF pen/touch gesture support. As a result, WPF applications must choose between full WPF touch support or rely on Windows mouse promotion.

Per-monitor DPI

To support the recent proliferation of high-DPI and hybrid-DPI environments for WPF apps, WPF in the .NET Framework 4.6.2 enables per-monitor awareness. See the samples and developer guide on GitHub for more information about how to enable your WPF app to become per-monitor DPI aware.

In previous versions of the .NET Framework, WPF apps are system-DPI aware. In other words, the application's UI is scaled by the OS as appropriate, depending on the DPI of the monitor on which the app is rendered. ,

For apps running under the .NET Framework 4.6.2, you can disable per-monitor DPI changes in WPF apps by adding a configuration statement to the <runtime> section of your application configuration file, as follows:

```
<runtime>
   <AppContextSwitchOverrides value="Switch.System.Windows.DoNotScaleForDpiChanges=false"/>
</runtime>
```

Windows Workflow Foundation (WF)

In the .NET Framework 4.6.2, Windows Workflow Foundation has been enhanced in the following area:

Support for C# expressions and IntelliSense in the Re-hosted WF Designer

Starting with the .NET Framework 4.5, WF supports C# expressions in both the Visual Studio Designer and in code workflows. The Re-hosted Workflow Designer is a key feature of WF that allows for the Workflow Designer to be in an application outside Visual Studio (for example, in WPF). Windows Workflow Foundation provides the ability to support C# expressions and IntelliSense in the Re-hosted Workflow Designer. For more information, see the Windows Workflow Foundation blog.

`Availability of IntelliSense when a customer rebuilds a workflow project from Visual Studio` In versions of the .NET Framework prior to the .NET Framework 4.6.2, WF Designer IntelliSense is broken when a customer rebuilds a workflow project from Visual Studio. While the project build is successful, the workflow types are not found on the designer, and warnings from IntelliSense for the missing workflow types appear in the **Error List** window. The .NET Framework 4.6.2 addresses this issue and makes IntelliSense available.

Workflow V1 applications with Workflow Tracking on now run under FIPS-mode

Machines with FIPS Compliance Mode enabled can now successfully run a workflow Version 1-style application with Workflow tracking on. To enable this scenario, you must make the following change to your app.config file:

```
<add key="microsoft:WorkflowRuntime:FIPSRequired" value="true" />
```

If this scenario is not enabled, running the application continues to generate an exception with the message, "This implementation is not part of the Windows Platform FIPS validated cryptographic algorithms."

Workflow Improvements when using Dynamic Update with Visual Studio Workflow Designer

The Workflow Designer, FlowChart Activity Designer, and other Workflow Activity Designers now successfully load and display workflows that have been saved after calling the DynamicUpdateServices.PrepareForUpdate method. In versions of the .NET Framework before .NET Framework 4.6.2, loading a XAML file in Visual Studio for a workflow that has been saved after calling DynamicUpdateServices.PrepareForUpdate can result in the following issues:

- The Workflow Designer can't load the XAML file correctly (when the ViewStateData.Id is at the end of the line).
- Flowchart Activity Designer or other Workflow Activity Designers may display all objects in their default locations as opposed to attached property values.

ClickOnce

ClickOnce has been updated to support TLS 1.1 and TLS 1.2 in addition to the 1.0 protocol, which it already supports. ClickOnce automatically detects which protocol is required; no extra steps within the ClickOnce application are required to enable TLS 1.1 and 1.2 support.

Converting Windows Forms and WPF apps to UWP apps

Windows now offers capabilities to bring existing Windows desktop apps, including WPF and Windows Forms apps, to the Universal Windows Platform (UWP). This technology acts as a bridge by enabling you to gradually migrate your existing code base to UWP, thereby bringing your app to all Windows 10 devices.

Converted desktop apps gain an app identity similar to the app identity of UWP apps, which makes UWP APIs accessible to enable features such as Live Tiles and notifications. The app continues to behave as before and runs as a full trust app. Once the app is converted, an app container process can be added to the existing full trust process to add an adaptive user interface. When all functionality is moved to the app container process, the full trust process can be removed and the new UWP app can be made available to all Windows 10 devices.

Debugging improvements

The *unmanaged debugging API* has been enhanced in the .NET Framework 4.6.2 to perform additional analysis when a NullReferenceException is thrown so that it is possible to determine which variable in a single line of source code is `null`. To support this scenario, the following APIs have been added to the unmanaged debugging API.

- The ICorDebugCode4, ICorDebugVariableHome, and ICorDebugVariableHomeEnum interfaces, which expose the native homes of managed variables. This enables debuggers to do some code flow analysis when a NullReferenceException occurs and to work backwards to determine the managed variable that corresponds to the native location that was `null`.
- The ICorDebugType2::GetTypeID method provides a mapping for ICorDebugType to COR_TYPEID, which allows the debugger to obtain a COR_TYPEID without an instance of the ICorDebugType. Existing APIs on COR_TYPEID can then be used to determine the class layout of the type.

What's new in .NET Framework 4.6.1

The .NET Framework 4.6.1 includes new features in the following areas:

- Cryptography
- ADO.NET
- Windows Presentation Foundation (WPF)
- Windows Workflow Foundation
- Profiling
- NGen

For more information on the .NET Framework 4.6.1, see the following topics:

- .NET Framework 4.6.1 list of changes
- Application Compatibility in 4.6.1
- .NET Framework API diff (on GitHub)

Cryptography: Support for X509 certificates containing ECDSA

.NET Framework 4.6 added RSACng support for X509 certificates. The .NET Framework 4.6.1 adds support for ECDSA (Elliptic Curve Digital Signature Algorithm) X509 certificates.

ECDSA offers better performance and is a more secure cryptography algorithm than RSA, providing an excellent choice where Transport Layer Security (TLS) performance and scalability is a concern. The .NET Framework implementation wraps calls into existing Windows functionality.

The following example code shows how easy it is to generate a signature for a byte stream by using the new support for ECDSA X509 certificates included in the .NET Framework 4.6.1.

```
using System;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;

public class Net461Code
{
    public static byte[] SignECDsaSha512(byte[] data, X509Certificate2 cert)
    {
        using (ECDsa privateKey = cert.GetECDsaPrivateKey())
        {
            return privateKey.SignData(data, HashAlgorithmName.SHA512);
        }
    }

    public static byte[] SignECDsaSha512(byte[] data, ECDsa privateKey)
    {
        return privateKey.SignData(data, HashAlgorithmName.SHA512);
    }
}
```

```
Imports System.Security.Cryptography
Imports System.Security.Cryptography.X509Certificates

Public Class Net461Code
    Public Shared Function SignECDsaSha512(data As Byte(), cert As X509Certificate2) As Byte()
        Using privateKey As ECDsa = cert.GetECDsaPrivateKey()
            Return privateKey.SignData(data, HashAlgorithmName.SHA512)
        End Using
    End Function

    Public Shared Function SignECDsaSha512(data As Byte, privateKey As ECDsa) As Byte()
        Return privateKey.SignData(data, HashAlgorithmName.SHA512)
    End Function
End Class
```

This offers a marked contrast to the code needed to generate a signature in .NET Framework 4.6.

```
using System;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;

public class Net46Code
{
    public static byte[] SignECDsaSha512(byte[] data, X509Certificate2 cert)
    {
        // This would require using cert.Handle and a series of p/invokes to get at the
        // underlying key, then passing that to a CngKey object, and passing that to
        // new ECDsa(CngKey).  It's a lot of work.
        throw new Exception("That's a lot of work...");
    }

    public static byte[] SignECDsaSha512(byte[] data, ECDsa privateKey)
    {
        // This way works, but SignData probably better matches what you want.
        using (SHA512 hasher = SHA512.Create())
        {
            byte[] signature1 = privateKey.SignHash(hasher.ComputeHash(data));
        }

        // This might not be the ECDsa you got!
        ECDsaCng ecDsaCng = (ECDsaCng)privateKey;
        ecDsaCng.HashAlgorithm = CngAlgorithm.Sha512;
        return ecDsaCng.SignData(data);
    }
}
```

```
Imports System.Security.Cryptography
Imports System.Security.Cryptography.X509Certificates

Public Class Net46Code
    Public Shared Function SignECDsaSha512(data As Byte(), cert As X509Certificate2) As Byte()
        ' This would require using cert.Handle and a series of p/invokes to get at the
        ' underlying key, then passing that to a CngKey object, and passing that to
        ' new ECDsa(CngKey).  It's a lot of work.
        Throw New Exception("That's a lot of work...")
    End Function

    Public Shared Function SignECDsaSha512(data As Byte(), privateKey As ECDsa) As Byte()
        ' This way works, but SignData probably better matches what you want.
        Using hasher As SHA512 = SHA512.Create()
            Dim signature1 As Byte() = privateKey.SignHash(hasher.ComputeHash(data))
        End Using

        ' This might not be the ECDsa you got!
        Dim ecDsaCng As ECDsaCng = CType(privateKey, ECDsaCng)
        ecDsaCng.HashAlgorithm = CngAlgorithm.Sha512
        Return ecDsaCng.SignData(data)
    End Function
End Class
```

ADO.NET

The following have been added to ADO.NET:

Always Encrypted support for hardware protected keys

ADO.NET now supports storing Always Encrypted column master keys natively in Hardware Security Modules (HSMs). With this support, customers can leverage asymmetric keys stored in HSMs without having to write custom column master key store providers and registering them in applications.

Customers need to install the HSM vendor-provided CSP provider or CNG key store providers on the app servers or client computers in order to access Always Encrypted data protected with column master keys stored in a HSM.

Improved MultiSubnetFailover connection behavior for AlwaysOn

SqlClient now automatically provides faster connections to an AlwaysOn Availability Group (AG). It transparently detects whether your application is connecting to an AlwaysOn availability group (AG) on a different subnet and quickly discovers the current active server and provides a connection to the server. Prior to this release, an application had to set the connection string to include `"MultisubnetFailover=true"` to indicate that it was connecting to an AlwaysOn Availability Group. Without setting the connection keyword to `true`, an application might experience a timeout while connecting to an AlwaysOn Availability Group. With this release, an application does *not* need to set MultiSubnetFailover to `true` anymore. For more information about SqlClient support for Always On Availability Groups, see SqlClient Support for High Availability, Disaster Recovery.

Windows Presentation Foundation (WPF)

Windows Presentation Foundation includes a number of improvements and changes.

Improved performance

The delay in firing touch events has been fixed in the .NET Framework 4.6.1. In addition, typing in a RichTextBox control no longer ties up the render thread during fast input.

Spell checking improvements

The spell checker in WPF has been updated on Windows 8.1 and later versions to leverage operating system support for spell-checking additional languages. There is no change in functionality on Windows versions prior to Windows 8.1.

As in previous versions of the .NET Framework, the language for a TextBox control ora RichTextBox block is detected by looking for information in the following order:

- `xml:lang`, if it is present.
- Current input language.
- Current thread culture.

For additional information on language support in WPF, see the WPF blog post on .NET Framework 4.6.1 features.

Additional support for per-user custom dictionaries

In .NET Framework 4.6.1, WPF recognizes custom dictionaries that are registered globally. This capability is available in addition to the ability to register them per-control.

In previous versions of WPF, custom dictionaries did not recognize Excluded Words and AutoCorrect lists. They are supported on Windows 8.1 and Windows 10 through the use of files that can be placed under the `%AppData%\Microsoft\Spelling\<language tag>` directory. The following rules apply to these files:

- The files should have extensions of .dic (for added words), .exc (for excluded words), or .acl (for AutoCorrect).
- The files should be UTF-16 LE plaintext that starts with the Byte Order Mark (BOM).
- Each line should consist of a word (in the added and excluded word lists), or an autocorrect pair with the words separated by a vertical bar ("|") (in the AutoCorrect word list).
- These files are considered read-only and are not modified by the system.

> **NOTE**
> These new file-formats are not directly supported by the WPF spell checking APIs, and the custom dictionaries supplied to WPF in applications should continue to use .lex files.

Samples

There are a number of WPF samples on the Microsoft/WPF-Samples GitHub repository. Help us improve our samples by sending us a pull-request or opening a GitHub issue.

DirectX extensions

WPF includes a NuGet package that provides new implementations of D3DImage that make it easy for you to interoperate with DX10 and Dx11 content. The code for this package has been open sourced and is available on GitHub.

Windows Workflow Foundation: Transactions

The Transaction.EnlistPromotableSinglePhase method can now use a distributed transaction manager other than MSDTC to promote the transaction. You do this by specifying a GUID transaction promoter identifier to the new Transaction.EnlistPromotableSinglePhase(IPromotableSinglePhaseNotification, Guid) overload . If this operation is successful, there are limitations placed on the capabilities of the transaction. Once a non-MSDTC transaction promoter is enlisted, the following methods throw a TransactionPromotionException because these methods require promotion to MSDTC:

- Transaction.EnlistDurable
- TransactionInterop.GetDtcTransaction
- TransactionInterop.GetExportCookie
- TransactionInterop.GetTransmitterPropagationToken

Once a non-MSDTC transaction promoter is enlisted, it must be used for future durable enlistments by using protocols that it defines. The Guid of the transaction promoter can be obtained by using the PromoterType property. When the transaction promotes, the transaction promoter provides a Byte array that represents the promoted token. An application can obtain the promoted token for a non-MSDTC promoted transaction with the GetPromotedToken method.

Users of the new Transaction.EnlistPromotableSinglePhase(IPromotableSinglePhaseNotification, Guid) overload must follow a specific call sequence in order for the promotion operation to complete successfully. These rules are documented in the method's documentation.

Profiling

The unmanaged profiling API has been enhanced as follows:

- Better support for accessing PDBs in the ICorProfilerInfo7 interface.

  In ASP.NET Core, it is becoming much more common for assemblies to be compiled in-memory by Roslyn. For developers making profiling tools, this means that PDBs that historically were serialized on disk may no longer be present. Profiler tools often use PDBs to map code back to source lines for tasks such as code coverage or line-by-line performance analysis. The ICorProfilerInfo7 interface now includes two new methods, ICorProfilerInfo7::GetInMemorySymbolsLength and ICorProfilerInfo7::ReadInMemorySymbols, to provide these profiler tools with access to the in-memory PDB data, By using the new APIs, a profiler can obtain the contents of an in-memory PDB as a byte array and then process it or serialize it to disk.

- Better instrumentation with the ICorProfiler interface.

  Profilers that are using the `ICorProfiler` APIs ReJit functionality for dynamic instrumentation can now modify some metadata. Previously such tools could instrument IL at any time, but metadata could only be modified at module load time. Because IL refers to metadata, this limited the kinds of instrumentation that could be done. We have lifted some of those limits by adding the ICorProfilerInfo7::ApplyMetaData method to support a subset of metadata edits after the module loads, in particular by adding new `AssemblyRef`, `TypeRef`, `TypeSpec`, `MemberRef`, `MemberSpec`, and `UserString` records. This change makes a much broader range of on-the-fly instrumentation possible.

Native Image Generator (NGEN) PDBs

Cross-machine event tracing allows customers to profile a program on Machine A and look at the profiling data with source line mapping on Machine B. Using previous versions of the .NET Framework, the user would copy all the modules and native images from the profiled machine to the analysis machine that contains the IL PDB to create the source-to-native mapping. While this process may work well when the files are relatively small, such as for phone applications, the files can be very large on desktop systems and require significant time to copy.

With Ngen PDBs, NGen can create a PDB that contains the IL-to-native mapping without a dependency on the IL PDB. In our cross-machine event tracing scenario, all that is needed is to copy the native image PDB that is generated by Machine A to Machine B and to use Debug Interface Access APIs to read the IL PDB's source-to-IL mapping and the native image PDB's IL-to-native mapping. Combining both mappings provides a source-to-native mapping. Since the native image PDB is much smaller than all the modules and native images, the process of copying from Machine A to Machine B is much faster.

What's new in .NET 2015

.NET 2015 introduces the .NET Framework 4.6 and .NET Core. Some new features apply to both, and other features are specific to .NET Framework 4.6 or .NET Core.

- **ASP.NET Core**

  .NET 2015 includes ASP.NET Core, which is a lean .NET implementation for building modern cloud-based apps. ASP.NET Core is modular so you can include only those features that are needed in your application. It can be hosted on IIS or self-hosted in a custom process, and you can run apps with different versions of the .NET Framework on the same server. It includes a new environment configuration system that is designed for cloud deployment.

  MVC, Web API, and Web Pages are unified into a single framework called MVC 6. You build ASP.NET Core apps through tools in Visual Studio 2015 or later. Your existing applications will work on the new .NET Framework; however to build an app that uses MVC 6 or SignalR 3, you must use the project system in Visual Studio 2015 or later.

  For information, see ASP.NET Core.

- **ASP.NET Updates**

  - **Task-based API for Asynchronous Response Flushing**

    ASP.NET now provides a simple task-based API for asynchronous response flushing, HttpResponse.FlushAsync, that allows responses to be flushed asynchronously by using your language's `async/await` support.

  - **Model binding supports task-returning methods**

    In the .NET Framework 4.5, ASP.NET added the Model Binding feature that enabled an extensible, code-focused approach to CRUD-based data operations in Web Forms pages and user controls. The Model Binding system now supports Task-returning model binding methods. This feature allows Web Forms developers to get the scalability benefits of async with the ease of the data-binding system when using newer versions of ORMs, including the Entity Framework.

    Async model binding is controlled by the `aspnet:EnableAsyncModelBinding` configuration setting.

```
<appSettings>
    <add key=" aspnet:EnableAsyncModelBinding" value="true|false" />
</appSettings>
```

On apps the target the .NET Framework 4.6, it defaults to `true`. On apps running on the .NET Framework 4.6 that target an earlier version of the .NET Framework, it is `false` by default. It can be enabled by setting the configuration setting to `true`.

- **HTTP/2 Support (Windows 10)**

  HTTP/2 is a new version of the HTTP protocol that provides much better connection utilization (fewer round-trips between client and server), resulting in lower latency web page loading for users. Web pages (as opposed to services) benefit the most from HTTP/2, since the protocol optimizes for multiple artifacts being requested as part of a single experience. HTTP/2 support has been added to ASP.NET in .NET Framework 4.6. Because networking functionality exists at multiple layers, new features were required in Windows, in IIS, and in ASP.NET to enable HTTP/2. You must be running on Windows 10 to use HTTP/2 with ASP.NET.

  HTTP/2 is also supported and on by default for Windows 10 Universal Windows Platform (UWP) apps that use the System.Net.Http.HttpClient API.

  In order to provide a way to use the PUSH_PROMISE feature in ASP.NET applications, a new method with two overloads, PushPromise(String) and PushPromise(String, String, NameValueCollection), has been added to the HttpResponse class.

  > **NOTE**
  > While ASP.NET Core supports HTTP/2, support for the PUSH PROMISE feature has not yet been added.

  The browser and the web server (IIS on Windows) do all the work. You don't have to do any heavy-lifting for your users.

  Most of the major browsers support HTTP/2, so it's likely that your users will benefit from HTTP/2 support if your server supports it.

- **Support for the Token Binding Protocol**

  Microsoft and Google have been collaborating on a new approach to authentication, called the Token Binding Protocol. The premise is that authentication tokens (in your browser cache) can be stolen and used by criminals to access otherwise secure resources (e.g. your bank account) without requiring your password or any other privileged knowledge. The new protocol aims to mitigate this problem.

  The Token Binding Protocol will be implemented in Windows 10 as a browser feature. ASP.NET apps will participate in the protocol, so that authentication tokens are validated to be legitimate. The client and the server implementations establish the end-to-end protection specified by the protocol.

- **Randomized string hash algorithms**

  .NET Framework 4.5 introduced a randomized string hash algorithm. However, it was not supported by ASP.NET because of some ASP.NET features depended on a stable hash code. In .NET Framework 4.6, randomized string hash algorithms are now supported. To enable this feature, use the `aspnet:UseRandomizedStringHashAlgorithm` config setting.

  ```
  <appSettings>
      <add key="aspnet:UseRandomizedStringHashAlgorithm" value="true|false" />
  </appSettings>
  ```

- **ADO.NET**

  ADO .NET now supports the Always Encrypted feature available in SQL Server 2016 Community Technology Preview 2 (CTP2). With Always Encrypted, SQL Server can perform operations on encrypted data, and best of all the encryption key resides with the application inside the customer's trusted environment and not on the server. Always Encrypted secures customer data so DBAs do not have access to plain text data. Encryption and decryption of data happens transparently at the driver level, minimizing changes that have to be made to existing applications. For details, see Always Encrypted (Database Engine) and Always Encrypted (client development).

- **64-bit JIT Compiler for managed code**

  .NET Framework 4.6 features a new version of the 64-bit JIT compiler (originally code-named RyuJIT). The new 64-bit compiler provides significant performance improvements over the older 64-bit JIT compiler. The new 64-bit compiler is enabled for 64-bit processes running on top of .NET Framework 4.6. Your app will run in a 64-bit process if it is compiled as 64-bit or AnyCPU and is running on a 64-bit operating system. While care has been taken to make the transition to the new compiler as transparent as possible, changes in behavior are possible. We would like to hear directly about any issues encountered when using the new JIT compiler. Please contact us through Microsoft Connect if you encounter an issue that may be related to the new 64-bit JIT compiler.

  The new 64-bit JIT compiler also includes hardware SIMD acceleration features when coupled with SIMD-enabled types in the System.Numerics namespace, which can yield good performance improvements.

- **Assembly loader improvements**

  The assembly loader now uses memory more efficiently by unloading IL assemblies after a corresponding NGEN image is loaded. This change decreases virtual memory, which is particularly beneficial for large 32-bit apps (such as Visual Studio), and also saves physical memory.

- **Base class library changes**

Many new APIs have been added around to .NET Framework 4.6 to enable key scenarios. These include the following changes and additions:

- **IReadOnlyCollection<T> implementations**

  Additional collections implement IReadOnlyCollection<T> such as Queue<T> and Stack<T>.

- **CultureInfo.CurrentCulture and CultureInfo.CurrentUICulture**

  The CultureInfo.CurrentCulture and CultureInfo.CurrentUICulture properties are now read-write rather than read-only. If you assign a new CultureInfo object to these properties, the current thread culture defined by the `Thread.CurrentThread.CurrentCulture` property and the current UI thread culture defined by the `Thread.CurrentThread.CurrentUICulture` properties also change.

- **Enhancements to garbage collection (GC)**

  The GC class now includes TryStartNoGCRegion and EndNoGCRegion methods that allow you to disallow garbage collection during the execution of a critical path.

  A new overload of the GC.Collect(Int32, GCCollectionMode, Boolean, Boolean) method allows you to control whether both the small object heap and the large object heap are swept and compacted or swept only.

- **SIMD-enabled types**

  The System.Numerics namespace now includes a number of SIMD-enabled types, such as Matrix3x2, Matrix4x4, Plane, Quaternion, Vector2, Vector3, and Vector4.

  Because the new 64-bit JIT compiler also includes hardware SIMD acceleration features, there are especially significant performance improvements when using the SIMD-enabled types with the new 64-bit JIT compiler.

- **Cryptography updates**

  The System.Security.Cryptography API is being updated to support the Windows CNG cryptography APIs. Previous versions of the .NET Framework have relied entirely on an earlier version of the Windows Cryptography APIs as the basis for the System.Security.Cryptography implementation. We have had requests to support the CNG API, since it supports modern cryptography algorithms, which are important for certain categories of apps.

  .NET Framework 4.6 includes the following new enhancements to support the Windows CNG cryptography APIs:

  - A set of extension methods for X509 Certificates, `System.Security.Cryptography.X509Certificates.RSACertificateExtensions.GetRSAPublicKey(System.Security.Cryptography.X509Certificates.X509Certificate2)` and `System.Security.Cryptography.X509Certificates.RSACertificateExtensions.GetRSAPrivateKey(System.Security.Cryptography.X509Certificates.X509Certificate2`, that return a CNG-based implementation rather than a CAPI-based implementation when possible. (Some smartcards, etc., still require CAPI, and the APIs handle the fallback).

  - The System.Security.Cryptography.RSACng class, which provides a CNG implementation of the RSA algorithm.

  - Enhancements to the RSA API so that common actions no longer require casting. For example, encrypting data using an X509Certificate2 object requires code like the following in previous versions of the .NET Framework.

    ```
    RSACryptoServiceProvider rsa = (RSACryptoServiceProvider)cert.PrivateKey;
    byte[] oaepEncrypted = rsa.Encrypt(data, true);
    byte[] pkcs1Encrypted = rsa.Encrypt(data, false);
    ```

    ```
    Dim rsa As RSACryptoServiceProvider = CType(cert.PrivateKey, RSACryptoServiceProvider)
    Dim oaepEncrypted() As Byte = rsa.Encrypt(data, True)
    Dim pkcs1Encrypted() As Byte = rsa.Encrypt(data, False)
    ```

    Code that uses the new cryptography APIs in .NET Framework 4.6 can be rewritten as follows to avoid the cast.

    ```
    RSA rsa = cert.GetRSAPrivateKey();
    if (rsa == null)
       throw new InvalidOperationException("An RSA certificate was expected");

    byte[] oaepEncrypted = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA1);
    byte[] pkcs1Encrypted = rsa.Encrypt(data, RSAEncryptionPadding.Pkcs1);
    ```

    ```
    Dim rsa As RSA = cert.GetRSAPrivateKey()
    If rsa Is Nothing Then
       Throw New InvalidOperationException("An RSA certificate was expected")
     End If

    Dim oaepEncrypted() As Byte = rsa.Encrypt(data, RSAEncryptionPadding.OaepSHA1)
    Dim pkcs1Encrypted() As Byte = rsa.Encrypt(data, RSAEncryptionPadding.Pkcs1)
    ```

- **Support for converting dates and times to or from Unix time**

  The following new methods have been added to the DateTimeOffset structure to support converting date and time values to or from Unix

time:

- DateTimeOffset.FromUnixTimeSeconds
- DateTimeOffset.FromUnixTimeMilliseconds
- DateTimeOffset.ToUnixTimeSeconds
- DateTimeOffset.ToUnixTimeMilliseconds

- **Compatibility switches**

  The new AppContext class adds a new compatibility feature that enables library writers to provide a uniform opt-out mechanism for new functionality for their users. It establishes a loosely-coupled contract between components in order to communicate an opt-out request. This capability is typically important when a change is made to existing functionality. Conversely, there is already an implicit opt-in for new functionality.

  With AppContext, libraries define and expose compatibility switches, while code that depends on them can set those switches to affect the library behavior. By default, libraries provide the new functionality, and they only alter it (that is, they provide the previous functionality) if the switch is set.

  An application (or a library) can declare the value of a switch (which is always a Boolean value) that a dependent library defines. The switch is always implicitly `false`. Setting the switch to `true` enables it. Explicitly setting the switch to `false` provides the new behavior.

```
AppContext.SetSwitch("Switch.AmazingLib.ThrowOnException", true);
```

```
AppContext.SetSwitch("Switch.AmazingLib.ThrowOnException", True)
```

  The library must check if a consumer has declared the value of the switch and then appropriately act on it.

```
if (!AppContext.TryGetSwitch("Switch.AmazingLib.ThrowOnException", out shouldThrow))
{
    // This is the case where the switch value was not set by the application.
    // The library can choose to get the value of shouldThrow by other means.
    // If no overrides nor default values are specified, the value should be 'false'.
    // A false value implies the latest behavior.
}

// The library can use the value of shouldThrow to throw exceptions or not.
if (shouldThrow)
{
    // old code
}
else
{
    // new code
}
```

```
If Not AppContext.TryGetSwitch("Switch.AmazingLib.ThrowOnException", shouldThrow) Then
    ' This is the case where the switch value was not set by the application.
    ' The library can choose to get the value of shouldThrow by other means.
    ' If no overrides nor default values are specified, the value should be 'false'.
    ' A false value implies the latest behavior.
End If

' The library can use the value of shouldThrow to throw exceptions or not.
If shouldThrow Then
    ' old code
Else
    ' new code
End If
```

  It's beneficial to use a consistent format for switches, since they are a formal contract exposed by a library. The following are two obvious formats.

  - *Switch.namespace.switchname*
  - *Switch.library.switchname*

- **Changes to the task-based asynchronous pattern (TAP)**

  For apps that target the .NET Framework 4.6, Task and Task<TResult> objects inherit the culture and UI culture of the calling thread. The behavior of apps that target previous versions of the .NET Framework, or that do not target a specific version of the .NET Framework, is unaffected. For more information, see the "Culture and task-based asynchronous operations" section of the CultureInfo class topic.

  The System.Threading.AsyncLocal<T> class allows you to represent ambient data that is local to a given asynchronous control flow, such as an `async` method. It can be used to persist data across threads. You can also define a callback method that is notified whenever the ambient data changes either because the AsyncLocal<T>.Value property was explicitly changed, or because the thread encountered a context transition.

Three convenience methods, Task.CompletedTask, Task.FromCanceled, and Task.FromException, have been added to the task-based asynchronous pattern (TAP) to return completed tasks in a particular state.

The NamedPipeClientStream class now supports asynchronous communication with its new ConnectAsync. method.

- **EventSource now supports writing to the Event log**

  You now can use the EventSource class to log administrative or operational messages to the event log, in addition to any existing ETW sessions created on the machine. In the past, you had to use the Microsoft.Diagnostics.Tracing.EventSource NuGet package for this functionality. This functionality is now built-into .NET Framework 4.6.

  Both the NuGet package and .NET Framework 4.6 have been updated with the following features:

  - **Dynamic events**

    Allows events defined "on the fly" without creating event methods.

  - **Rich payloads**

    Allows specially attributed classes and arrays as well as primitive types to be passed as a payload

  - **Activity tracking**

    Causes Start and Stop events to tag events between them with an ID that represents all currently active activities.

  To support these features, the overloaded Write method has been added to the EventSource class.

- **Windows Presentation Foundation (WPF)**

  - **HDPI improvements**

    HDPI support in WPF is now better in the .NET Framework 4.6. Changes have been made to layout rounding to reduce instances of clipping in controls with borders. By default, this feature is enabled only if your TargetFrameworkAttribute is set to .NET 4.6. Applications that target earlier versions of the framework but are running on the .NET Framework 4.6 can opt in to the new behavior by adding the following line to the <runtime> section of the app.config file:

    ```
    <AppContextSwitchOverrides
    value="Switch.MS.Internal.DoNotApplyLayoutRoundingToMarginsAndBorderThickness=false"
    />
    ```

    WPF windows straddling multiple monitors with different DPI settings (Multi-DPI setup) are now completely rendered without blacked-out regions. You can opt out of this behavior by adding the following line to the `<appSettings>` section of the app.config file to disable this new behavior:

    ```
    <add key="EnableMultiMonitorDisplayClipping" value="true"/>
    ```

    Support for automatically loading the right cursor based on DPI setting has been added to System.Windows.Input.Cursor.

  - **Touch is better**

    Customer reports on Connect that touch produces unpredictable behavior have been addressed in the .NET Framework 4.6. The double tap threshold for Windows Store applications and WPF applications is now the same in Windows 8.1 and above.

  - **Transparent child window support**

    WPF in the .NET Framework 4.6 supports transparent child windows in Windows 8.1 and above. This allows you to create non-rectangular and transparent child windows in your top-level windows. You can enable this feature by setting the HwndSourceParameters.UsesPerPixelTransparency property to `true`.

- **Windows Communication Foundation (WCF)**

  - **SSL support**

    WCF now supports SSL version TLS 1.1 and TLS 1.2, in addition to SSL 3.0 and TLS 1.0, when using NetTcp with transport security and client authentication. It is now possible to select which protocol to use, or to disable old lesser secure protocols. This can be done either by setting the SslProtocols property or by adding the following to a configuration file.

    ```
    <netTcpBinding>
        <binding>
          <security mode= "None|Transport|Message|TransportWithMessageCredential" >
              <transport clientCredentialType="None|Windows|Certificate"
                        protectionLevel="None|Sign|EncryptAndSign"
                        sslProtocols="Ssl3|Tls1|Tls11|Tls12">
                </transport>
          </security>
        </binding>
    </netTcpBinding>
    ```

  - **Sending messages using different HTTP connections**

WCF now allows users to ensure certain messages are sent using different underlying HTTP connections. There are two ways to do this:

- **Using a connection group name prefix**

  Users can specify a string that WCF will use as a prefix for the connection group name. Two messages with different prefixes are sent using different underlying HTTP connections. You set the prefix by adding a key/value pair to the message's Message.Properties property. The key is "HttpTransportConnectionGroupNamePrefix"; the value is the desired prefix.

- **Using different channel factories**

  Users can also enable a feature that ensures that messages sent using channels created by different channel factories will use different underlying HTTP connections. To enable this feature, users must set the following `appSetting` to `true`:

  ```
  <appSettings>
      <add key="wcf:httpTransportBinding:useUniqueConnectionPoolPerFactory" value="true" />
  </appSettings>
  ```

- **Windows Workflow Foundation (WWF)**

  You can now specify the number of seconds a workflow service will hold on to an out-of-order operation request when there is an outstanding "non-protocol" bookmark before timing out the request. A "non-protocol" bookmark is a bookmark that is not related to outstanding Receive activities. Some activities create non-protocol bookmarks within their implementation, so it may not be obvious that a non-protocol bookmark exists. These include State and Pick. So if you have a workflow service implemented with a state machine or containing a Pick activity, you will most likely have non-protocol bookmarks. You specify the interval by adding a line like the following to the `appSettings` section of your app.config file:

  ```
  <add key="microsoft:WorkflowServices:FilterResumeTimeoutInSeconds" value="60"/>
  ```

  The default value is 60 seconds. If `value` is set to 0, out-of-order requests are immediately rejected with a fault with text that looks like this:

  ```
  Operation 'Request3|{http://tempuri.org/}IService' on service instance with identifier '2b0667b6-09c8-4093-9d02-f6c67d534292' cannot be
  performed at this time. Please ensure that the operations are performed in the correct order and that the binding in use provides ordered
  delivery guarantees.
  ```

  This is the same message that you receive if an out-of-order operation message is received and there are no non-protocol bookmarks.

  If the value of the `FilterResumeTimeoutInSeconds` element is non-zero, there are non-protocol bookmarks, and the timeout interval expires, the operation fails with a timeout message.

- **Transactions**

  You can now include the distributed transaction identifier for the transaction that has caused an exception derived from TransactionException to be thrown. You do this by adding the following key to the `appSettings` section of your app.config file:

  ```
  <add key="Transactions:IncludeDistributedTransactionIdInExceptionMessage" value="true"/>
  ```

  The default value is `false`.

- **Networking**

  - **Socket reuse**

    Windows 10 includes a new high-scalability networking algorithm that makes better use of machine resources by reusing local ports for outbound TCP connections. .NET Framework 4.6 supports the new algorithm, enabling .NET apps to take advantage of the new behavior. In previous versions of Windows, there was an artificial concurrent connection limit (typically 16,384, the default size of the dynamic port range), which could limit the scalability of a service by causing port exhaustion when under load.

    In the .NET Framework 4.6, two new APIs have been added to enable port reuse, which effectively removes the 64K limit on concurrent connections:

    - The System.Net.Sockets.SocketOptionName enumeration value.
    - The ServicePointManager.ReusePort property.

    By default, the ServicePointManager.ReusePort property is `false` unless the `HWRPortReuseOnSocketBind` value of the `HKLM\SOFTWARE\Microsoft\.NETFramework\v4.0.30319` registry key is set to 0x1. To enable local port reuse on HTTP connections, set the ServicePointManager.ReusePort property to `true`. This causes all outgoing TCP socket connections from HttpClient and HttpWebRequest to use a new Windows 10 socket option, SO_REUSE_UNICASTPORT, that enables local port reuse.

    Developers writing a sockets-only application can specify the System.Net.Sockets.SocketOptionName option when calling a method such as Socket.SetSocketOption so that outbound sockets reuse local ports during binding.

  - **Support for international domain names and PunyCode**

    A new property, IdnHost, has been added to the Uri class to better support international domain names and PunyCode.

- **Resizing in Windows Forms controls.**

  This feature has been expanded in .NET Framework 4.6 to include the DomainUpDown, NumericUpDown, DataGridViewComboBoxColumn, DataGridViewColumn and ToolStripSplitButton types and the rectangle specified by the Bounds property used when drawing a UITypeEditor.

  This is an opt-in feature. To enable it, set the `EnableWindowsFormsHighDpiAutoResizing` element to `true` in the application configuration (app.config) file:

```
<appSettings>
    <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />
</appSettings>
```

- **Support for code page encodings**

  .NET Core primarily supports the Unicode encodings and by default provides limited support for code page encodings. You can add support for code page encodings available in .NET Framework but unsupported in .NET Core by registering code page encodings with the Encoding.RegisterProvider method. For more information, see System.Text.CodePagesEncodingProvider.

- **.NET Native**

  Windows apps for Windows 10 that target .NET Core and are written in C# or Visual Basic can take advantage of a new technology that compiles apps to native code rather than IL. They produce apps characterized by faster startup and execution times. For more information, see Compiling Apps with .NET Native. For an overview of .NET Native that examines how it differs from both JIT compilation and NGEN and what that means for your code, see .NET Native and Compilation.

  Your apps are compiled to native code by default when you compile them with Visual Studio 2015 or later. For more information, see Getting Started with .NET Native.

  To support debugging .NET Native apps, a number of new interfaces and enumerations have been added to the unmanaged debugging API. For more information, see the Debugging (Unmanaged API Reference) topic.

- **Open-source .NET Framework packages**

  .NET Core packages such as the immutable collections, SIMD APIs, and networking APIs such as those found in the System.Net.Http namespace are now available as open source packages on GitHub. To access the code, see CoreFx on GitHub. For more information and how to contribute to these packages, see .NET Core and Open-Source, .NET Home Page on GitHub.

What's new in .NET Framework 4.5.2

- **New APIs for ASP.NET apps.** The new HttpResponse.AddOnSendingHeaders and HttpResponseBase.AddOnSendingHeaders methods let you inspect and modify response headers and status code as the response is being flushed to the client app. Consider using these methods instead of the PreSendRequestHeaders and PreSendRequestContent events; they are more efficient and reliable.

  The HostingEnvironment.QueueBackgroundWorkItem method lets you schedule small background work items. ASP.NET tracks these items and prevents IIS from abruptly terminating the worker process until all background work items have completed. This method can't be called outside an ASP.NET managed app domain.

  The new HttpResponse.HeadersWritten and HttpResponseBase.HeadersWritten properties return Boolean values that indicate whether the response headers have been written. You can use these properties to make sure that calls to APIs such as HttpResponse.StatusCode (which throw exceptions if the headers have been written) will succeed.

- **Resizing in Windows Forms controls.** This feature has been expanded. You can now use the system DPI setting to resize components of the following additional controls (for example, the drop-down arrow in combo boxes):

  - ComboBox
  - ToolStripComboBox
  - ToolStripMenuItem
  - Cursor
  - DataGridView
  - DataGridViewComboBoxColumn

  This is an opt-in feature. To enable it, set the `EnableWindowsFormsHighDpiAutoResizing` element to `true` in the application configuration (app.config) file:

```
<appSettings>
    <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />
</appSettings>
```

- **New workflow feature.** A resource manager that's using the EnlistPromotableSinglePhase method (and therefore implementing the IPromotableSinglePhaseNotification interface) can use the new Transaction.PromoteAndEnlistDurable method to request the following:

  - Promote the transaction to a Microsoft Distributed Transaction Coordinator (MSDTC) transaction.
  - Replace IPromotableSinglePhaseNotification with an ISinglePhaseNotification, which is a durable enlistment that supports single phase commits.

This can be done within the same app domain, and doesn't require any extra unmanaged code to interact with MSDTC to perform the promotion. The new method can be called only when there's an outstanding call from System.Transactions to the IPromotableSinglePhaseNotification `Promote` method that's implemented by the promotable enlistment.

- **Profiling improvements.** The following new unmanaged profiling APIs provide more robust profiling:
  - COR_PRF_ASSEMBLY_REFERENCE_INFO Structure
  - COR_PRF_HIGH_MONITOR Enumeration
  - GetAssemblyReferences Method
  - GetEventMask2 Method
  - SetEventMask2 Method
  - AddAssemblyReference Method

  Previous `ICorProfiler` implementations supported lazy loading of dependent assemblies. The new profiling APIs require dependent assemblies that are injected by the profiler to be loadable immediately, instead of being loaded after the app is fully initialized. This change doesn't affect users of the existing `ICorProfiler` APIs.

- **Debugging improvements.** The following new unmanaged debugging APIs provide better integration with a profiler. You can now access metadata inserted by the profiler as well as local variables and code produced by compiler ReJIT requests when dump debugging.
  - SetWriteableMetadataUpdateMode Method
  - EnumerateLocalVariablesEx Method
  - GetLocalVariableEx Method
  - GetCodeEx Method
  - GetActiveReJitRequestILCode Method
  - GetInstrumentedILMap Method
- **Event tracing changes.** .NET Framework 4.5.2 enables out-of-process, Event Tracing for Windows (ETW)-based activity tracing for a larger surface area. This enables Advanced Power Management (APM) vendors to provide lightweight tools that accurately track the costs of individual requests and activities that cross threads. These events are raised only when ETW controllers enable them; therefore, the changes don't affect previously written ETW code or code that runs with ETW disabled.
- **Promoting a transaction and converting it to a durable enlistment**

  Transaction.PromoteAndEnlistDurable is a new API added to .NET Framework 4.5.2 and 4.6:

```
[System.Security.Permissions.PermissionSetAttribute(System.Security.Permissions.SecurityAction.LinkDemand, Name = "FullTrust")]
public Enlistment PromoteAndEnlistDurable(Guid resourceManagerIdentifier,
                                          IPromotableSinglePhaseNotification promotableNotification,
                                          ISinglePhaseNotification enlistmentNotification,
                                          EnlistmentOptions enlistmentOptions)
```

```
<System.Security.Permissions.PermissionSetAttribute(System.Security.Permissions.SecurityAction.LinkDemand, Name:="FullTrust")>
public Function PromoteAndEnlistDurable(resourceManagerIdentifier As Guid,
                                        promotableNotification As IPromotableSinglePhaseNotification,
                                        enlistmentNotification As ISinglePhaseNotification,
                                        enlistmentOptions As EnlistmentOptions) As Enlistment
```

  The method may be used by an enlistment that was previously created by Transaction.EnlistPromotableSinglePhase in response to the ITransactionPromoter.Promote method. It asks `System.Transactions` to promote the transaction to an MSDTC transaction and to "convert" the promotable enlistment to a durable enlistment. After this method completes successfully, the IPromotableSinglePhaseNotification interface will no longer be referenced by `System.Transactions`, and any future notifications will arrive on the provided ISinglePhaseNotification interface. The enlistment in question must act as a durable enlistment, supporting transaction logging and recovery. Refer to Transaction.EnlistDurable for details. In addition, the enlistment must support ISinglePhaseNotification. This method can *only* be called while processing an ITransactionPromoter.Promote call. If that is not the case, a TransactionException exception is thrown.

What's new in .NET Framework 4.5.1

April 2014 updates:

- Visual Studio 2013 Update 2 includes updates to the Portable Class Library templates to support these scenarios:
  - You can use Windows Runtime APIs in portable libraries that target Windows 8.1, Windows Phone 8.1, and Windows Phone Silverlight 8.1.
  - You can include XAML (Windows.UI.XAML types) in portable libraries when you target Windows 8.1 or Windows Phone 8.1. The following XAML templates are supported: Blank Page, Resource Dictionary, Templated Control, and User Control.
  - You can create a portable Windows Runtime component (.winmd file) for use in Store apps that target Windows 8.1 and Windows Phone 8.1.
  - You can retarget a Windows Store or Windows Phone Store class library like a Portable Class Library.

  For more information about these changes, see Portable Class Library.
- The .NET Framework content set now includes documentation for .NET Native, which is a precompilation technology for building and deploying

Windows apps. .NET Native compiles your apps directly to native code, rather than to intermediate language (IL), for better performance. For details, see Compiling Apps with .NET Native.

- The .NET Framework Reference Source provides a new browsing experience and enhanced functionality. You can now browse through the .NET Framework source code online, download the reference for offline viewing, and step through the sources (including patches and updates) during debugging. For more information, see the blog entry A new look for .NET Reference Source.

New features and enhancements in the base classes in .NET Framework 4.5.1 include:

- Automatic binding redirection for assemblies. Starting with Visual Studio 2013, when you compile an app that targets the .NET Framework 4.5.1, binding redirects may be added to the app configuration file if your app or its components reference multiple versions of the same assembly. You can also enable this feature for projects that target older versions of the .NET Framework. For more information, see How to: Enable and Disable Automatic Binding Redirection.
- Ability to collect diagnostics information to help developers improve the performance of server and cloud applications. For more information, see the WriteEventWithRelatedActivityId and WriteEventWithRelatedActivityIdCore methods in the EventSource class.
- Ability to explicitly compact the large object heap (LOH) during garbage collection. For more information, see the GCSettings.LargeObjectHeapCompactionMode property.
- Additional performance improvements such as ASP.NET app suspension, multi-core JIT improvements, and faster app startup after a .NET Framework update. For details, see the .NET Framework 4.5.1 announcement and the ASP.NET app suspend blog post.

Improvements to Windows Forms include:

- Resizing in Windows Forms controls. You can use the system DPI setting to resize components of controls (for example, the icons that appear in a property grid) by opting in with an entry in the application configuration file (app.config) for your app. This feature is currently supported in the following Windows Forms controls:
  - PropertyGrid
  - TreeView
  - Some aspects of the DataGridView (see new features in 4.5.2 for additional controls supported)

  To enable this feature, add a new <appSettings> element to the configuration file (app.config) and set the `EnableWindowsFormsHighDpiAutoResizing` element to `true`:

```
<appSettings>
    <add key="EnableWindowsFormsHighDpiAutoResizing" value="true" />
</appSettings>
```

Improvements when debugging your .NET Framework apps in Visual Studio 2013 include:

- Return values in the Visual Studio debugger. When you debug a managed app in Visual Studio 2013, the Autos window displays return types and values for methods. This information is available for desktop, Windows Store, and Windows Phone apps. For more information, see Examine return values of method calls.
- Edit and Continue for 64-bit apps. Visual Studio 2013 supports the Edit and Continue feature for 64-bit managed apps for desktop, Windows Store, and Windows Phone. The existing limitations remain in effect for both 32-bit and 64-bit apps (see the last section of the Supported Code Changes (C#) article).
- Async-aware debugging. To make it easier to debug asynchronous apps in Visual Studio 2013, the call stack hides the infrastructure code provided by compilers to support asynchronous programming, and also chains in logical parent frames so you can follow logical program execution more clearly. A Tasks window replaces the Parallel Tasks window and displays tasks that relate to a particular breakpoint, and also displays any other tasks that are currently active or scheduled in the app. You can read about this feature in the "Async-aware debugging" section of the .NET Framework 4.5.1 announcement.
- Better exception support for Windows Runtime components. In Windows 8.1, exceptions that arise from Windows Store apps preserve information about the error that caused the exception, even across language boundaries. You can read about this feature in the "Windows Store app development" section of the .NET Framework 4.5.1 announcement.

Starting with Visual Studio 2013, you can use the Managed Profile Guided Optimization Tool (Mpgo.exe) to optimize Windows 8.x Store apps as well as desktop apps.

For new features in ASP.NET 4.5.1, see ASP.NET and Web Tools for Visual Studio 2013 Release Notes.

What's new in .NET Framework 4.5

Base classes

- Ability to reduce system restarts by detecting and closing .NET Framework 4 applications during deployment. See Reducing System Restarts During .NET Framework 4.5 Installations.
- Support for arrays that are larger than 2 gigabytes (GB) on 64-bit platforms. This feature can be enabled in the application configuration file. See the <gcAllowVeryLargeObjects> element, which also lists other restrictions on object size and array size.
- Better performance through background garbage collection for servers. When you use server garbage collection in the .NET Framework 4.5, background garbage collection is automatically enabled. See the Background Server Garbage Collection section of the Fundamentals of Garbage

Collection topic.

- Background just-in-time (JIT) compilation, which is optionally available on multi-core processors to improve application performance. See ProfileOptimization.
- Ability to limit how long the regular expression engine will attempt to resolve a regular expression before it times out. See the Regex.MatchTimeout property.
- Ability to define the default culture for an application domain. See the CultureInfo class.
- Console support for Unicode (UTF-16) encoding. See the Console class.
- Support for versioning of cultural string ordering and comparison data. See the SortVersion class.
- Better performance when retrieving resources. See Packaging and Deploying Resources.
- Zip compression improvements to reduce the size of a compressed file. See the System.IO.Compression namespace.
- Ability to customize a reflection context to override default reflection behavior through the CustomReflectionContext class.
- Support for the 2008 version of the Internationalized Domain Names in Applications (IDNA) standard when the System.Globalization.IdnMapping class is used on Windows 8.
- Delegation of string comparison to the operating system, which implements Unicode 6.0, when the .NET Framework is used on Windows 8. When running on other platforms, the .NET Framework includes its own string comparison data, which implements Unicode 5.x. See the String class and the Remarks section of the SortVersion class.
- Ability to compute the hash codes for strings on a per application domain basis. See <UseRandomizedStringHashAlgorithm> Element.
- Type reflection support split between Type and TypeInfo classes. See Reflection in the .NET Framework for Windows Store Apps.

Managed Extensibility Framework (MEF)

In the .NET Framework 4.5, the Managed Extensibility Framework (MEF) provides the following new features:

- Support for generic types.
- Convention-based programming model that enables you to create parts based on naming conventions rather than attributes.
- Multiple scopes.
- A subset of MEF that you can use when you create Windows 8.x Store apps. This subset is available as a downloadable package from the NuGet Gallery. To install the package, open your project in Visual Studio, choose **Manage NuGet Packages** from the **Project** menu, and search online for the `Microsoft.Composition` package.

For more information, see Managed Extensibility Framework (MEF).

Asynchronous file operations

In the .NET Framework 4.5, new asynchronous features were added to the C# and Visual Basic languages. These features add a task-based model for performing asynchronous operations. To use this new model, use the asynchronous methods in the I/O classes. See Asynchronous File I/O.

Tools

In the .NET Framework 4.5, Resource File Generator (Resgen.exe) enables you to create a .resw file for use in Windows 8.x Store apps from a .resources file embedded in a .NET Framework assembly. For more information, see Resgen.exe (Resource File Generator).

Managed Profile Guided Optimization (Mpgo.exe) enables you to improve application startup time, memory utilization (working set size), and throughput by optimizing native image assemblies. The command-line tool generates profile data for native image application assemblies. See Mpgo.exe (Managed Profile Guided Optimization Tool). Starting with Visual Studio 2013, you can use Mpgo.exe to optimize Windows 8.x Store apps as well as desktop apps.

Parallel computing

The .NET Framework 4.5 provides several new features and improvements for parallel computing. These include improved performance, increased control, improved support for asynchronous programming, a new dataflow library, and improved support for parallel debugging and performance analysis. See the entry What's New for Parallelism in .NET 4.5 in the Parallel Programming with .NET blog.

Web

ASP.NET 4.5 and 4.5.1 add model binding for Web Forms, WebSocket support, asynchronous handlers, performance enhancements, and many other features. For more information, see the following resources:

- ASP.NET 4.5 and Visual Studio 2012
- ASP.NET and Web Tools for Visual Studio 2013 Release Notes

Networking

The .NET Framework 4.5 provides a new programming interface for HTTP applications. For more information, see the new System.Net.Http and System.Net.Http.Headers namespaces.

Support is also included for a new programming interface for accepting and interacting with a WebSocket connection by using the existing HttpListener and related classes. For more information, see the new System.Net.WebSockets namespace and the HttpListener class.

In addition, the .NET Framework 4.5 includes the following networking improvements:

- RFC-compliant URI support. For more information, see Uri and related classes.
- Support for Internationalized Domain Name (IDN) parsing. For more information, see Uri and related classes.
- Support for Email Address Internationalization (EAI). For more information, see the System.Net.Mail namespace.
- Improved IPv6 support. For more information, see the System.Net.NetworkInformation namespace.
- Dual-mode socket support. For more information, see the Socket and TcpListener classes.

Windows Presentation Foundation (WPF)

In the .NET Framework 4.5, Windows Presentation Foundation (WPF) contains changes and improvements in the following areas:

- The new Ribbon control, which enables you to implement a ribbon user interface that hosts a Quick Access Toolbar, Application Menu, and tabs.
- The new INotifyDataErrorInfo interface, which supports synchronous and asynchronous data validation.
- New features for the VirtualizingPanel and Dispatcher classes.
- Improved performance when displaying large sets of grouped data, and by accessing collections on non-UI threads.
- Data binding to static properties, data binding to custom types that implement the ICustomTypeProvider interface, and retrieval of data binding information from a binding expression.
- Repositioning of data as the values change (live shaping).
- Ability to check whether the data context for an item container is disconnected.
- Ability to set the amount of time that should elapse between property changes and data source updates.
- Improved support for implementing weak event patterns. Also, events can now accept markup extensions.

Windows Communication Foundation (WCF)

In the .NET Framework 4.5, the following features have been added to make it simpler to write and maintain Windows Communication Foundation (WCF) applications:

- Simplification of generated configuration files.
- Support for contract-first development.
- Ability to configure ASP.NET compatibility mode more easily.
- Changes in default transport property values to reduce the likelihood that you will have to set them.
- Updates to the XmlDictionaryReaderQuotas class to reduce the likelihood that you will have to manually configure quotas for XML dictionary readers.
- Validation of WCF configuration files by Visual Studio as part of the build process, so you can detect configuration errors before you run your application.
- New asynchronous streaming support.
- New HTTPS protocol mapping to make it easier to expose an endpoint over HTTPS with Internet Information Services (IIS).
- Ability to generate metadata in a single WSDL document by appending `?singleWSDL` to the service URL.
- Websockets support to enable true bidirectional communication over ports 80 and 443 with performance characteristics similar to the TCP transport.
- Support for configuring services in code.
- XML Editor tooltips.
- ChannelFactory caching support.
- Binary encoder compression support.
- Support for a UDP transport that enables developers to write services that use "fire and forget" messaging. A client sends a message to a service and expects no response from the service.
- Ability to support multiple authentication modes on a single WCF endpoint when using the HTTP transport and transport security.
- Support for WCF services that use internationalized domain names (IDNs).

For more information, see What's New in Windows Communication Foundation.

Windows Workflow Foundation (WF)

In the .NET Framework 4.5, several new features were added to Windows Workflow Foundation (WF), including:

- State machine workflows, which were first introduced as part of .NET Framework 4.0.1 (.NET Framework 4 Platform Update 1). This update included several new classes and activities that enabled developers to create state machine workflows. These classes and activities were updated for the .NET Framework 4.5 to include:
  - The ability to set breakpoints on states.

- The ability to copy and paste transitions in the workflow designer.
- Designer support for shared trigger transition creation.
- Activities for creating state machine workflows, including: StateMachine, State, and Transition.

- Enhanced Workflow Designer features such as the following:
  - Enhanced workflow search capabilities in Visual Studio, including **Quick Find** and **Find in Files**.
  - Ability to automatically create a Sequence activity when a second child activity is added to a container activity, and to include both activities in the Sequence activity.
  - Panning support, which enables the visible portion of a workflow to be changed without using the scroll bars.
  - A new **Document Outline** view that shows the components of a workflow in a tree-style outline view and lets you select a component in the **Document Outline** view.
  - Ability to add annotations to activities.
  - Ability to define and consume activity delegates by using the workflow designer.
  - Auto-connect and auto-insert for activities and transitions in state machine and flowchart workflows.
- Storage of the view state information for a workflow in a single element in the XAML file, so you can easily locate and edit the view state information.
- A NoPersistScope container activity to prevent child activities from persisting.
- Support for C# expressions:
  - Workflow projects that use Visual Basic will use Visual Basic expressions, and C# workflow projects will use C# expressions.
  - C# workflow projects that were created in Visual Studio 2010 and that have Visual Basic expressions are compatible with C# workflow projects that use C# expressions.
- Versioning enhancements:
  - The new WorkflowIdentity class, which provides a mapping between a persisted workflow instance and its workflow definition.
  - Side-by-side execution of multiple workflow versions in the same host, including WorkflowServiceHost.
  - In Dynamic Update, the ability to modify the definition of a persisted workflow instance.
- Contract-first workflow service development, which provides support for automatically generating activities to match an existing service contract.

For more information, see What's New in Windows Workflow Foundation.

.NET for Windows 8.x Store apps

Windows 8.x Store apps are designed for specific form factors and leverage the power of the Windows operating system. A subset of the .NET Framework 4.5 or 4.5.1 is available for building Windows 8.x Store apps for Windows by using C# or Visual Basic. This subset is called .NET for Windows 8.x Store apps and is discussed in an overview in the Windows Dev Center.

Portable Class Libraries

The Portable Class Library project in Visual Studio 2012 (and later versions) enables you to write and build managed assemblies that work on multiple .NET Framework platforms. Using a Portable Class Library project, you choose the platforms (such as Windows Phone and .NET for Windows 8.x Store apps) to target. The available types and members in your project are automatically restricted to the common types and members across these platforms. For more information, see Portable Class Library.

See also

- The .NET Framework and Out-of-Band Releases
- What's new in accessibility in the .NET Framework
- What's New in Visual Studio 2017
- ASP.NET
- What's New in Visual C++

Get started with the .NET Framework

6/5/2019 • 6 minutes to read • Edit Online

The .NET Framework is a runtime execution environment that manages apps that target the .NET Framework. It consists of the common language runtime, which provides memory management and other system services, and an extensive class library, which enables programmers to take advantage of robust, reliable code for all major areas of app development.

> **NOTE**
>
> The .NET Framework is available on Windows systems only. You can use .NET Core to run apps on Windows, MacOS, and Linux.

What is the .NET Framework?

The .NET Framework is a managed execution environment for Windows that provides a variety of services to its running apps. It consists of two major components: the common language runtime (CLR), which is the execution engine that handles running apps, and the .NET Framework Class Library, which provides a library of tested, reusable code that developers can call from their own apps. The services that the .NET Framework provides to running apps include the following:

- Memory management. In many programming languages, programmers are responsible for allocating and releasing memory and for handling object lifetimes. In .NET Framework apps, the CLR provides these services on behalf of the app.
- A common type system. In traditional programming languages, basic types are defined by the compiler, which complicates cross-language interoperability. In the .NET Framework, basic types are defined by the .NET Framework type system and are common to all languages that target the .NET Framework.
- An extensive class library. Instead of having to write vast amounts of code to handle common low-level programming operations, programmers use a readily accessible library of types and their members from the .NET Framework Class Library.
- Development frameworks and technologies. The .NET Framework includes libraries for specific areas of app development, such as ASP.NET for web apps, ADO.NET for data access, Windows Communication Foundation for service-oriented apps, and Windows Presentation Foundation for Windows desktop apps.
- Language interoperability. Language compilers that target the .NET Framework emit an intermediate code named Common Intermediate Language (CIL), which, in turn, is compiled at runtime by the common language runtime. With this feature, routines written in one language are accessible to other languages, and programmers focus on creating apps in their preferred languages.
- Version compatibility. With rare exceptions, apps that are developed by using a particular version of the .NET Framework run without modification on a later version.
- Side-by-side execution. The .NET Framework helps resolve version conflicts by allowing multiple versions of the common language runtime to exist on the same computer. This means that multiple versions of apps can coexist and that an app can run on the version of the .NET Framework with which it was built. Side-by-side execution applies to the .NET Framework version groups 1.0/1.1, 2.0/3.0/3.5, and 4/4.5.x/4.6.x/4.7.x/4.8.
- Multitargeting. By targeting .NET Standard, developers create class libraries that work on multiple .NET

Framework platforms supported by that version of the standard. For example, libraries that target the .NET Standard 2.0 can be used by apps that target the .NET Framework 4.6.1, .NET Core 2.0, and UWP 10.0.16299.

The .NET Framework for users

If you don't develop .NET Framework apps, but you use them, you aren't required to have specific knowledge about the .NET Framework or its operation. For the most part, the .NET Framework is completely transparent to users.

If you're using the Windows operating system, the .NET Framework may already be installed on your computer. In addition, if you install an app that requires the .NET Framework, the app's setup program might install a specific version of the .NET Framework on your computer. In some cases, you may see a dialog box that asks you to install the .NET Framework. If you've just tried to run an app when this dialog box appears and if your computer has Internet access, you can go to a webpage that lets you install the missing version of the .NET Framework. For more information, see the Installation guide.

In general, you shouldn't uninstall versions of the .NET Framework that are installed on your computer. There are two reasons for this:

- If an app that you use depends on a specific version of the .NET Framework, that app may break if that version is removed.
- Some versions of the .NET Framework are in-place updates to earlier versions. For example, the .NET Framework 3.5 is an in-place update to version 2.0, and the .NET Framework 4.8 is an in-place update to versions 4 through 4.7.2. For more information, see .NET Framework Versions and Dependencies.

On Windows versions before Windows 8, if you do choose to remove the .NET Framework, always use **Programs and Features** from Control Panel to uninstall it. Never remove a version of the .NET Framework manually. On Windows 8 and above, the .NET Framework is an operating system component and cannot be independently uninstalled.

Note that multiple versions of the .NET Framework can coexist on a single computer at the same time. This means that you don't have to uninstall previous versions in order to install a later version.

The .NET Framework for developers

If you're a developer, choose any programming language that supports the .NET Framework to create your apps. Because the .NET Framework provides language independence and interoperability, you interact with other .NET Framework apps and components regardless of the language with which they were developed.

To develop .NET Framework apps or components, do the following:

1. If it's not preinstalled on your operating system, install the version of the .NET Framework that your app will target. The most recent production version is the .NET Framework 4.8. It is preinstalled on Windows 10 May 2019 Update, and it is available for download on earlier versions of the Windows operating system. For .NET Framework system requirements, see System Requirements. For information on installing other versions of the .NET Framework, see Installation Guide. Additional .NET Framework packages are released out of band, which means that they're released on a rolling basis outside of any regular or scheduled release cycle. For information about these packages, see The .NET Framework and Out-of-Band Releases.
2. Select the language or languages supported by the .NET Framework that you intend to use to develop your apps. A number of languages are available, including Visual Basic, C#, F#, and C++/CLI from Microsoft. (A programming language that allows you to develop apps for the .NET Framework adheres to the Common Language Infrastructure (CLI) specification.)
3. Select and install the development environment to use to create your apps and that supports your selected

programming language or languages. The Microsoft integrated development environment (IDE) for .NET Framework apps is Visual Studio. It's available in a number of editions.

For more information on developing apps that target the .NET Framework, see the Development Guide.

Related articles

TITLE	DESCRIPTION
Overview	Provides detailed information for developers who build apps that target the .NET Framework.
Installation guide	Provides information about installing the .NET Framework.
The .NET Framework and Out-of-Band Releases	Describes the .NET Framework out of band releases and how to use them in your app.
System Requirements	Lists the hardware and software requirements for running the .NET Framework.
.NET Core and Open-Source	Describes .NET Core in relation to the .NET Framework and how to access the open-source .NET Core projects.
.NET Core documentation	Provides the conceptual and API reference documentation for .NET Core.
.NET Standard	Discusses .NET Standard, a versioned specification that individual .NET implementations support to guarantee that a consistent set of APIs are available on multiple platforms.

See also

- .NET Framework Guide
- What's New
- .NET API Browser
- Development Guide

Installation guide

1/23/2019 • 2 minutes to read • Edit Online

You can install .NET Framework on various Windows versions.

Supported Windows versions

- Windows 10 and Windows Server 2016
- Windows 8.1 and Windows Server 2012 R2
- Windows 8 and Windows Server 2012
- Windows 7 and Windows Server 2008 R2
- Windows Vista and Windows Server 2008

Unsupported Windows versions

- Windows XP and Windows Server 2003

See also

- Download the .NET Framework
- Troubleshoot blocked .NET Framework installations and uninstallations
- Install the .NET Framework for developers
- Deploy the .NET Framework for developers

Migration Guide to the .NET Framework 4.8, 4.7, 4.6, and 4.5

5/15/2019 • 2 minutes to read • Edit Online

If you created your app using an earlier version of the .NET Framework, you can generally upgrade it to .NET Framework 4.5 and its point releases (4.5.1 and 4.5.2), .NET Framework 4.6 and its point releases (4.6.1 and 4.6.2), .NET Framework 4.7 and its point releases (4.7.1 and 4.7.2), or .NET Framework 4.8 easily. Open your project in Visual Studio. If your project was created in an earlier version of Visual Studio, the **Project Compatibility** dialog box automatically opens. For more information about upgrading a project in Visual Studio, see Port, Migrate, and Upgrade Visual Studio Projects and Visual Studio 2019 Platform Targeting and Compatibility.

However, some changes in the .NET Framework require changes to your code. You may also want to take advantage of functionality that is new in .NET Framework 4.5 and its point releases, in .NET Framework 4.6 and its point releases, in .NET Framework 4.7 and its point releases, or in .NET Framework 4.8. Making these types of changes to your app for a new version of the .NET Framework is typically referred to as *migration*. If your app doesn't have to be migrated, you can run it in the .NET Framework 4.5 or a later version without recompiling it.

Migration resources

Review the following documents before you migrate your app from earlier versions of the .NET Framework to version 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, or 4.8:

- See Versions and Dependencies to understand the CLR version underlying each version of the .NET Framework and to review guidelines for targeting your apps successfully.
- Review Application Compatibility to find out about runtime and retargeting changes that might affect your app and how to handle them.
- Review What's Obsolete in the Class Library to determine any types or members in your code that have been made obsolete, and the recommended alternatives.
- See What's New for descriptions of new features that you may want to add to your app.

See also

- Application Compatibility
- Migrating from the .NET Framework 1.1
- Version Compatibility
- Versions and Dependencies
- How to: Configure an app to support .NET Framework 4 or later versions
- What's New
- What's Obsolete in the Class Library
- .NET Framework Version and Assembly Information
- Microsoft .NET Framework Support Lifecycle Policy
- .NET Framework 4 migration issues

.NET Framework Development Guide

8/22/2019 • 2 minutes to read • Edit Online

This section explains how to create, configure, debug, secure, and deploy your .NET Framework apps. The section also provides information about technology areas such as dynamic programming, interoperability, extensibility, memory management, and threading.

In This Section

Application Essentials

Provides information about basic app development tasks, such as programming with app domains and assemblies, using attributes, formatting and parsing base types, using collections, handling events and exceptions, using files and data streams, and using generics.

Data and Modeling

Provides information about how to access data using ADO.NET, Language Integrated Query (LINQ), WCF Data Services, and XML.

Client Applications

Explains how to create Windows-based apps by using Windows Presentation Foundation (WPF) or Windows Forms.

Web Applications with ASP.NET

Provides links to information about using ASP.NET to build enterprise-class web apps with a minimum of coding.

Service-Oriented Applications with WCF

Describes how to use Windows Communication Foundation (WCF) to build service-oriented apps that are secure and reliable.

Building workflows with Windows Workflow Foundation

Provides information about the programming model, samples, and tools for using Windows Workflow Foundation (WF).

Windows Service Applications

Explains how you can use Visual Studio and the .NET Framework to create an app that is installed as a service, and start, stop, and otherwise control its behavior.

Parallel Processing, Concurrency, and Async Programming in .NET

Provides information about managed threading, parallel programming, and asynchronous programming design patterns.

Network Programming in the .NET Framework

Describes the layered, extensible, and managed implementation of Internet services that you can quickly and easily integrate into your apps.

Configuring .NET Framework Apps

Explains how you can use configuration files to change settings without having to recompile your .NET Framework apps.

Compiling Apps with .NET Native

Explains how you can use the .NET Native precompilation technology to build and deploy Windows Store apps. .NET Native compiles apps that are written in managed code (C#) and that target the .NET Framework to native

code.

Security
Provides information about the classes and services in the .NET Framework that facilitate secure app development.

Debugging, Tracing, and Profiling
Explains how to test, optimize, and profile .NET Framework apps and the app environment. This section includes information for administrators as well as developers.

Developing for Multiple Platforms
Provides information about how you can use the .NET Framework to build assemblies that can be shared across multiple platforms and multiple devices such as phones, desktop, and web.

Deployment
Explains how to package and distribute your .NET Framework app, and includes deployment guides for both developers and administrators.

Performance
Provides information about caching, lazy initialization, reliability, and ETW events.

Reference

.NET Framework Class Library
Supplies syntax, code examples, and usage information for each class that is contained in the .NET Framework namespaces.

Related Sections

Getting Started
Provides a comprehensive overview of the .NET Framework and links to additional resources.

What's New
Describes key new features and changes in the latest version of the .NET Framework. Includes lists of new and obsolete types and members, and provides a guide for migrating your apps from the previous version of the .NET Framework.

Tools
Describes the tools that help you develop, configure, and deploy apps by using .NET Framework technologies.

.NET samples and tutorials
Provides links to samples and tutorials that help you learn about .NET.

Programming with Application Domains and Assemblies

5/2/2018 • 2 minutes to read • Edit Online

Hosts such as Microsoft Internet Explorer, ASP.NET, and the Windows shell load the common language runtime into a process, create an application domain in that process, and then load and execute user code in that application domain when running a .NET Framework application. In most cases, you do not have to worry about creating application domains and loading assemblies into them because the runtime host performs those tasks.

However, if you are creating an application that will host the common language runtime, creating tools or code you want to unload programmatically, or creating pluggable components that can be unloaded and reloaded on the fly, you will be creating your own application domains. Even if you are not creating a runtime host, this section provides important information on how to work with application domains and assemblies loaded in these application domains.

In This Section

Application Domains and Assemblies How-to Topics
Provides links to all How-to topics found in the conceptual documentation for programming with application domains and assemblies.

Using Application Domains
Provides examples of creating, configuring, and using application domains.

Programming with Assemblies
Describes how to create, sign, and set attributes on assemblies.

Related Sections

Emitting Dynamic Methods and Assemblies
Describes how to create dynamic assemblies.

Assemblies in the Common Language Runtime
Provides a conceptual overview of assemblies.

Application Domains
Provides a conceptual overview of application domains.

Reflection Overview
Describes how to use the **Reflection** class to obtain information about an assembly.

Resources in .NET Apps

8/22/2019 • 3 minutes to read • Edit Online

Nearly every production-quality app has to use resources. A resource is any nonexecutable data that is logically deployed with an app. A resource might be displayed in an app as error messages or as part of the user interface. Resources can contain data in a number of forms, including strings, images, and persisted objects. (To write persisted objects to a resource file, the objects must be serializable.) Storing your data in a resource file enables you to change the data without recompiling your entire app. It also enables you to store data in a single location, and eliminates the need to rely on hard-coded data that is stored in multiple locations.

The .NET Framework and .NET Core provide comprehensive support for the creation and localization of resources. In addition, .NET supports a simple model for packaging and deploying localized resources.

For information about resources in ASP.NET, see ASP.NET Web Page Resources Overview.

Creating and Localizing Resources

In a non-localized app, you can use resource files as a repository for app data, particularly for strings that might otherwise be hard-coded in multiple locations in source code. Most commonly, you create resources as either text (.txt) or XML (.resx) files, and use Resgen.exe (Resource File Generator) to compile them into binary .resources files. These files can then be embedded in the app's executable file by a language compiler. For more information about creating resources, see Creating Resource Files.

You can also localize your app's resources for specific cultures. This enables you to build localized (translated) versions of your apps. When you develop an app that uses localized resources, you designate a culture that serves as the neutral or fallback culture whose resources are used if no suitable resources are available. Typically, the resources of the neutral culture are stored in the app's executable. The remaining resources for individual localized cultures are stored in standalone satellite assemblies. For more information, see Creating Satellite Assemblies.

Packaging and Deploying Resources

You deploy localized app resources in satellite assemblies. A satellite assembly contains the resources of a single culture; it does not contain any app code. In the satellite assembly deployment model, you create an app with one default assembly (which is typically the main assembly) and one satellite assembly for each culture that the app supports. Because the satellite assemblies are not part of the main assembly, you can easily replace or update resources corresponding to a specific culture without replacing the app's main assembly.

Carefully determine which resources will make up your app's default resource assembly. Because it is a part of the main assembly, any changes to it will require you to replace the main assembly. If you do not provide a default resource, an exception will be thrown when the resource fallback process attempts to find it. In a well-designed app, using resources should never throw an exception.

For more information, see the Packaging and Deploying Resources article.

Retrieving Resources

At run time, an app loads the appropriate localized resources on a per-thread basis, based on the culture specified by the CultureInfo.CurrentUICulture property. This property value is derived as follows:

- By directly assigning a CultureInfo object that represents the localized culture to the Thread.CurrentUICulture property.

- If a culture is not explicitly assigned, by retrieving the default thread UI culture from the CultureInfo.DefaultThreadCurrentUICulture property.
- If a default thread UI culture is not explicitly assigned, by retrieving the culture for the current user on the local computer. .NET implementations running on Windows do this by calling the Windows `GetUserDefaultUILanguage` function.

For more information about how the current UI culture is set, see the CultureInfo and CultureInfo.CurrentUICulture reference pages.

You can then retrieve resources for the current UI culture or for a specific culture by using the System.Resources.ResourceManager class. Although the ResourceManager class is most commonly used for retrieving resources, the System.Resources namespace contains additional types that you can use to retrieve resources. These include:

- The ResourceReader class, which enables you to enumerate resources embedded in an assembly or stored in a standalone binary .resources file. It is useful when you don't know the precise names of the resources that are available at run time.
- The ResXResourceReader class, which enables you to retrieve resources from an XML (.resx) file.
- The ResourceSet class, which enables you to retrieve the resources of a specific culture without observing fallback rules. The resources can be stored in an assembly or a standalone binary .resources file. You can also develop an IResourceReader implementation that enables you to use the ResourceSet class to retrieve resources from some other source.
- The ResXResourceSet class, which enables you to retrieve all the items in an XML resource file into memory.

See also

- CultureInfo
- CultureInfo.CurrentUICulture
- Application Essentials
- Creating Resource Files
- Packaging and Deploying Resources
- Creating Satellite Assemblies
- Retrieving Resources

Accessibility

8/22/2019 • 2 minutes to read • Edit Online

> **NOTE**
> This documentation is intended for .NET Framework developers who want to use the managed UI Automation classes defined in the System.Windows.Automation namespace. For the latest information about UI Automation, see Windows Automation API: UI Automation.

Microsoft UI Automation is the new accessibility framework for Microsoft Windows. It addresses the needs of assistive technology products and automated test frameworks by providing programmatic access to information about the user interface (UI). In addition, UI Automation enables control and application developers to make their products accessible.

This documentation describes the UI Automation API for managed code. For information on programming for UI Automation in C++, see UI Automation for Win32 Applications.

In This Section

Accessibility Best Practices
UI Automation Fundamentals
UI Automation Providers for Managed Code
UI Automation Clients for Managed Code
UI Automation Control Patterns
UI Automation Text Pattern
UI Automation Control Types
UI Automation Specification and Community Promise

Related Sections

- Accessibility Samples

Data and Modeling in the .NET Framework

8/22/2019 • 2 minutes to read • Edit Online

This section provides information on how to access data using ADO.NET, Language Integrated Query (LINQ), WCF Data Services, and XML.

In This Section

ADO.NET

Describes the ADO.NET architecture and how to use the ADO.NET classes to manage application data and interact with data sources, including Microsoft SQL Server, OLE DB data sources, and XML.

Language Integrated Query (LINQ)

Provides links to relevant documentation for Language Integrated Query (LINQ) using C#.

Language-Integrated Query (LINQ) (Visual Basic)

Provides links to relevant documentation for Language Integrated Query (LINQ) using Visual Basic.

Transaction Processing

Discusses the .NET Framework support for transactions.

WCF Data Services 4.5

Provides information about how to use WCF Data Services to deploy data services on the Web or an intranet.

XML Documents and Data

Provides an overview to a comprehensive and integrated set of classes that work with XML documents and data in the .NET Framework.

XML Standards Reference

Provides reference information on XML standards that Microsoft supports.

Related Sections

Microsoft SQL Server Modeling Technologies

Describes a set of technologies that enable rapid and customized data-based application design and development.

Development Guide

Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.

Security

Provides links to more information on the classes and services in the common language runtime and the .NET Framework that facilitate secure application development.

Developing client applications with the .NET Framework

1/23/2019 • 2 minutes to read • Edit Online

There are several ways to develop Windows-based applications with the .NET Framework. You can use any of these tools and frameworks:

- Universal Windows Platform (UWP)
- Windows Presentation Foundation (WPF)
- Windows Forms

This section contains topics that describe how to create Windows-based applications by using Windows Presentation Foundation or by using Windows Forms. However, you can also create web applications using the .NET Framework, and client applications for computers or devices that you make available through the Microsoft Store.

In this section

Windows Presentation Foundation
Provides information about developing applications by using WPF.

Windows Forms
Provides information about developing applications by using Windows Forms.

Common Client Technologies
Provides information about additional technologies that can be used when developing client applications.

Related sections

Universal Windows Platform
Describes how to create applications for Windows 10 that you can make available to users through the Windows Store.

.NET for UWP apps
Describes the .NET Framework support for Store apps, which can be deployed to Windows computers and devices.

.NET API for Windows Phone Silverlight
Lists the .NET Framework APIs you can use for building apps with Windows Phone Silverlight.

Developing for Multiple Platforms
Describes the different methods you can use the .NET Framework to target multiple client app types.

Get Started with ASP.NET Web Sites
Describes the ways you can develop web apps using ASP.NET.

See also

- .NET Standard
- Overview
- Development Guide

- Windows Service Applications

Windows Presentation Foundation

5/4/2018 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) in Visual Studio provides developers with a unified programming model for building line-of-business desktop applications on Windows.

Create Desktop Applications with Windows Presentation Foundation

Designing XAML in Visual Studio and Blend for Visual Studio

Get Visual Studio

Windows Forms

5/14/2019 • 2 minutes to read • Edit Online

As forms are the base unit of your application, it is essential that you give some thought to their function and design. A form is ultimately a blank slate that you, as a developer, enhance with controls to create a user interface and with code to manipulate data. To that end, Visual Studio provides you with an integrated development environment (IDE) to aid in writing code, as well as a rich control set written with the .NET Framework. By complementing the functionality of these controls with your code, you can easily and quickly develop the solutions you need.

In This Section

Getting Started with Windows Forms
Provides links to topics about how to harness the power of Windows Forms to display data, handle user input, and deploy your applications easily and with more robust security.

Enhancing Windows Forms Applications
Provides links to topics about how to enhance your Windows Forms with a variety of features.

Related Sections

Windows Forms Controls
Contains links to topics that describe Windows Forms controls and show how to implement them.

Windows Forms Data Binding
Contains links to topics that describe the Windows Forms data-binding architecture.

Graphics Overview
Discusses how to create graphics, draw text, and manipulate graphical images as objects using the advanced implementation of the Windows graphics design interface.

ClickOnce Security and Deployment
Discusses the principles of ClickOnce deployment.

Windows Forms/MFC Programming Differences
Discusses the differences between MFC applications and Windows Forms.

Accessing data in Visual Studio
Discusses incorporating data access functionality into your applications.

Windows Forms Applications
Discusses the process of debugging applications created with the Windows Application project template, as well as how to change the Debug and Release configurations.

First look at deployment in Visual Studio
Describes the process by which you distribute a finished application or component to be installed on other computers.

Building Console Applications
Describes the basics of creating a console application using the Console class.

Developing Service-Oriented Applications with WCF

5/5/2018 • 2 minutes to read • Edit Online

This section of the documentation provides information about Windows Communication Foundation (WCF), which is a unified programming model for building service-oriented applications. It enables developers to build secure, reliable, transacted solutions that integrate across platforms and interoperate with existing investments.

In this section

What's New in Windows Communication Foundation 4.5
Discusses features new to Windows Communication Foundation.

WCF Simplification Features
Discusses new features that make writing WCF applications simpler.

Guide to the Documentation
A description of the WCF documentation

Conceptual Overview
Summarizes information about the Windows Communication Foundation (WCF) messaging system and the classes that support its use.

Getting Started Tutorial
A step by step tutorial to create a WCF service and client

Basic WCF Programming
Describes the fundamentals for creating Windows Communication Foundation applications.

WCF Feature Details
Shows topics that let you choose which WCF feature or features you need to employ.

Extending WCF
Describes how to modify and extend WCF runtime components

Guidelines and Best Practices
Provides guidelines for creating Windows Communication Foundation (WCF) applications.

Administration and Diagnostics
Describes the diagnostic features of WCF

System Requirements
Describes system requirements needed to run WCF

Operating System Resources Required by WCF
Describes operating system resources required by WCF

Troubleshooting Setup Issues
Provides guidance for fixing WCF setup issues

Migrating from .NET Remoting to WCF
Compares .NET Remoting to WCF and provides migration guidance for common scenarios.

Using the WCF Development Tools
Describes the Visual Studio Windows Communication Foundation development tools that can assist you in developing your WCFservice.

Windows Communication Foundation Tools

Describes WCF tools designed to make it easier to create, deploy, and manage WCF applications

Windows Communication Foundation Samples

Samples that provide instruction on various aspects of Windows Communication Foundation

Windows Communication Foundation Glossary

Shows a list of terms specific to WCF

General Reference

The section describes the elements that are used to configure Windows Communication Foundation clients and services.

Feedback and Community

Information about how to provide feedback about Windows Communication Foundation

Privacy Information

Information regarding WCF and Privacy

Windows Workflow Foundation

3/9/2019 • 2 minutes to read • Edit Online

This section describes the programming model, samples, and tools of the Windows Workflow Foundation (WF).

In This Section

Guide to the Windows Workflow Documentation

A set of suggested topics to read, depending upon your familiarity (novice to well-acquainted), and requirements.

What's New in Windows Workflow Foundation

Discusses the changes in several development paradigms from previous versions.

What's New in Windows Workflow Foundation in .NET 4.5

Describes the new features in Windows Workflow Foundation in .NET Framework 4.6.1.

Windows Workflow Foundation Feature Specifics

Describes the new features in Windows Workflow Foundation in .NET Framework 4

Windows Workflow Conceptual Overview

A set of topics that discusses the larger concepts behind Windows Workflow Foundation.

Getting Started Tutorial

A set of walkthrough topics that introduce you to programming Windows Workflow Foundation applications.

Windows Workflow Foundation Programming

A set of primer topics that you should understand to become a proficient WF programmer.

Extending Windows Workflow Foundation

A set of topics that discusses how to extend or customize Windows Workflow Foundation to suit your needs.

Windows Workflow Foundation Glossary for .NET Framework 4.5

Defines a list of terms that are specific to WF.

Windows Workflow Samples

Contains sample applications that demonstrate WF features and scenarios.

Develop Windows service apps

2/20/2019 • 2 minutes to read • Edit Online

Using Visual Studio or the .NET Framework SDK, you can easily create services by creating an application that is installed as a service. This type of application is called a Windows service. With framework features, you can create services, install them, and start, stop, and otherwise control their behavior.

> **NOTE**
> In Visual Studio you can create a service in managed code in Visual C# or Visual Basic, which can interoperate with existing C++ code if required. Or, you can create a Windows service in native C++ by using the ATL Project Wizard.

In this section

Introduction to Windows Service Applications

Provides an overview of Windows service applications, the lifetime of a service, and how service applications differ from other common project types.

Walkthrough: Creating a Windows Service Application in the Component Designer

Provides an example of creating a service in Visual Basic and Visual C#.

Service Application Programming Architecture

Explains the language elements used in service programming.

How to: Create Windows Services

Describes the process of creating and configuring Windows services using the Windows service project template.

Related sections

ServiceBase - Describes the major features of the ServiceBase class, which is used to create services.

ServiceProcessInstaller - Describes the features of the ServiceProcessInstaller class, which is used along with the ServiceInstaller class to install and uninstall your services.

ServiceInstaller - Describes the features of the ServiceInstaller class, which is used along with the ServiceProcessInstaller class to install and uninstall your service.

Create Projects from Templates - Describes the projects types used in this chapter and how to choose between them.

64-bit Applications

8/23/2019 • 3 minutes to read • Edit Online

When you compile an application, you can specify that it should run on a Windows 64-bit operating system either as a native application or under WOW64 (Windows 32-bit on Windows 64-bit). WOW64 is a compatibility environment that enables a 32-bit application to run on a 64-bit system. WOW64 is included in all 64-bit versions of the Windows operating system.

Running 32-bit vs. 64-bit Applications on Windows

All applications that are built on the .NET Framework 1.0 or 1.1 are treated as 32-bit applications on a 64-bit operating system and are always executed under WOW64 and the 32-bit common language runtime (CLR). 32-bit applications that are built on the .NET Framework 4 or later versions also run under WOW64 on 64-bit systems.

Visual Studio installs the 32-bit version of the CLR on an x86 computer, and both the 32-bit version and the appropriate 64-bit version of the CLR on a 64-bit Windows computer. (Because Visual Studio is a 32-bit application, when it is installed on a 64-bit system, it runs under WOW64.)

> **NOTE**
> Because of the design of x86 emulation and the WOW64 subsystem for the Itanium processor family, applications are restricted to execution on one processor. These factors reduce the performance and scalability of 32-bit .NET Framework applications that run on Itanium-based systems. We recommend that you use the .NET Framework 4, which includes native 64-bit support for Itanium-based systems, for increased performance and scalability.

By default, when you run a 64-bit managed application on a 64-bit Windows operating system, you can create an object of no more than 2 gigabytes (GB). However, in the .NET Framework 4.5, you can increase this limit. For more information, see the <gcAllowVeryLargeObjects> element.

Many assemblies run identically on both the 32-bit CLR and the 64-bit CLR. However, some programs may behave differently, depending on the CLR, if they contain one or more of the following:

- Structures that contain members that change size depending on the platform (for example, any pointer type).
- Pointer arithmetic that includes constant sizes.
- Incorrect platform invoke or COM declarations that use `Int32` for handles instead of `IntPtr`.
- Code that casts `IntPtr` to `Int32`.

For more information about how to port a 32-bit application to run on the 64-bit CLR, see Migrating 32-bit Managed Code to 64-bit.

General 64-Bit Programming Information

For general information about 64-bit programming, see the following documents:

- For more information about the 64-bit version of the CLR on a 64-bit Windows computer, see the .NET Framework Developer Center on the MSDN website.
- In the Windows SDK documentation, see Programming Guide for 64-bit Windows.

- For information about how to download a 64-bit version of the CLR, see .NET Framework Developer Center Downloads on the MSDN website.
- For information about Visual Studio support for creating 64-bit applications, see Visual Studio IDE 64-Bit Support.

Compiler Support for Creating 64-Bit Applications

By default, when you use the .NET Framework to build an application on either a 32-bit or a 64-bit computer, the application will run on a 64-bit computer as a native application (that is, not under WOW64). The following table lists documents that explain how to use Visual Studio compilers to create 64-bit applications that will run as native, under WOW64, or both.

COMPILER	COMPILER OPTION
Visual Basic	/platform (Visual Basic)
Visual C#	/platform (C# Compiler Options)
Visual C++	You can create platform-agnostic, Microsoft intermediate language (MSIL) applications by using **/clr:safe**. For more information, see /clr (Common Language Runtime Compilation). Visual C++ includes a separate compiler for each 64-bit operating system. For more information about how to use Visual C++ to create native applications that run on a 64-bit Windows operating system, see 64-bit Programming.

Determining the Status of an .exe File or .dll File

To determine whether an .exe file or .dll file is meant to run only on a specific platform or under WOW64, use CorFlags.exe (CorFlags Conversion Tool) with no options. You can also use CorFlags.exe to change the platform status of an .exe file or .dll file. The CLR header of a Visual Studio assembly has the major runtime version number set to 2 and the minor runtime version number set to 5. Applications that have the minor runtime version set to 0 are treated as legacy applications and are always executed under WOW64.

To programmatically query an .exe or .dll to see whether it is meant to run only on a specific platform or under WOW64, use the Module.GetPEKind method.

Developing Web apps with ASP.NET

8/14/2019 • 2 minutes to read • Edit Online

ASP.NET is a .NET Framework technology for creating web apps. For more information on ASP.NET, see:

- ASP.NET documentation
- ASP.NET MVC
- ASP.NET Web Pages
- ASP.NET Web API
- Create an ASP.NET Framework web app in Azure

Developing Web apps with ASP.NET Core

ASP.NET Core is a redesign of ASP.NET 4.x. Some of the benefits ASP.NET Core provides over ASP.NET:

- Cross platform.
- Leaner and more modular.
- A unified story for building web UI and web APIs.

See Why use ASP.NET Core? for an expanded list of benefits.

For more information on ASP.NET Core](/aspnet/core), see:

- Get started with Razor Pages
- Create a Web API
- Create an ASP.NET Core web app in Azure

See also

- Development Guide

Network Programming in the .NET Framework

1/23/2019 • 4 minutes to read • Edit Online

The Microsoft .NET Framework provides a layered, extensible, and managed implementation of Internet services that can be quickly and easily integrated into your applications. Your network applications can build on pluggable protocols to automatically take advantage of new Internet protocols, or they can use a managed implementation of the Windows socket interface to work with the network on the socket level.

In This Section

Introducing Pluggable Protocols
Describes how to access an Internet resource without regard to the access protocol that it requires.

Requesting Data
Explains how to use pluggable protocols to upload and download data from Internet resources.

Programming Pluggable Protocols
Explains how to derive protocol-specific classes to implement pluggable protocols.

Using Application Protocols
Describes programming applications that take advantage of network protocols such as TCP, UDP, and HTTP.

Internet Protocol Version 6
Describes the advantages of Internet Protocol version 6 (IPv6) over the current version of the Internet Protocol suite (IPv4), describes IPv6 addressing, routing and auto-configuration, and how to enable and disable IPv6.

Configuring Internet Applications
Explains how to use the .NET Framework configuration files to configure Internet applications.

Network Tracing in the .NET Framework
Explains how to use network tracing to get information about method invocations and network traffic generated by a managed application.

Cache Management for Network Applications
Describes how to use caching for applications that use the System.Net.WebClient, System.Net.WebRequest, and System.Net.HttpWebRequest classes.

Security in Network Programming
Describes how to use standard Internet security and authentication techniques.

Best Practices for System.Net Classes
Provides tips and tricks for getting the most out of your Internet applications.

Accessing the Internet Through a Proxy
Describes how to configure proxies.

NetworkInformation
Describes how to gather information about network events, changes, statistics, and properties and also explains how to determine whether a remote host is reachable by using the System.Net.NetworkInformation.Ping class.

Changes to the System.Uri namespace in Version 2.0
Describes several changes made to the System.Uri class in Version 2.0 to fixed incorrect behavior, enhance usability, and enhance security.

International Resource Identifier Support in System.Uri
Describes enhancements to the System.Uri class in Version 3.5, 3.0 SP1, and 2.0 SP1 for International Resource Identifier (IRI) and Internationalized Domain Name (IDN) support.

Socket Performance Enhancements in Version 3.5
Describes a set of enhancements to the System.Net.Sockets.Socket class in Version 3.5, 3.0 SP1, and 2.0 SP1 that provide an alternative asynchronous pattern that can be used by specialized high-performance socket applications.

Peer Name Resolution Protocol
Describes support added in Version 3.5 to support the Peer Name Resolution Protocol (PNRP), a serverless and dynamic name registration and name resolution protocol. These new features are supported by the System.Net.PeerToPeer namespace.

Peer-to-Peer Collaboration
Describes support added in Version 3.5 to support the Peer-to-Peer Collaboration that builds on PNRP. These new features are supported by the System.Net.PeerToPeer.Collaboration namespace.

Changes to NTLM authentication for HttpWebRequest in Version 3.5 SP1
Describes security changes made in Version 3.5 SP1 that affect how integrated Windows authentication is handled by the System.Net.HttpWebRequest, System.Net.HttpListener, System.Net.Security.NegotiateStream, and related classes in the System.Net namespace.

Integrated Windows Authentication with Extended Protection
Describes enhancements for extended protection that affect how integrated Windows authentication is handled by the System.Net.HttpWebRequest, System.Net.HttpListener, System.Net.Mail.SmtpClient, System.Net.Security.SslStream, System.Net.Security.NegotiateStream, and related classes in the System.Net and related namespaces.

NAT Traversal using IPv6 and Teredo
Describes enhancements added to the System.Net, System.Net.NetworkInformation, and System.Net.Sockets namespaces to support NAT traversal using IPv6 and Teredo.

Network Isolation for Windows Store Apps
Describes the impact of network isolation when classes in the System.Net, System.Net.Http, and System.Net.Http.Headers namespaces are used in Windows 8.x Store apps.

Network Programming Samples
Links to downloadable network programming samples that use classes in the System.Net, System.Net.Cache, System.Net.Configuration, System.Net.Mail, System.Net.Mime, System.Net.NetworkInformation, System.Net.PeerToPeer, System.Net.Security, System.Net.Sockets namespaces.

Reference

System.Net
Provides a simple programming interface for many of the protocols used on networks today. The System.Net.WebRequest and System.Net.WebResponse classes in this namespace are the basis for pluggable protocols.

System.Net.Cache
Defines the types and enumerations used to define cache policies for resources obtained using the System.Net.WebRequest and System.Net.HttpWebRequest classes.

System.Net.Configuration
Classes that applications use to programmatically access and update configuration settings for the System.Net namespaces.

System.Net.Http

Classes that provides a programming interface for modern HTTP applications.

System.Net.Http.Headers
Provides support for collections of HTTP headers used by the System.Net.Http namespace

System.Net.Mail
Classes to compose and send mail using the SMTP protocol.

System.Net.Mime
Defines types that are used to represent Multipurpose Internet Mail Exchange (MIME) headers used by classes in the System.Net.Mail namespace.

System.Net.NetworkInformation
Classes to programmatically gather information about network events, changes, statistics, and properties.

System.Net.PeerToPeer
Provides a managed implementation of the Peer Name Resolution Protocol (PNRP) for developers.

System.Net.PeerToPeer.Collaboration
Provides a managed implementation of the Peer-to-Peer Collaboration interface for developers.

System.Net.Security
Classes to provide network streams for secure communications between hosts.

System.Net.Sockets
Provides a managed implementation of the Windows Sockets (Winsock) interface for developers who need to help control access to the network.

System.Net.WebSockets
Provides a managed implementation of the WebSocket interface for developers.

System.Uri
Provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.

System.Security.Authentication.ExtendedProtection
Provides support for authentication using extended protection for applications.

System.Security.Authentication.ExtendedProtection.Configuration
Provides support for configuration of authentication using extended protection for applications.

See also

- Transport Layer Security (TLS) best practices with .NET Framework
- Network Programming How-to Topics
- Network Programming Samples
- Networking Samples for .NET on MSDN Code Gallery
- HttpClient Sample

Configuring Apps by using Configuration Files

8/22/2019 • 4 minutes to read • Edit Online

The .NET Framework, through configuration files, gives developers and administrators control and flexibility over the way applications run. Configuration files are XML files that can be changed as needed. An administrator can control which protected resources an application can access, which versions of assemblies an application will use, and where remote applications and objects are located. Developers can put settings in configuration files, eliminating the need to recompile an application every time a setting changes. This section describes what can be configured and why configuring an application might be useful.

> **NOTE**
>
> Managed code can use the classes in the System.Configuration namespace to read settings from the configuration files, but not to write settings to those files.

This topic describes the syntax of configuration files and provides information about the three types of configuration files: machine, application, and security.

Configuration File Format

Configuration files contain elements, which are logical data structures that set configuration information. Within a configuration file, you use tags to mark the beginning and end of an element. For example, the `<runtime>` element consists of `<runtime>` *child elements* `</runtime>`. An empty element would be written as `<runtime/>` or `<runtime></runtime>`.

As with all XML files, the syntax in configuration files is case-sensitive.

You specify configuration settings using predefined attributes, which are name/value pairs inside an element's start tag. The following example specifies two attributes (`version` and `href`) for the `<codeBase>` element, which specifies where the runtime can locate an assembly (for more information, see Specifying an Assembly's Location).

```
<codeBase version="2.0.0.0"
          href="http://www.litwareinc.com/myAssembly.dll"/>
```

Machine Configuration Files

The machine configuration file, Machine.config, contains settings that apply to an entire computer. This file is located in the %*runtime install path*%\Config directory. Machine.config contains configuration settings for machine-wide assembly binding, built-in remoting channels, and ASP.NET.

The configuration system first looks in the machine configuration file for the **<appSettings>** element and other configuration sections that a developer might define. It then looks in the application configuration file. To keep the machine configuration file manageable, it is best to put these settings in the application configuration file. However, putting the settings in the machine configuration file can make your system more maintainable. For example, if you have a third-party component that both your client and server application uses, it is easier to put the settings for that component in one place. In this case, the machine configuration file is the appropriate place for the settings, so you don't have the same settings in two different files.

> **NOTE**
> Deploying an application using XCOPY will not copy the settings in the machine configuration file.

For more information about how the common language runtime uses the machine configuration file for assembly binding, see How the Runtime Locates Assemblies.

Application Configuration Files

An application configuration file contains settings that are specific to an app. This file includes configuration settings that the common language runtime reads (such as assembly binding policy, remoting objects, and so on), and settings that the app can read.

The name and location of the application configuration file depend on the app's host, which can be one of the following:

- Executable–hosted app.

  These apps have two configuration files: a source configuration file, which is modified by the developer during development, and an output file that is distributed with the app.

  When you develop in Visual Studio, place the source configuration file for your app in the project directory and set its **Copy To Output Directory** property to **Copy always** or **Copy if newer**. The name of the configuration file is the name of the app with a .config extension. For example, an app called myApp.exe should have a source configuration file called myApp.exe.config.

  Visual Studio automatically copies the source configuration file to the directory where the compiled assembly is placed to create the output configuration file, which is deployed with the app. In some cases, Visual Studio may modify the output configuration file; for more information, see the Redirecting assembly versions at the app level section of the Redirecting Assembly Versions article.

- ASP.NET-hosted app.

  For more information about ASP.NET configuration files, see ASP.NET Configuration Settings.

- Internet Explorer-hosted app.

  If an app hosted in Internet Explorer has a configuration file, the location of this file is specified in a `<link>` tag with the following syntax:

  <link rel="*ConfigurationFileName*" href="*location*">

  In this tag, `location` is a URL to the configuration file. This sets the app base. The configuration file must be located on the same website as the app.

Security Configuration Files

Security configuration files contain information about the code group hierarchy and permission sets associated with a policy level. We strongly recommend that you use the Code Access Security Policy tool (Caspol.exe) to modify security policy to ensure that policy changes do not corrupt the security configuration files.

> **NOTE**
> Starting with the .NET Framework 4, the security configuration files are present only if security policy has been changed.

The security configuration files are in the following locations:

- Enterprise policy configuration file: %*runtime-install-path*%\Config\Enterprisesec.config
- Machine policy configuration file: %*runtime-install-path*%\Config\Security.config
- User policy configuration file: %USERPROFILE%\Application data\Microsoft\CLR security config\v*xx.xx*\Security.config

In This Section

How to: Locate Assemblies by Using DEVPATH
Describes how to direct the runtime to use the DEVPATH environment variable when searching for assemblies.

Redirecting Assembly Versions
Describes how to specify the location of an assembly and which version of an assembly to use.

Specifying an Assembly's Location
Describes how to specify where the runtime should search for an assembly.

Configuring Cryptography Classes
Describes how to map an algorithm name to a cryptography class and an object identifier to a cryptography algorithm.

How to: Create a Publisher Policy
Describes when and how you should add a publisher policy file to specify assembly redirection and code base settings.

Configuration File Schema
Describes the schema hierarchy for startup, runtime, network, and other types of configuration settings.

See also

- Configuration File Schema
- Specifying an Assembly's Location
- Redirecting Assembly Versions
- ASP.NET Web Site Administration
- Security Policy Management
- Caspol.exe (Code Access Security Policy Tool)
- Assemblies in the Common Language Runtime

Compiling Apps with .NET Native

7/9/2019 • 2 minutes to read • Edit Online

.NET Native is a precompilation technology for building and deploying Windows apps that is included with Visual Studio 2015 and later versions. It automatically compiles the release version of apps that are written in managed code (C# or Visual Basic) and that target the .NET Framework and Windows 10 to native code.

Typically, apps that target the .NET Framework are compiled to intermediate language (IL). At run time, the just-in-time (JIT) compiler translates the IL to native code. In contrast, .NET Native compiles Windows apps directly to native code. For developers, this means:

- Your apps feature the performance of native code. Usually, performance will be superior to code that is first compiled to IL and then compiled to native code by the JIT compiler.
- You can continue to program in C# or Visual Basic.
- You can continue to take advantage of the resources provided by the .NET Framework, including its class library, automatic memory management and garbage collection, and exception handling.

For users of your apps, .NET Native offers these advantages:

- Faster execution times for the majority of apps and scenarios.
- Faster startup times for the majority of apps and scenarios.
- Low deployment and update costs.
- Optimized app memory usage.

> **IMPORTANT**
> For the vast majority of apps and scenarios, .NET Native offers significantly faster startup times and superior performance when compared to an app compiled to IL or to an NGEN image. However, your results may vary. To ensure that your app has benefited from the performance enhancements of .NET Native, you should compare its performance with that of the non-.NET Native version of your app. For more information, see Performance Session Overview.

But .NET Native involves more than a compilation to native code. It transforms the way that .NET Framework apps are built and executed. In particular:

- During precompilation, required portions of the .NET Framework are statically linked into your app. This allows the app to run with app-local libraries of the .NET Framework, and the compiler to perform global analysis to deliver performance wins. As a result, apps launch consistently faster even after .NET Framework updates.
- The .NET Native runtime is optimized for static precompilation and in the vast majority of cases offers superior performance. At the same time, it retains the core reflection features that developers find so productive.
- .NET Native uses the same back end as the C++ compiler, which is optimized for static precompilation scenarios.

.NET Native is able to bring the performance benefits of C++ to managed code developers because it uses the same or similar tools as C++ under the hood, as shown in this table.

	.NET NATIVE	C++
Libraries	The .NET Framework + Windows Runtime	Win32 + Windows Runtime
Compiler	UTC optimizing compiler	UTC optimizing compiler
Deployed	Ready-to-run binaries	Ready-to-run binaries (ASM)
Runtime	MRT.dll (Minimal CLR Runtime)	CRT.dll (C Runtime)

For Windows apps for Windows 10, you upload .NET Native Code Compilation binaries in app packages (.appx files) to the Windows Store.

In This Section

For more information about developing apps with .NET Native Code Compilation, see these topics:

- Getting Started with .NET Native Code Compilation: The Developer Experience Walkthrough
- .NET Native and Compilation: How .NET Native compiles your project to native code.
- Reflection and .NET Native
  - APIs That Rely on Reflection
  - Reflection API Reference
  - Runtime Directives (rd.xml) Configuration File Reference
- Serialization and Metadata
- Migrating Your Windows Store App to .NET Native
- .NET Native General Troubleshooting

Windows Identity Foundation

4/11/2019 • 2 minutes to read • Edit Online

- What's New in Windows Identity Foundation 4.5
- Windows Identity Foundation 4.5 Overview
  - Claims-Based Identity Model
  - Claims Based Authorization Using WIF
  - WIF Claims Programming Model
- Getting Started With WIF
  - Building My First Claims-Aware ASP.NET Web Application
  - Building My First Claims-Aware WCF Service
- WIF Features
  - Identity and Access Tool for Visual Studio 2012
  - WIF Session Management
  - WIF and Web Farms
  - WSFederation Authentication Module Overview
  - WSTrustChannelFactory and WSTrustChannel
- WIF How-To's Index
  - How To: Build Claims-Aware ASP.NET MVC Web Application Using WIF
  - How To: Build Claims-Aware ASP.NET Web Forms Application Using WIF
  - How To: Build Claims-Aware ASP.NET Application Using Forms-Based Authentication
  - How To: Build Claims-Aware ASP.NET Application Using Windows Authentication
  - How To: Debug Claims-Aware Applications And Services Using WIF Tracing
  - How To: Display Signed In Status Using WIF
  - How To: Enable Token Replay Detection
  - How To: Enable WIF Tracing
  - How To: Enable WIF for a WCF Web Service Application
  - How To: Transform Incoming Claims
- WIF Guidelines
  - Guidelines for Migrating an Application Built Using WIF 3.5 to WIF 4.5
  - Namespace Mapping between WIF 3.5 and WIF 4.5
- WIF Code Sample Index

- WIF Extensions
- WIF API Reference
- WIF Configuration Reference
  - WIF Configuration Schema Conventions

Debugging, Tracing, and Profiling

2/12/2019 • 2 minutes to read • Edit Online

To debug a .NET Framework application, the compiler and runtime environment must be configured to enable a debugger to attach to the application and to produce both symbols and line maps, if possible, for the application and its corresponding Microsoft intermediate language (MSIL). After a managed application has been debugged, it can be profiled to boost performance. Profiling evaluates and describes the lines of source code that generate the most frequently executed code, and how much time it takes to execute them.

.NET Framework applications are easily debugged by using Visual Studio, which handles many of the configuration details. If Visual Studio is not installed, you can examine and improve the performance of .NET Framework applications by using the debugging classes in the .NET Framework System.Diagnostics namespace. This namespace includes the Trace, Debug, and TraceSource classes for tracing execution flow, and the Process, EventLog, and PerformanceCounter classes for profiling code.

In This Section

Enabling JIT-Attach Debugging
Shows how to configure the registry to JIT-attach a debug engine to a .NET Framework application.

Making an Image Easier to Debug
Shows how to turn JIT tracking on and optimization off to make an assembly easier to debug.

Tracing and Instrumenting Applications
Describes how to monitor the execution of your application while it is running, and how to instrument it to display how well it is performing or whether something has gone wrong.

Diagnosing Errors with Managed Debugging Assistants
Describes managed debugging assistants (MDAs), which are debugging aids that work in conjunction with the common language runtime (CLR) to provide information on runtime state.

Enhancing Debugging with the Debugger Display Attributes
Describes how the developer of a type can specify what that type will look like when it is displayed in a debugger.

Performance Counters
Describes the counters that you can use to track the performance of an application.

Related Sections

Debug ASP.NET or ASP.NET Core apps in Visual Studio
Provides prerequisites and instructions for how to debug an ASP.NET application during development or after deployment.

Development Guide
Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.

Deploying the .NET Framework and Applications

8/22/2019 • 5 minutes to read • Edit Online

This article helps you get started deploying the .NET Framework with your application. Most of the information is intended for developers, OEMs, and enterprise administrators. Users who want to install the .NET Framework on their computers should read Installing the .NET Framework.

Key Deployment Resources

Use the following links to other MSDN topics for specific information about deploying and servicing the .NET Framework.

Setup and deployment

- General installer and deployment information:
  - Installer options:
    - Web installer
    - Offline installer
  - Installation modes:
    - Silent installation
    - Displaying a UI
  - Reducing system restarts during .NET Framework 4.5 installations
  - Troubleshoot blocked .NET Framework installations and uninstallations
- Deploying the .NET Framework with a client application (for developers):
  - Using InstallShield in a setup and deployment project
  - Using a Visual Studio ClickOnce application
  - Creating a WiX installation package
  - Using a custom installer
  - Additional information for developers
- Deploying the .NET Framework (for OEMs and administrators):
  - Windows Assessment and Deployment Kit (ADK)
  - Administrator's guide

Servicing

- For general information, see the .NET Framework blog
- Detecting versions
- Detecting service packs and updates

Features That Simplify Deployment

The .NET Framework provides a number of basic features that make it easier to deploy your applications:

- No-impact applications.

  This feature provides application isolation and eliminates DLL conflicts. By default, components do not affect other applications.

- Private components by default.

  By default, components are deployed to the application directory and are visible only to the containing application.

- Controlled code sharing.

  Code sharing requires you to explicitly make code available for sharing instead of being the default behavior.

- Side-by-side versioning.

  Multiple versions of a component or application can coexist, you can choose which versions to use, and the common language runtime enforces versioning policy.

- XCOPY deployment and replication.

  Self-described and self-contained components and applications can be deployed without registry entries or dependencies.

- On-the-fly updates.

  Administrators can use hosts, such as ASP.NET, to update program DLLs, even on remote computers.

- Integration with the Windows Installer.

  Advertisement, publishing, repair, and install-on-demand are all available when deploying your application.

- Enterprise deployment.

  This feature provides easy software distribution, including using Active Directory.

- Downloading and caching.

  Incremental downloads keep downloads smaller, and components can be isolated for use only by the application for low-impact deployment.

- Partially trusted code.

  Identity is based on the code instead of the user, and no certificate dialog boxes appear.

Packaging and Distributing .NET Framework Applications

Some of the packaging and deployment information for the .NET Framework is described in other sections of the documentation. Those sections provide information about the self-describing units called assemblies, which require no registry entries, strong-named assemblies, which ensure name uniqueness and prevent name spoofing, and assembly versioning, which addresses many of the problems associated with DLL conflicts. The following sections provide information about packaging and distributing .NET Framework applications.

Packaging

The .NET Framework provides the following options for packaging applications:

- As a single assembly or as a collection of assemblies.

With this option, you simply use the .dll or .exe files as they were built.

- As cabinet (CAB) files.

  With this option, you compress files into .cab files to make distribution or download less time consuming.

- As a Windows Installer package or in other installer formats.

  With this option, you create .msi files for use with the Windows Installer, or you package your application for use with some other installer.

Distribution

The .NET Framework provides the following options for distributing applications:

- Use XCOPY or FTP.

  Because common language runtime applications are self-describing and require no registry entries, you can use XCOPY or FTP to simply copy the application to an appropriate directory. The application can then be run from that directory.

- Use code download.

  If you are distributing your application over the Internet or through a corporate intranet, you can simply download the code to a computer and run the application there.

- Use an installer program such as Windows Installer 2.0.

  Windows Installer 2.0 can install, repair, or remove .NET Framework assemblies in the global assembly cache and in private directories.

Installation Location

To determine where to deploy your application's assemblies so they can be found by the runtime, see How the Runtime Locates Assemblies.

Security considerations can also affect how you deploy your application. Security permissions are granted to managed code according to where the code is located. Deploying an application or component to a location where it receives little trust, such as the Internet, limits what the application or component can do. For more information about deployment and security considerations, see Code Access Security Basics.

Related Topics

TITLE	DESCRIPTION
How the Runtime Locates Assemblies	Describes how the common language runtime determines which assembly to use to fulfill a binding request.
Best Practices for Assembly Loading	Discusses ways to avoid problems of type identity that can lead to InvalidCastException, MissingMethodException, and other errors.
Reducing System Restarts During .NET Framework 4.5 Installations	Describes the Restart Manager, which prevents reboots whenever possible, and explains how applications that install the .NET Framework can take advantage of it.
Deployment Guide for Administrators	Explains how a system administrator can deploy the .NET Framework and its system dependencies across a network by using System Center Configuration Manager (SCCM).

TITLE	DESCRIPTION
Deployment Guide for Developers	Explains how developers can install .NET Framework on their users' computers with their applications.
Deploying Applications, Services, and Components	Discusses deployment options in Visual Studio, including instructions for publishing an application using the ClickOnce and Windows Installer technologies.
Publishing ClickOnce Applications	Describes how to package a Windows Forms application and deploy it with ClickOnce to client computers on a network.
Packaging and Deploying Resources	Describes the hub and spoke model that the .NET Framework uses to package and deploy resources; covers resource naming conventions, fallback process, and packaging alternatives.
Deploying an Interop Application	Explains how to ship and install interop applications, which typically include a .NET Framework client assembly, one or more interop assemblies representing distinct COM type libraries, and one or more registered COM components.
How to: Get Progress from the .NET Framework 4.5 Installer	Describes how to silently launch and track the .NET Framework setup process while showing your own view of the setup progress.

See also

- Development Guide

.NET Framework Performance

8/22/2019 • 4 minutes to read • Edit Online

If you want to create apps with great performance, you should design and plan for performance just as you would design any other feature of your app. You can use the tools provided by Microsoft to measure your app's performance, and, if needed, make improvements to memory use, code throughput, and responsiveness. This topic lists the performance analysis tools that Microsoft provides, and provides links to other topics that cover performance for specific areas of app development.

Designing and planning for performance

If you want a great performing app, you must design performance into your app just as you would design any other feature. You should determine the performance-critical scenarios in your app, set performance goals, and measure performance for these app scenarios early and often. Because each app is different and has different performance-critical execution paths, determining those paths early and focusing your efforts enable you to maximize your productivity.

You don't have to be completely familiar with your target platform to create a high-performance app. However, you should develop an understanding of which parts of your target platform are costly in terms of performance. You can do this by measuring performance early in your development process.

To determine the areas that are crucial to performance and to establish your performance goals, always consider the user experience. Startup time and responsiveness are two key areas that will affect the user's perception of your app. If your app uses a lot of memory, it may appear sluggish to the user or affect other apps running on the system, or, in some cases, it could fail the Windows Store or Windows Phone Store submission process. Also, if you determine which parts of your code execute more frequently, you can make sure that these portions of your code are well optimized.

Analyzing performance

As part of your overall development plan, set points during development where you will measure the performance of your app and compare the results with the goals you set previously. Measure your app in the environment and hardware that you expect your users to have. By analyzing your app's performance early and often you can change architectural decisions that would be costly and expensive to fix later in the development cycle. The following sections describe performance tools you can use to analyze your apps and discuss event tracing, which is used by these tools.

Performance tools

Here are some of the performance tools you can use with your .NET Framework apps.

TOOL	DESCRIPTION
Visual Studio Performance Analysis	Use to analyze the CPU usage of your .NET Framework apps that will be deployed to computers that are running the Windows operating system. This tool is available from the **Debug** menu in Visual Studio after you open a project. For more information, see Performance Explorer. **Note:** Use Windows Phone Application Analysis (see next row) when targeting Windows Phone.

TOOL	DESCRIPTION
Windows Phone Application Analysis	Use to analyze the CPU and memory, network data transfer rate, app responsiveness, and battery consumption in your Windows Phone apps. This tool is available from the **Debug** menu for a Windows Phone project in Visual Studio after you install the Windows Phone SDK. For more information, see App profiling for Windows Phone 8.
PerfView	Use to identify CPU and memory-related performance issues. This tool uses event tracing for Windows (ETW) and CLR profiling APIs to provide advanced memory and CPU investigations as well as information about garbage collection and JIT compilation. For more information about how to use PerfView, see the tutorial and help files that are included with the app, Channel 9 video tutorials, and blog posts. For memory-specific issues, see Using PerfView for Memory Investigations.
Windows Performance Analyzer	Use to determine overall system performance such as your app's memory and storage use when multiple apps are running on the same computer. This tool is available from the download center as part of the Windows Assessment and Deployment Kit (ADK) for Windows 8. For more information, see Windows Performance Analyzer.

Event tracing for Windows (ETW)

ETW is a technique that lets you obtain diagnostic information about running code and is essential for many of the performance tools mentioned previously. ETW creates logs when particular events are raised by .NET Framework apps and Windows. With ETW, you can enable and disable logging dynamically, so that you can perform detailed tracing in a production environment without restarting your app. The .NET Framework offers support for ETW events, and ETW is used by many profiling and performance tools to generate performance data. These tools often enable and disable ETW events, so familiarity with them is helpful. You can use specific ETW events to collect performance information about particular components of your app. For more information about ETW support in the .NET Framework, see ETW Events in the Common Language Runtime and ETW Events in Task Parallel Library and PLINQ.

Performance by app type

Each type of .NET Framework app has its own best practices, considerations, and tools for evaluating performance. The following table links to performance topics for specific .NET Framework app types.

APP TYPE	SEE
.NET Framework apps for all platforms	Garbage Collection and Performance Performance Tips
Windows 8.x Store apps written in C++, C#, and Visual Basic	Performance best practices for Windows Store apps using C++, C#, and Visual Basic
Windows Presentation Foundation (WPF)	WPF Performance Suite

APP TYPE	SEE
ASP.NET	ASP.NET Performance Overview

Related Topics

TITLE	DESCRIPTION
Caching in .NET Framework Applications	Describes techniques for caching data to improve performance in your app.
Lazy Initialization	Describes how to initialize objects as-needed to improve performance, particularly at app startup.
Reliability	Provides information about preventing asynchronous exceptions in a server environment.
Writing Large, Responsive .NET Framework Apps	Provides performance tips gathered from rewriting the C# and Visual Basic compilers in managed code, and includes several real examples from the C# compiler.

Dynamic Programming in the .NET Framework

4/8/2019 • 2 minutes to read • Edit Online

This section of the documentation provides information about dynamic programming in the .NET Framework.

In This Section

Reflection

Describes how to use reflection to work with objects at run time.

Emitting Dynamic Methods and Assemblies

Describes how to create methods and assemblies at run time by using Reflection.Emit.

Dynamic Language Runtime Overview

Describes the features of the dynamic language runtime.

Dynamic Source Code Generation and Compilation

Describes how to generate and compile dynamic source code.

Related Sections

Development Guide

Managed Extensibility Framework (MEF)

7/24/2019 • 16 minutes to read • Edit Online

This topic provides an overview of the Managed Extensibility Framework that was introduced in the .NET Framework 4.

What is MEF?

The Managed Extensibility Framework or MEF is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.

The Problem of Extensibility

Imagine that you are the architect of a large application that must provide support for extensibility. Your application has to include a potentially large number of smaller components, and is responsible for creating and running them.

The simplest approach to the problem is to include the components as source code in your application, and call them directly from your code. This has a number of obvious drawbacks. Most importantly, you cannot add new components without modifying the source code, a restriction that might be acceptable in, for example, a Web application, but is unworkable in a client application. Equally problematic, you may not have access to the source code for the components, because they might be developed by third parties, and for the same reason you cannot allow them to access yours.

A slightly more sophisticated approach would be to provide an extension point or interface, to permit decoupling between the application and its components. Under this model, you might provide an interface that a component can implement, and an API to enable it to interact with your application. This solves the problem of requiring source code access, but it still has its own difficulties.

Because the application lacks any capacity for discovering components on its own, it must still be explicitly told which components are available and should be loaded. This is typically accomplished by explicitly registering the available components in a configuration file. This means that assuring that the components are correct becomes a maintenance issue, particularly if it is the end user and not the developer who is expected to do the updating.

In addition, components are incapable of communicating with one another, except through the rigidly defined channels of the application itself. If the application architect has not anticipated the need for a particular communication, it is usually impossible.

Finally, the component developers must accept a hard dependency on what assembly contains the interface they implement. This makes it difficult for a component to be used in more than one application, and can also create problems when you create a test framework for components.

What MEF Provides

Instead of this explicit registration of available components, MEF provides a way to discover them implicitly, via *composition*. A MEF component, called a *part*, declaratively specifies both its dependencies (known as *imports*) and what capabilities (known as *exports*) it makes available. When a part is created, the MEF composition engine satisfies its imports with what is available from other parts.

This approach solves the problems discussed in the previous section. Because MEF parts declaratively specify their

capabilities, they are discoverable at runtime, which means an application can make use of parts without either hard-coded references or fragile configuration files. MEF allows applications to discover and examine parts by their metadata, without instantiating them or even loading their assemblies. As a result, there is no need to carefully specify when and how extensions should be loaded.

In addition to its provided exports, a part can specify its imports, which will be filled by other parts. This makes communication among parts not only possible, but easy, and allows for good factoring of code. For example, services common to many components can be factored into a separate part and easily modified or replaced.

Because the MEF model requires no hard dependency on a particular application assembly, it allows extensions to be reused from application to application. This also makes it easy to develop a test harness, independent of the application, to test extension components.

An extensible application written by using MEF declares an import that can be filled by extension components, and may also declare exports in order to expose application services to extensions. Each extension component declares an export, and may also declare imports. In this way, extension components themselves are automatically extensible.

Where Is MEF Available?

MEF is an integral part of the .NET Framework 4, and is available wherever the .NET Framework is used. You can use MEF in your client applications, whether they use Windows Forms, WPF, or any other technology, or in server applications that use ASP.NET.

MEF and MAF

Previous versions of the .NET Framework introduced the Managed Add-in Framework (MAF), designed to allow applications to isolate and manage extensions. The focus of MAF is slightly higher-level than MEF, concentrating on extension isolation and assembly loading and unloading, while MEF's focus is on discoverability, extensibility, and portability. The two frameworks interoperate smoothly, and a single application can take advantage of both.

SimpleCalculator: An Example Application

The simplest way to see what MEF can do is to build a simple MEF application. In this example, you build a very simple calculator named SimpleCalculator. The goal of SimpleCalculator is to create a console application that accepts basic arithmetic commands, in the form "5+3" or "6-2", and returns the correct answers. Using MEF, you'll be able to add new operators without changing the application code.

To download the complete code for this example, see the SimpleCalculator sample.

> **NOTE**
> The purpose of SimpleCalculator is to demonstrate the concepts and syntax of MEF, rather than to necessarily provide a realistic scenario for its use. Many of the applications that would benefit most from the power of MEF are more complex than SimpleCalculator. For more extensive examples, see the Managed Extensibility Framework on GitHub.

- To start, in Visual Studio, create a new Console Application project and name it `SimpleCalculator`.
- Add a reference to the System.ComponentModel.Composition assembly, where MEF resides.
- Open Module1.vb or Program.cs and add `Imports` or `using` statements for System.ComponentModel.Composition and System.ComponentModel.Composition.Hosting. These two namespaces contain MEF types you will need to develop an extensible application.
- If you're using Visual Basic, add the `Public` keyword to the line that declares the `Module1` module.

Composition Container and Catalogs

The core of the MEF composition model is the *composition container*, which contains all the parts available and performs composition. Composition is the matching up of imports to exports. The most common type of composition container is CompositionContainer, and you'll use this for SimpleCalculator.

If you're using Visual Basic, in Module1.vb, add a public class named `Program`.

Add the following line to the `Program` class in Module1.vb or Program.cs:

```
Dim _container As CompositionContainer
```

```
private CompositionContainer _container;
```

In order to discover the parts available to it, the composition containers makes use of a *catalog*. A catalog is an object that makes available parts discovered from some source. MEF provides catalogs to discover parts from a provided type, an assembly, or a directory. Application developers can easily create new catalogs to discover parts from other sources, such as a Web service.

Add the following constructor to the `Program` class:

```
Public Sub New()
    ' An aggregate catalog that combines multiple catalogs.
     Dim catalog = New AggregateCatalog()

    ' Adds all the parts found in the same assembly as the Program class.
    catalog.Catalogs.Add(New AssemblyCatalog(GetType(Program).Assembly))

    ' Create the CompositionContainer with the parts in the catalog.
    _container = New CompositionContainer(catalog)

    ' Fill the imports of this object.
    Try
        _container.ComposeParts(Me)
    Catch ex As CompositionException
        Console.WriteLine(ex.ToString)
    End Try
End Sub
```

```
private Program()
{
    // An aggregate catalog that combines multiple catalogs.
    var catalog = new AggregateCatalog();
    // Adds all the parts found in the same assembly as the Program class.
    catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));

    // Create the CompositionContainer with the parts in the catalog.
    _container = new CompositionContainer(catalog);

    // Fill the imports of this object.
    try
    {
        this._container.ComposeParts(this);
    }
    catch (CompositionException compositionException)
    {
        Console.WriteLine(compositionException.ToString());
   }
}
```

The call to ComposeParts tells the composition container to compose a specific set of parts, in this case the current instance of `Program`. At this point, however, nothing will happen, since `Program` has no imports to fill.

Imports and Exports with Attributes

First, you have `Program` import a calculator. This allows the separation of user interface concerns, such as the console input and output that will go into `Program`, from the logic of the calculator.

Add the following code to the `Program` class:

```
<Import(GetType(ICalculator))>
Public Property calculator As ICalculator
```

```
[Import(typeof(ICalculator))]
public ICalculator calculator;
```

Notice that the declaration of the `calculator` object is not unusual, but that it is decorated with the ImportAttribute attribute. This attribute declares something to be an import; that is, it will be filled by the composition engine when the object is composed.

Every import has a *contract*, which determines what exports it will be matched with. The contract can be an explicitly specified string, or it can be automatically generated by MEF from a given type, in this case the interface `ICalculator`. Any export declared with a matching contract will fulfill this import. Note that while the type of the `calculator` object is in fact `ICalculator`, this is not required. The contract is independent from the type of the importing object. (In this case, you could leave out the `typeof(ICalculator)`. MEF will automatically assume the contract to be based on the type of the import unless you specify it explicitly.)

Add this very simple interface to the module or `SimpleCalculator` namespace:

```
Public Interface ICalculator
    Function Calculate(ByVal input As String) As String
End Interface
```

```
public interface ICalculator
{
    String Calculate(String input);
}
```

Now that you have defined `ICalculator`, you need a class that implements it. Add the following class to the module or `SimpleCalculator` namespace:

```
<Export(GetType(ICalculator))>
Public Class MySimpleCalculator
   Implements ICalculator

End Class
```

```
[Export(typeof(ICalculator))]
class MySimpleCalculator : ICalculator
{

}
```

Here is the export that will match the import in `Program`. In order for the export to match the import, the export must have the same contract. Exporting under a contract based on `typeof(MySimpleCalculator)` would produce a mismatch, and the import would not be filled; the contract needs to match exactly.

Since the composition container will be populated with all the parts available in this assembly, the `MySimpleCalculator` part will be available. When the constructor for `Program` performs composition on the `Program` object, its import will be filled with a `MySimpleCalculator` object, which will be created for that purpose.

The user interface layer (`Program`) does not need to know anything else. You can therefore fill in the rest of the user interface logic in the `Main` method.

Add the following code to the `Main` method:

```
Sub Main()
    ' Composition is performed in the constructor.
    Dim p As New Program()
    Dim s As String
    Console.WriteLine("Enter Command:")
    While (True)
        s = Console.ReadLine()
        Console.WriteLine(p.calculator.Calculate(s))
    End While
End Sub
```

```
static void Main(string[] args)
{
    // Composition is performed in the constructor.
    var p = new Program();
    String s;
    Console.WriteLine("Enter Command:");
    while (true)
    {
        s = Console.ReadLine();
        Console.WriteLine(p.calculator.Calculate(s));
    }
}
```

This code simply reads a line of input and calls the `Calculate` function of `ICalculator` on the result, which it writes back to the console. That is all the code you need in `Program`. All the rest of the work will happen in the parts.

Further Imports and ImportMany

In order for SimpleCalculator to be extensible, it needs to import a list of operations. An ordinary ImportAttribute attribute is filled by one and only one ExportAttribute. If more than one is available, the composition engine produces an error. To create an import that can be filled by any number of exports, you can use the ImportManyAttribute attribute.

Add the following operations property to the `MySimpleCalculator` class:

```
<ImportMany()>
Public Property operations As IEnumerable(Of Lazy(Of IOperation, IOperationData))
```

```
[ImportMany]
IEnumerable<Lazy<IOperation, IOperationData>> operations;
```

Lazy<T,TMetadata> is a type provided by MEF to hold indirect references to exports. Here, in addition to the exported object itself, you also get *export metadata*, or information that describes the exported object. Each Lazy<T,TMetadata> contains an `IOperation` object, representing an actual operation, and an `IOperationData` object, representing its metadata.

Add the following simple interfaces to the module or `SimpleCalculator` namespace:

```
Public Interface IOperation
    Function Operate(ByVal left As Integer, ByVal right As Integer) As Integer
End Interface

Public Interface IOperationData
    ReadOnly Property Symbol As Char
End Interface
```

```
public interface IOperation
{
     int Operate(int left, int right);
}

public interface IOperationData
{
    Char Symbol { get; }
}
```

In this case, the metadata for each operation is the symbol that represents that operation, such as +, -, *, and so on. To make the addition operation available, add the following class to the module or `SimpleCalculator` namespace:

```
<Export(GetType(IOperation))>
<ExportMetadata("Symbol", "+"c)>
Public Class Add
    Implements IOperation

    Public Function Operate(ByVal left As Integer, ByVal right As Integer) As Integer Implements
IOperation.Operate
        Return left + right
    End Function
End Class
```

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '+')]
class Add: IOperation
{
    public int Operate(int left, int right)
    {
        return left + right;
    }
}
```

The ExportAttribute attribute functions as it did before. The ExportMetadataAttribute attribute attaches metadata, in the form of a name-value pair, to that export. While the `Add` class implements `IOperation`, a class that implements `IOperationData` is not explicitly defined. Instead, a class is implicitly created by MEF with properties based on the names of the metadata provided. (This is one of several ways to access metadata in MEF.)

Composition in MEF is *recursive*. You explicitly composed the `Program` object, which imported an `ICalculator` that turned out to be of type `MySimpleCalculator`. `MySimpleCalculator`, in turn, imports a collection of `IOperation` objects, and that import will be filled when `MySimpleCalculator` is created, at the same time as the imports of

`Program`. If the `Add` class declared a further import, that too would have to be filled, and so on. Any import left unfilled results in a composition error. (It is possible, however, to declare imports to be optional or to assign them default values.)

Calculator Logic

With these parts in place, all that remains is the calculator logic itself. Add the following code in the `MySimpleCalculator` class to implement the `Calculate` method:

```
Public Function Calculate(ByVal input As String) As String Implements ICalculator.Calculate
    Dim left, right As Integer
    Dim operation As Char
    ' Finds the operator.
    Dim fn = FindFirstNonDigit(input)
    If fn < 0 Then
        Return "Could not parse command."
    End If
    operation = input(fn)
    Try
        ' Separate out the operands.
        left = Integer.Parse(input.Substring(0, fn))
        right = Integer.Parse(input.Substring(fn + 1))
    Catch ex As Exception
        Return "Could not parse command."
    End Try
    For Each i As Lazy(Of IOperation, IOperationData) In operations
        If i.Metadata.symbol = operation Then
            Return i.Value.Operate(left, right).ToString()
        End If
    Next
    Return "Operation not found!"
End Function
```

```
public String Calculate(String input)
{
    int left;
    int right;
    char operation;
    // Finds the operator.
    int fn = FindFirstNonDigit(input);
    if (fn < 0) return "Could not parse command.";

    try
    {
        // Separate out the operands.
        left = int.Parse(input.Substring(0, fn));
        right = int.Parse(input.Substring(fn + 1));
    }
    catch
    {
        return "Could not parse command.";
    }

    operation = input[fn];

    foreach (Lazy<IOperation, IOperationData> i in operations)
    {
        if (i.Metadata.Symbol.Equals(operation)) return i.Value.Operate(left, right).ToString();
    }
    return "Operation Not Found!";
}
```

The initial steps parse the input string into left and right operands and an operator character. In the `foreach` loop,

every member of the `operations` collection is examined. These objects are of type Lazy<T,TMetadata>, and their metadata values and exported object can be accessed with the Metadata property and the Value property respectively. In this case, if the `Symbol` property of the `IOperationData` object is discovered to be a match, the calculator calls the `Operate` method of the `IOperation` object and returns the result.

To complete the calculator, you also need a helper method that returns the position of the first non-digit character in a string. Add the following helper method to the `MySimpleCalculator` class:

```
Private Function FindFirstNonDigit(ByVal s As String) As Integer
    For i = 0 To s.Length - 1
        If (Not (Char.IsDigit(s(i)))) Then Return i
    Next
    Return -1
End Function
```

```
private int FindFirstNonDigit(string s)
{
    for (int i = 0; i < s.Length; i++)
    {
        if (!(Char.IsDigit(s[i]))) return i;
    }
    return -1;
}
```

You should now be able to compile and run the project. In Visual Basic, make sure that you added the `Public` keyword to `Module1`. In the console window, type an addition operation, such as "5+3", and the calculator returns the results. Any other operator results in the "Operation Not Found!" message.

Extending SimpleCalculator Using A New Class

Now that the calculator works, adding a new operation is easy. Add the following class to the module or `SimpleCalculator` namespace:

```
<Export(GetType(IOperation))>
<ExportMetadata("Symbol", "-"c)>
Public Class Subtract
    Implements IOperation

    Public Function Operate(ByVal left As Integer, ByVal right As Integer) As Integer Implements
IOperation.Operate
        Return left - right
    End Function
End Class
```

```
[Export(typeof(IOperation))]
[ExportMetadata("Symbol", '-')]
class Subtract : IOperation
{
    public int Operate(int left, int right)
    {
        return left - right;
    }
}
```

Compile and run the project. Type a subtraction operation, such as "5-3". The calculator now supports subtraction as well as addition.

Extending SimpleCalculator Using A New Assembly

Adding classes to the source code is simple enough, but MEF provides the ability to look outside an application's own source for parts. To demonstrate this, you will need to modify SimpleCalculator to search a directory, as well as its own assembly, for parts, by adding a DirectoryCatalog.

Add a new directory named `Extensions` to the SimpleCalculator project. Make sure to add it at the project level, and not at the solution level. Then add a new Class Library project to the solution, named `ExtendedOperations`. The new project will compile into a separate assembly.

Open the Project Properties Designer for the ExtendedOperations project and click the **Compile** or **Build** tab. Change the **Build output path** or **Output path** to point to the Extensions directory in the SimpleCalculator project directory (..\SimpleCalculator\Extensions\).

In Module1.vb or Program.cs, add the following line to the `Program` constructor:

```
catalog.Catalogs.Add(New DirectoryCatalog("C:\SimpleCalculator\SimpleCalculator\Extensions"))
```

```
catalog.Catalogs.Add(new DirectoryCatalog("C:\\SimpleCalculator\\SimpleCalculator\\Extensions"));
```

Replace the example path with the path to your Extensions directory. (This absolute path is for debugging purposes only. In a production application, you would use a relative path.) The DirectoryCatalog will now add any parts found in any assemblies in the Extensions directory to the composition container.

In the ExtendedOperations project, add references to SimpleCalculator and System.ComponentModel.Composition. In the ExtendedOperations class file, add an `Imports` or a `using` statement for System.ComponentModel.Composition. In Visual Basic, also add an `Imports` statement for SimpleCalculator. Then add the following class to the ExtendedOperations class file:

```
<Export(GetType(SimpleCalculator.IOperation))>
<ExportMetadata("Symbol", "%"c)>
Public Class Modulo
    Implements IOperation

    Public Function Operate(ByVal left As Integer, ByVal right As Integer) As Integer Implements
IOperation.Operate
        Return left Mod right
    End Function
End Class
```

```
[Export(typeof(SimpleCalculator.IOperation))]
[ExportMetadata("Symbol", '%')]
public class Mod : SimpleCalculator.IOperation
{
    public int Operate(int left, int right)
    {
        return left % right;
    }
}
```

Note that in order for the contract to match, the ExportAttribute attribute must have the same type as the ImportAttribute.

Compile and run the project. Test the new Mod (%) operator.

Conclusion

This topic covered the basic concepts of MEF.

- Parts, catalogs, and the composition container

  Parts and the composition container are the basic building blocks of a MEF application. A part is any object that imports or exports a value, up to and including itself. A catalog provides a collection of parts from a particular source. The composition container uses the parts provided by a catalog to perform composition, the binding of imports to exports.

- Imports and exports

  Imports and exports are the way by which components communicate. With an import, the component specifies a need for a particular value or object, and with an export it specifies the availability of a value. Each import is matched with a list of exports by way of its contract.

Where Do I Go Now?

To download the complete code for this example, see the SimpleCalculator sample.

For more information and code examples, see Managed Extensibility Framework. For a list of the MEF types, see the System.ComponentModel.Composition namespace.

Interoperating with unmanaged code

3/25/2019 • 2 minutes to read • Edit Online

The .NET Framework promotes interaction with COM components, COM+ services, external type libraries, and many operating system services. Data types, method signatures, and error-handling mechanisms vary between managed and unmanaged object models. To simplify interoperation between .NET Framework components and unmanaged code and to ease the migration path, the common language runtime conceals from both clients and servers the differences in these object models.

Code that executes under the control of the runtime is called managed code. Conversely, code that runs outside the runtime is called unmanaged code. COM components, ActiveX interfaces, and Windows API functions are examples of unmanaged code.

In this section

Exposing COM Components to the .NET Framework
Describes how to use COM components from .NET Framework applications.

Exposing .NET Framework Components to COM
Describes how to use .NET Framework components from COM applications.

Consuming Unmanaged DLL Functions
Describes how to call unmanaged DLL functions using platform invoke.

Interop Marshaling
Describes marshaling for COM interop and platform invoke.

How to: Map HRESULTs and Exceptions
Describes the mapping between exceptions and HRESULTs.

COM Wrappers
Describes the wrappers provided by COM interop.

Type Equivalence and Embedded Interop Types
Describes how type information for COM types is embedded in assemblies, and how the common language runtime determines the equivalence of embedded COM types.

How to: Generate Primary Interop Assemblies Using Tlbimp.exe
Describes how to produce primary interop assemblies using *Tlbimp.exe* (Type Library Importer).

How to: Register Primary Interop Assemblies
Describes how to register the primary interop assemblies before you can reference them in your projects.

Registration-Free COM Interop
Describes how COM interop can activate components without using the Windows registry.

How to: Configure .NET Framework-Based COM Components for Registration-Free Activation
Describes how to create an application manifest and how to create and embed a component manifest.

Unmanaged API Reference

2/15/2019 • 2 minutes to read • Edit Online

This section includes information on unmanaged APIs that can be used by managed-code-related applications, such as runtime hosts, compilers, disassemblers, obfuscators, debuggers, and profilers.

In This Section

Common Data Types
Lists the common data types that are used, particularly in the unmanaged profiling and debugging APIs.

ALink
Describes the ALink API, which supports the creation of .NET Framework assemblies and unbound modules.

Authenticode
Supports the Authenticode XrML license creation and verification module.

Constants
Describes the constants that are defined in CorSym.idl.

Custom Interface Attributes
Describes component object model (COM) custom interface attributes.

Debugging
Describes the debugging API, which enables a debugger to debug code that runs in the common language runtime (CLR) environment.

Diagnostics Symbol Store
Describes the diagnostics symbol store API, which enables a compiler to generate symbol information for use by a debugger.

Fusion
Describes the fusion API, which enables a runtime host to access the properties of an application's resources in order to locate the correct versions of those resources for the application.

Hosting
Describes the hosting API, which enables unmanaged hosts to integrate the CLR into their applications.

Metadata
Describes the metadata API, which enables a client such as a compiler to generate or access a component's metadata without the types being loaded by the CLR.

Profiling
Describes the profiling API, which enables a profiler to monitor a program's execution by the CLR.

Strong Naming
Describes the strong naming API, which enables a client to administer strong name signing for assemblies.

WMI and Performance Counters
Describes the APIs that wrap calls to Windows Management Instrumentation (WMI) libraries.

Tlbexp Helper Functions
Describes the two helper functions and interface used by the Type Library Exporter (Tlbexp.exe) during the assembly-to-type-library conversion process.

Related Sections

Development Guide

XAML Services

6/4/2019 • 8 minutes to read • Edit Online

This topic describes the capabilities of a technology set known as .NET Framework XAML Services. The majority of the services and APIs described are located in the assembly System.Xaml, which is an assembly introduced with the .NET Framework 4 set of .NET core assemblies. Services include readers and writers, schema classes and schema support, factories, attributing of classes, XAML language intrinsic support, and other XAML language features.

About This Documentation

Conceptual documentation for .NET Framework XAML Services assumes that you have previous experience with the XAML language and how it might apply to a specific framework, for example Windows Presentation Foundation (WPF) or Windows Workflow Foundation, or a specific technology feature area, for example the build customization features in Microsoft.Build.Framework.XamlTypes. This documentation does not attempt to explain the basics of XAML as a markup language, XAML syntax terminology, or other introductory material. Instead, this documentation focuses on specifically using the .NET Framework XAML Services that are enabled in the System.Xaml assembly library. Most of these APIs are for scenarios of XAML language integration and extensibility. This might include any of the following:

- Extending the capabilities of the base XAML readers or XAML writers (processing the XAML node stream directly; deriving your own XAML reader or XAML writer).
- Defining XAML-usable custom types that do not have specific framework dependencies, and attributing the types to convey their XAML type system characteristics to .NET Framework XAML Services.
- Hosting XAML readers or XAML writers as a component of an application, such as a visual designer or interactive editor for XAML markup sources.
- Writing XAML value converters (markup extensions; type converters for custom types).
- Defining a custom XAML schema context (using alternate assembly-loading techniques for backing type sources; using known-types lookup techniques instead of always reflecting assemblies; using loaded assembly concepts that do not use the CLR `AppDomain` and its associated security model).
- Extending the base XAML type system.
- Using the `Lookup` or `Invoker` techniques to influence the XAML type system and how type backings are evaluated.

If you are looking for introductory material on XAML as a language, you might try XAML Overview (WPF). That topic discusses XAML for an audience that is new both to Windows Presentation Foundation (WPF) and also to using XAML markup and XAML language features. Another useful document is the introductory material in the XAML language specification.

.NET Framework XAML Services and System.Xaml in the .NET Architecture

In previous versions of Microsoft .NET Framework, support for XAML language features was implemented by frameworks that built on Microsoft .NET Framework (Windows Presentation Foundation (WPF), Windows Workflow Foundation and Windows Communication Foundation (WCF)), and therefore varied in its behavior and the API used depending on which specific framework you were using. This included the XAML parser and its object

graph creation mechanism, XAML language intrinsics, serialization support, and so on.

In .NET Framework 4, .NET Framework XAML Services and the System.Xaml assembly define much of what is needed for supporting XAML language features. This includes base classes for XAML readers and XAML writers. The most important feature added to .NET Framework XAML Services that was not present in any of the framework-specific XAML implementations is a type system representation for XAML. The type system representation presents XAML in an object-oriented way that centers on XAML capabilities without taking dependencies on specific capabilities of frameworks.

The XAML type system is not limited by the markup form or run-time specifics of the XAML origin; nor is it limited by any specific backing type system. The XAML type system includes object representations for types, members, XAML schema contexts, XML-level concepts, and other XAML language concepts or XAML intrinsics. Using or extending the XAML type system makes it possible to derive from classes like XAML readers and XAML writers, and extend the functionality of XAML representations into specific features enabled by a framework, a technology, or an application that consumes or emits XAML. The concept of a XAML schema context enables practical object graph write operations from the combination of a XAML object writer implementation, a technology's backing type system as communicated through assembly information in the context, and the XAML node source. For more information on the XAML schema concept. see Default XAML Schema Context and WPF XAML Schema Context.

XAML Node Streams, XAML Readers, and XAML Writers

To understand the role that .NET Framework XAML Services plays in the relationship between the XAML language and specific technologies that use XAML as a language, it is helpful to understand the concept of a XAML node stream and how that concept shapes the API and terminology. The XAML node stream is a conceptual intermediate between a XAML language representation and the object graph that the XAML represents or defines.

- A XAML reader is an entity that processes XAML in some form, and produces a XAML node stream. In the API, a XAML reader is represented by the base class XamlReader.
- A XAML writer is an entity that processes a XAML node stream and produces something else. In the API, a XAML writer is represented by the base class XamlWriter.

The two most common scenarios involving XAML are loading XAML to instantiate an object graph, and saving an object graph from an application or tool and producing a XAML representation (typically in markup form saved as text file). Loading XAML and creating an object graph is often referred to in this documentation as the load path. Saving or serializing an existing object graph to XAML is often referred to in this documentation as the save path.

The most common type of load path can be described as follows:

- Start with a XAML representation, in UTF-encoded XML format and saved as a text file.
- Load that XAML into XamlXmlReader. XamlXmlReader is a XamlReader subclass.
- The result is a XAML node stream. You can access individual nodes of the XAML node stream using XamlXmlReader / XamlReader API. The most typical operation here is to advance through the XAML node stream, processing each node using a "current record" metaphor.
- Pass the resulting nodes from the XAML node stream to a XamlObjectWriter API. XamlObjectWriter is a XamlWriter subclass.
- The XamlObjectWriter writes an object graph, one object at a time, in accordance to progress through the source XAML node stream. This is done with the assistance of a XAML schema context and an implementation that can access the assemblies and types of a backing type system and framework.
- Call Result at the end of the XAML node stream to obtain the root object of the object graph.

The most common type of save path can be described as follows:

- Start with the object graph of an entire application run time, the UI content and state of a run time, or a smaller segment of an overall application's object representation at run time.
- From a logical start object, such as an application root or document root, load the objects into XamlObjectReader. XamlObjectReader is a XamlReader subclass.
- The result is a XAML node stream. You can access individual nodes of the XAML node stream using XamlObjectReader and XamlReader API. The most typical operation here is to advance through the XAML node stream, processing each node using a "current record" metaphor.
- Pass the resulting nodes from the XAML node stream to a XamlXmlWriter API. XamlXmlWriter is a XamlWriter subclass.
- The XamlXmlWriter writes XAML in an XML UTF encoding. You can save this as a text file, as a stream, or in other forms.
- Call Flush to obtain the final output.

For more information about XAML node stream concepts, see Understanding XAML Node Stream Structures and Concepts.

The XamlServices Class

It is not always necessary to deal with a XAML node stream. If you want a basic load path or a basic save path, you can use APIs in the XamlServices class.

- Various signatures of Load implement a load path. You can either load a file or stream, or can load an XmlReader, TextReader or XamlReader that wrap your XAML input by loading with that reader's APIs.
- Various signatures of Save save an object graph and produce output as a stream, file, or XmlWriter/TextWriter instance.
- Transform converts XAML by linking a load path and a save path as a single operation. A different schema context or different backing type system could be used for XamlReader and XamlWriter, which is what influences how the resulting XAML is transformed.

For more information about how to use XamlServices, see XAMLServices Class and Basic XAML Reading or Writing.

XAML Type System

The XAML type system provides the APIs that are required to work with a given individual node of a XAML node stream.

XamlType is the representation for an object - what you are processing between a start object node and end object node.

XamlMember is the representation for a member of an object - what you are processing between a start member node and end member node.

APIs such as GetAllMembers and GetMember and DeclaringType report the relationships between a XamlType and XamlMember.

The default behavior of the XAML type system as implemented by .NET Framework XAML Services is based on the common language runtime (CLR), and static analysis of CLR types in assemblies by using reflection. Therefore, for a specific CLR type, the default implementation of the XAML type system can expose the XAML schema of that type and its members and report it in terms of the XAML type system. In the default XAML type system, the concept of assignability of types is mapped onto CLR inheritance, and the concepts of instances, value types and so on are also mapped to the supporting behaviors and features of the CLR.

Reference for XAML Language Features

To support XAML, .NET Framework XAML Services provides specific implementation of XAML language concepts as defined for the XAML language XAML namespace. These are documented as specific reference pages. The language features are documented from the perspective of how these language features behave when they are processed by a XAML reader or XAML writer that is defined by .NET Framework XAML Services. For more information, see XAML Namespace (x:) Language Features.

.NET Framework Tools

8/22/2019 • 5 minutes to read • Edit Online

The .NET Framework tools make it easier for you to create, deploy, and manage applications and components that target the .NET Framework.

Most of the .NET Framework tools described in this section are automatically installed with Visual Studio. To download Visual Studio, visit the Visual Studio Downloads page.

You can run all the tools from the command line with the exception of the Assembly Cache Viewer (Shfusion.dll). You must access Shfusion.dll from File Explorer.

The best way to run the command-line tools is by using the Developer Command Prompt for Visual Studio. These utilities enable you to run the tools easily, without navigating to the installation folder. For more information, see Command Prompts.

> **NOTE**
> Some tools are specific to either 32-bit computers or 64-bit computers. Be sure to run the appropriate version of the tool for your computer.

In This Section

Al.exe (Assembly Linker)
Generates a file that has an assembly manifest from modules or resource files.

Aximp.exe (Windows Forms ActiveX Control Importer)
Converts type definitions in a COM type library for an ActiveX control into a Windows Forms control.

Caspol.exe (Code Access Security Policy Tool)
Enables you to view and configure security policy for the machine policy level, the user policy level, and the enterprise policy level. In the .NET Framework 4 and later, this tool does not affect code access security (CAS) policy unless the <legacyCasPolicy> element is set to `true`. For more information, see Security Changes.

Cert2spc.exe (Software Publisher Certificate Test Tool)
Creates a Software Publisher's Certificate (SPC) from one or more X.509 certificates. This tool is for testing purposes only.

Certmgr.exe (Certificate Manager Tool)
Manages certificates, certificate trust lists (CTLs), and certificate revocation lists (CRLs).

Clrver.exe (CLR Version Tool)
reports all the installed versions of the common language runtime (CLR) on the computer.

CorFlags.exe (CorFlags Conversion Tool)
Lets you configure the CorFlags section of the header of a portable executable (PE) image.

Fuslogvw.exe (Assembly Binding Log Viewer)
Displays information about assembly binds to help you diagnose why the .NET Framework cannot locate an assembly at run time.

Gacutil.exe (Global Assembly Cache Tool)
Lets you view and manipulate the contents of the global assembly cache and download cache.

Ilasm.exe (IL Assembler)

Generates a portable executable (PE) file from intermediate language (IL). You can run the resulting executable to determine whether the IL performs as expected.

Ildasm.exe (IL Disassembler)

Takes a portable executable (PE) file that contains intermediate language (IL) code and creates a text file that can be input to the IL Assembler (Ilasm.exe).

Installutil.exe (Installer Tool)

Enables you to install and uninstall server resources by executing the installer components in a specified assembly. (Works with classes in the System.Configuration.Install namespace.)

Lc.exe (License Compiler)

Reads text files that contain licensing information and produces a .licenses file that can be embedded in a common language runtime executable as a resource.

Mage.exe (Manifest Generation and Editing Tool)

Lets you create, edit, and sign application and deployment manifests. As a command-line tool, Mage.exe can be run from both batch scripts and other Windows-based applications, including ASP.NET applications.

MageUI.exe (Manifest Generation and Editing Tool, Graphical Client)

Supports the same functionality as the command-line tool Mage.exe, but uses a Windows-based user interface (UI). Supports the same functionality as the command-line tool Mage.exe, but uses a Windows-based user interface (UI).

MDbg.exe (.NET Framework Command-Line Debugger)

Helps tools vendors and application developers find and fix bugs in programs that target the .NET Framework common language runtime. This tool uses the runtime debugging API to provide debugging services.

Mgmtclassgen.exe (Management Strongly Typed Class Generator)

Enables you to generate an early-bound managed class for a specified Windows Management Instrumentation (WMI) class.

Mpgo.exe (Managed Profile Guided Optimization Tool)

Enables you to tune native image assemblies using common end-user scenarios. Mpgo.exe allows the generation and consumption of profile data for native image application assemblies (not the .NET Framework assemblies) using training scenarios selected by the application developer.

Ngen.exe (Native Image Generator)

Improves the performance of managed applications through the use of native images (files containing compiled processor-specific machine code). The runtime can use native images from the cache instead of using the just-in-time (JIT) compiler to compile the original assembly.

Peverify.exe (PEVerify Tool)

Helps you verify whether your Microsoft intermediate language (MSIL) code and associated metadata meet type safety requirements. Helps you verify whether your Microsoft intermediate language (MSIL) code and associated metadata meet type safety requirements.

Regasm.exe (Assembly Registration Tool)

Reads the metadata within an assembly and adds the necessary entries to the registry. This enables COM clients to appear as .NET Framework classes.

Regsvcs.exe (.NET Services Installation Tool)

Loads and registers an assembly, generates and installs a type library into a specified COM+ version 1.0 application, and configures services that you have added programmatically to a class.

Resgen.exe (Resource File Generator)

Converts text (.txt or .restext) files and XML-based resource format (.resx) files to common language runtime

binary (.resources) files that can be embedded in a runtime binary executable or compiled into satellite assemblies.

SecAnnotate.exe (.NET Security Annotator Tool)
Identifies the SecurityCritical and SecuritySafeCritical portions of an assembly. Identifies the `SecurityCritical` and `SecuritySafeCritical` portions of an assembly.

SignTool.exe (Sign Tool)
Digitally signs files, verifies signatures in files, and time-stamps files.

Sn.exe (Strong Name Tool)
Helps create assemblies with strong names. This tool provides options for key management, signature generation, and signature verification.

SOS.dll (SOS Debugging Extension)
Helps you debug managed programs in the WinDbg.exe debugger and in Visual Studio by providing information about the internal common language runtime environment.

SqlMetal.exe (Code Generation Tool)
Generates code and mapping for the LINQ to SQL component of the .NET Framework.

Storeadm.exe (Isolated Storage Tool)
Manages isolated storage; provides options for listing the user's stores and deleting them.

Tlbexp.exe (Type Library Exporter)
Generates a type library that describes the types that are defined in a common language runtime assembly.

Tlbimp.exe (Type Library Importer)
Converts the type definitions found in a COM type library into equivalent definitions in a common language runtime assembly.

Winmdexp.exe (Windows Runtime Metadata Export Tool)
Exports a .NET Framework assembly that is compiled as a .winmdobj file into a Windows Runtime component, which is packaged as a .winmd file that contains both Windows Runtime metadata and implementation information.

Winres.exe (Windows Forms Resource Editor)
Helps you localize user interface (UI) resources (.resx or .resources files) that are used by Windows Forms. You can translate strings, and then size, move, and hide controls to accommodate the localized strings.

Related Sections

WPF Tools
Includes tools such as the isXPS Conformance tool (isXPS.exe) and performance profiling tools.

Windows Communication Foundation Tools
Includes tools that make it easier for you to create, deploy, and manage Windows Communication Foundation (WCF) applications.

Additional class libraries and APIs

2/3/2019 • 2 minutes to read • Edit Online

The .NET Framework is constantly evolving. To improve cross-platform development and introduce new functionality early, new features are released out of band (OOB). This topic lists the OOB projects that we provide documentation for.

In addition, some libraries target specific platforms or implementations of the .NET Framework. For example, the CodePagesEncodingProvider class makes code page encodings available to UWP apps developed using the .NET Framework. This topic lists these libraries as well.

OOB projects

PROJECT	DESCRIPTION
System.Collections.Immutable	Provides collections that are thread safe and guaranteed to never change their contents.
WinHttpHandler	Provides a message handler for HttpClient based on the WinHTTP interface of Windows.
System.Numerics	Provides a library of vector types that can take advantage of SIMD hardware-based acceleration.
System.Threading.Tasks.Dataflow	The TPL Dataflow Library provides dataflow components to help increase the robustness of concurrency-enabled applications.

Platform-specific libraries

PROJECT	DESCRIPTION
CodePagesEncodingProvider	Extends the EncodingProvider class to make code page encodings available to apps that target the Universal Windows Platform.

Private APIs

These APIs support the product infrastructure and are not intended/supported to be used directly from your code.

API NAME
System.Net.Connection Class
System.Net.Connection.m_WriteList Field
System.Net.ConnectionGroup Class
System.Net.ConnectionGroup.m_ConnectionList Field

API NAME
System.Net.CoreResponseData Class
System.Net.CoreResponseData.m_ResponseHeaders Field
System.Net.CoreResponseData.m_StatusCode Field
System.Net.HttpWebRequest._AutoRedirects Field
System.Net.HttpWebRequest._CoreResponse Field
System.Net.HttpWebRequest._HttpResponse Field
System.Net.ServicePoint.m_ConnectionGroupList Field
System.Net.ServicePointManager.s_ServicePointTable Field
System.Windows.Diagnostics.VisualDiagnostics.s_isDebuggerCheckDisabledForTestPurposes Field
System.Windows.Forms.Design.DataMemberFieldEditor Class
System.Windows.Forms.Design.DataMemberListEditor Class

See also

- The .NET Framework and Out-of-Band Releases

C# Guide

8/19/2019 • 3 minutes to read • Edit Online

The C# guide provides many resources about the C# language. This site has many different audiences. Depending on your experience with programming, or with the C# language and .NET, you may wish to explore different sections of this guide.

- For brand-new developers:
  - Start with the Introduction to C# tutorials. These tutorials let you explore the C# language interactively in your browser. From there, you can move on to other tutorials. These tutorials show you how to create C# programs from scratch. The tutorials provide a step-by-step process to create programs. They show the language concepts and how to build C# programs on your own. If you prefer reading overview information first, try the tour of the C# language. It explains the concepts of the C# language. After reading this, you'll have a basic understanding of the language, and be ready to try the tutorials, or build something on your own.
- For developers new to C#:
  - If you've done development before, but are new to C#, read the tour of the C# language. It covers the basic syntax and structure for the language, and you can use the language tour to contrast C# with other languages you've used. You can also browse the tutorials to try basic C# programs.
- Experienced C# developers:
  - If you've used C# before, you should start by reading what's in the latest version of the language. Check out What's new in C# for the new features in the current version.

How the C# guide is organized

There are several sections in the C# Guide. You can read them in order, or jump directly to what interests you the most. Some of the sections are heavily focused on the language. Others provide end-to-end scenarios that demonstrate a few of the types of programs you can create using C# and the .NET Framework.

- Get Started
  - This section covers what you need to install for a C# development environment on your preferred platform. The different topics under this section explain how to create your first C# program in different supported environments.
- Introduction to C# tutorials
  - Introduction to C# tutorials presents interactive tutorials for brand-new developers to explore and learn the C# language in the browser using a Read-Eval-Print Loop (REPL) interface. After you finish the interactive lessons, you can improve your coding skills by practicing the same lessons on your machine.
- Tutorials
  - This section provides a variety of end-to-end scenarios, including descriptions and code. It shows why certain idioms are preferred, what C# features work best in different situations, and reference implementations for common tasks. If you learn best by seeing the code, start in this section. You can also download all the code and experiment in your own environment.
- Tour of C#
  - This section provides an overview of the language. It covers the elements that make up C# programs and the capabilities of the language. It shows small samples of all the syntax elements of C# and

discussions of the major C# language topics.

- What's new in C#
  - Provides overviews of new features added in the latest language releases and of the history of the C# language.
- C# Programming Guide
  - Provides information and practical examples about how to use C# language constructs.
- Walkthroughs
  - Provides links to programming walkthroughs that use C# and a brief description of each walkthrough.
- Language Reference
  - This section contains the reference material on the C# language. This material helps you understand the syntax and semantics of C#. It also includes reference material on types, operators, attributes, preprocessor directives, compiler switches, compiler errors, and compiler warnings.
- C# Language Specification
  - Links to the latest versions of the C# language specification.

See also

- Getting Started with Visual C# and Visual Basic
- .NET Guide
- C# Samples

Get started with C#

8/19/2019 • 2 minutes to read • Edit Online

This section provides short, simple tutorials that let you quickly build an application using C# and .NET Core. There are getting started topics for Visual Studio 2017 and Visual Studio Code. These articles assume some programming experience. If you are new to programming, try our introduction to C# interactive tutorials.

The following topics are available:

- Introduction to the C# Language and the .NET Framework

  Provides an overview of the C# language and .NET.

- Building a C# Hello World application with .NET Core in Visual Studio 2017

  Visual Studio 2017, the latest release of Visual Studio, lets you code, compile, run, debug, profile, and publish your applications from an integrated development environment for Windows.

  The topic lets you create and run a simple Hello World application and then modify it to run a slightly more interactive Hello World application. Once you've finished building and running your application, you can also learn how to debug it and how to publish it so that it can be run on any platform supported by .NET Core.

- Building a class library with C# and .NET Core in Visual Studio 2017

  A class library lets you define types and type members that can be called from another application. This topic lets you create a class library with a single method that determines whether a string begins with an uppercase character. Once you've finished building the library, you can develop a unit test to ensure that it works as expected, and then you can make it available to applications that want to consume it.

- Get started with C# and Visual Studio Code

  Visual Studio Code is a free code editor optimized for building and debugging modern web and cloud applications. It supports IntelliSense and is available for Linux, macOS, and Windows.

  This topic shows you how to create and run a simple Hello World application with Visual Studio Code and .NET Core.

Related sections

- C# Programming Guide

  Provides information about C# programming concepts, and describes how to perform various tasks in C#.

- C# Reference

  Provides detailed reference information about C# keywords, operators, preprocessor directives, compiler options, and compiler errors and warnings.

- Walkthroughs

  Provides links to programming walkthroughs that use C# and a brief description of each walkthrough.

See also

- ○ C# Development with Visual Studio

C# Tutorials

5/13/2019 • 3 minutes to read • Edit Online

Welcome to the C# tutorials. These start with interactive lessons that you can run in your browser. Later tutorials and more advanced tutorials help you work with the .NET development tools to create C# programs on your machine.

Introduction to C# interactive tutorials

The first lessons explain C# concepts using small snippets of code. You'll learn the basics of C# syntax and how to work with data types like strings, numbers, and booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language.

Hello world

In the Hello world tutorial, you'll create the most basic C# program. You'll explore the `string` type and how to work with text.

Numbers in C#

In the Numbers in C# tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the Hello world lesson.

Branches and loops

The Branches and loops tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the Hello world and Numbers in C# lessons.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the lessons listed above.

Introduction to C# -- work locally

All the introductory tutorials following the Hello World lesson are available in your own local development environment. At the end of each tutorial, you decide if you want to continue with the next lesson online or on your own machine. There are links to help you setup your environment and continue with the next tutorial on your machine.

Explore new features in C#

- Try new features in C# 6 interactively: Explore the features added in C# 6 interactively in your browser.
- String Interpolation: demonstrates how to use string interpolation to create formatted strings in C#.
- Nullable reference types: demonstrates how to use nullable reference types to express your intent for null references.
- Update a project to make use of Nullable reference types: demonstrates techniques to upgrade an existing project to make use of nullable reference types.
- Extend data capabilities using pattern matching: demonstrates how to use pattern matching to extend types beyond their core features.
- Work with data sequences using indexes and ranges: Demonstrates new convenient syntax for accessing single elements or ranges of a sequential data container.

General Tutorials

The following tutorials enable you to build C# programs using .NET Core:

- Console Application: demonstrates Console I/O, the structure of a Console application, and the basics of the task-based asynchronous programming model.
- REST Client: demonstrates web communications, JSON serialization, and object-oriented features in the C# language.
- Inheritance in C# and .NET: demonstrates inheritance in C#, including the use of inheritance to define base classes, abstract base classes, and derived classes.
- Working with LINQ: demonstrates many of the features of LINQ and the language elements that support it.
- Using Attributes: demonstrates how to create and use attributes in C#.
- String interpolation tutorial shows you how to insert values into a string. You'll learn how to create an interpolated string with embedded C# expressions and how to control the text appearance of the expression results in the result string. This tutorial is also available to run locally on your machine.

Introduction to C#

8/24/2019 • 2 minutes to read • Edit Online

Welcome to the introduction to C# tutorials. These start with interactive lessons that you can run in your browser.

The first lessons explain C# concepts using small snippets of code. You'll learn the basics of C# syntax and how to work with data types like strings, numbers, and booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language.

All the introductory tutorials following the Hello World lesson are available using the online browser experience or in your own local development environment. At the end of each tutorial, you decide if you want to continue with the next lesson online or on your own machine. There are links to help you set up your environment and continue with the next tutorial on your machine.

Hello world

In the Hello world tutorial, you'll create the most basic C# program. You'll explore the `string` type and how to work with text.

Numbers in C#

In the Numbers in C# tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the Hello world lesson.

Branches and loops

The Branches and loops tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the Hello world and Numbers in C# lessons.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists. This tutorial is also available to run locally on your machine.

This tutorial assumes that you have finished the lessons listed above.

Introduction to classes

This final tutorial is only available to run on your machine, using your own local development environment and .NET Core. You'll build a console application and see the basic object-oriented features that are part of the C# language.

This tutorial assumes you've finished the online introductory tutorials, and you've installed .NET Core SDK and Visual Studio Code.

Become familiar with the .NET development tools

4/4/2019 • 2 minutes to read • Edit Online

The first step in running a tutorial on your machine is to set up a development environment. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux.

Alternatively, you can install the .NET Core SDK and Visual Studio Code.

Basic application development flow

You'll create applications using the `dotnet new` command. This command generates the files and assets necessary for your application. The introduction to C# tutorials all use the `console` application type. Once you've got the basics, you can expand to other application types.

The other commands you'll use are `dotnet build` to build the executable, and `dotnet run` to run the executable.

Pick your tutorial

You can start with any of the following tutorials:

Numbers in C#

In the Numbers in C# tutorial, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding and how to perform mathematical calculations using C#.

This tutorial assumes that you have finished the Hello world lesson.

Branches and loops

The Branches and loops tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions.

This tutorial assumes that you have finished the Hello world and Numbers in C# lessons.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists.

This tutorial assumes that you have finished the lessons listed above.

Introduction to classes

This final introduction to C# tutorial is only available to run on your machine, using your own local development environment and .NET Core. You'll build a console application and see the basic object-oriented features that are part of the C# language.

Manipulate integral and floating point numbers in C#

7/9/2019 • 8 minutes to read • Edit Online

This tutorial teaches you about the numeric types in C# interactively. You'll write small amounts of code, then you'll compile and run that code. The tutorial contains a series of lessons that explore numbers and math operations in C#. These lessons teach you the fundamentals of the C# language.

This tutorial expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the Become familiar with the development tools with links to more details.

Explore integer math

Create a directory named **numbers-quickstart**. Make that the current directory and run `dotnet new console -n NumbersInCSharp -o .`.

Open **Program.cs** in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following:

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You've just seen one of the fundamental math operations with integers. The `int` type represents an **integer**, a positive or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

- `-` for subtraction
- `*` for multiplication
- `/` for division

Start by exploring those different operations. Add these lines after the line that writes the value of `c`:

```
c = a - b;
Console.WriteLine(c);
c = a * b;
Console.WriteLine(c);
c = a / b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You can also experiment by performing multiple mathematics operations in the same line, if you'd like. Try `c = a + b - 12 * 17;` for example. Mixing variables and constant numbers is allowed.

> **TIP**
> As you explore C# (or any programming language), you'll make mistakes when you write code. The **compiler** will find those errors and report them to you. When the output contains error messages, look closely at the example code and the code in your window to see what to fix. That exercise will help you learn the structure of C# code.

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `WorkingWithIntegers` and write a new `Main` method that calls `WorkingWithIntegers`. When you have finished, your code should look like this:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;
            int c = a + b;
            Console.WriteLine(c);
            c = a - b;
            Console.WriteLine(c);
            c = a * b;
            Console.WriteLine(c);
            c = a / b;
            Console.WriteLine(c);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();
        }
    }
}
```

Explore order of operations

Comment out the call to `WorkingWithIntegers()`. It will make the output less cluttered as you work in this section:

```
//WorkingWithIntegers();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler does not generate any executable code from comments.

The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that by adding the following code to your `Main` method, and executing `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

The output demonstrates that the multiplication is performed before the addition.

You can force a different order of operation by adding parentheses around the operation or operations you want performed first. Add the following lines and run again:

```
d = (a  + b) * c;
Console.WriteLine(d);
```

Explore more by combining many different operations. Add something like the following lines at the bottom of your `Main` method. Try `dotnet run` again.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

You may have noticed an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

If you haven't seen this behavior, try the following code at the end of your `Main` method:

```
int e = 7;
int f = 4;
int g = 3;
int h = (e  + f) / g;
Console.WriteLine(h);
```

Type `dotnet run` again to see the results.

Before moving on, let's take all the code you've written in this section and put it in a new method. Call that new method `OrderPrecedence`. You should end up with something like this:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;
            int c = a + b;
            Console.WriteLine(c);
            c = a - b;
            Console.WriteLine(c);
            c = a * b;
            Console.WriteLine(c);
            c = a / b;
            Console.WriteLine(c);
        }

        static void OrderPrecedence()
        {
            int a = 5;
            int b = 4;
            int c = 2;
            int d = a + b * c;
            Console.WriteLine(d);

            d = (a  + b) * c;
            Console.WriteLine(d);

            d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
            Console.WriteLine(d);

            int e = 7;
            int f = 4;
            int g = 3;
            int h = (e  + f) / g;
            Console.WriteLine(h);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();

            OrderPrecedence();

        }
    }
}
```

Explore integer precision and limits

That last sample showed you that integer division truncates the result. You can get the **remainder** by using the **modulo** operator, the `%` character. Try the following code in your `Main` method:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a  + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Add this code to your `Main` method to see those limits:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

If a calculation produces a value that exceeds those limits, you have an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. Add these two lines to your `Main` method to see an example:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflowed** the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

There are other numeric types with different limits and precision that you would use when the `int` type doesn't meet your needs. Let's explore those next.

Once again, let's move the code you wrote in this section into a separate method. Name it `TestLimits`.

Work with the double type

The `double` numeric type represents a double-precision floating point number. Those terms may be new to you. A **floating point** number is useful to represent non-integral numbers that may be very large or small in magnitude. **Double-precision** means that these numbers are stored using greater precision than **single-precision**. On modern computers, it is more common to use double precision than single precision numbers. Let's explore. Add the following code and see the result:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a  + b) / c;
Console.WriteLine(d);
```

Notice that the answer includes the decimal portion of the quotient. Try a slightly more complicated expression with doubles:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e  + f) / g;
Console.WriteLine(h);
```

The range of a double value is much greater than integer values. Try the following code below what you've written so far:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

These values are printed out in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10.

Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

You know that `0.3` repeating is not exactly the same as `1/3`.

Challenge

Try other calculations with large numbers, small numbers, multiplication and division using the `double` type. Try more complicated calculations.

After you've spent some time with the challenge, take the code you've written and place it in a new method. Name that new method `WorkWithDoubles`.

Work with fixed point types

You've seen the basic numeric types in C#: integers and doubles. There is one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`. The term **fixed point** means that the decimal point (or binary point) doesn't move. Let's take a look:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notice that the range is smaller than the `double` type. You can see the greater precision with the decimal type by trying the following code:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type.

Notice that the math using the decimal type has more digits to the right of the decimal point.

Challenge

Now that you've seen the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: .NET contains a constant for PI, Math.PI that you can use for that value.

You should get an answer between 19 and 20. You can check your answer by looking at the finished sample code on GitHub

Try some other formulas if you'd like.

You've completed the "Numbers in C#" quickstart. You can continue with the Branches and loops quickstart in your own development environment.

You can learn more about numbers in C# in the following topics:

- Integral types

- Floating-Point Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

Learn conditional logic with branch and loop statements

8/19/2019 • 8 minutes to read • Edit Online

This tutorial teaches you how to write code that examines variables and changes the execution path based on those variables. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

This tutorial expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the Become familiar with the development tools with links to more details.

Make decisions using the `if` statement

Create a directory named **branches-tutorial**. Make that the current directory and run `dotnet new console -n BranchesAndLoops -o .`. This command creates a new .NET Core console application in the current directory.

Open **Program.cs** in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code:

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Try this code by typing `dotnet run` in your console window. You should see the message "The answer is greater than 10." printed to your console.

Modify the declaration of `b` so that the sum is less than 10:

```
int b = 3;
```

Type `dotnet run` again. Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you've only written one of the possible branches for an `if` statement: the true branch.

> **TIP**
> As you explore C# (or any programming language), you'll make mistakes when you write code. The compiler will find and report the errors. Look closely at the error output and the code that generated the error. The compiler error can usually help you find the problem.

This first sample shows the power of `if` and Boolean types. A *Boolean* is a variable that can have one of two values: `true` or `false`. C# defines a special type, `bool` for Boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it is skipped.

This process of checking conditions and executing statements based on those conditions is very powerful.

Make if and else work together

To execute different code in both the true and false branches, you create an `else` branch that executes when the condition is false. Try this. Add the last two lines in the code below to your `Main` method (you should already have the first four):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with Boolean conditions provides all the power you need to handle both a `true` and a `false` condition.

> **IMPORTANT**
> The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword will be executed based on the condition. All the samples in this tutorial follow a common practice to indent lines based on the control flow of statements.

Because indentation is not significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses. The following example is the same as the one you just created. Modify your code above to match the following code:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **TIP**
> Through the rest of this tutorial, the code samples all include the braces, following accepted practices.

You can test more complicated conditions. Add the following code in your `Main` method after the code you've written so far:

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

The `==` symbol tests for *equality*. Using `==` distinguishes the test for equality from assignment, which you saw in `a = 5`.

The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`.

You can also use `||` to represent "or". Add the following code after what you've written so far:

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modify the values of `a`, `b`, and `c` and switch between `&&` and `||` to explore. You'll gain more understanding of how the `&&` and `||` operators work.

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `ExploreIf` and write a new `Main` method that calls `ExploreIf`. When you have finished, your code should look like this:

```
using System;

namespace BranchesAndLoops
{
    class Program
    {
        static void ExploreIf()
        {
            int a = 5;
            int b = 3;
            if (a + b > 10)
            {
                Console.WriteLine("The answer is greater than 10");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
            }

            int c = 4;
            if ((a + b + c > 10) && (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("And the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("Or the first number is not greater than the second");
            }

            if ((a + b + c > 10) || (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("Or the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("And the first number is not greater than the second");
            }
        }

        static void Main(string[] args)
        {
            ExploreIf();
        }
    }
}
```

Comment out the call to `ExploreIf()`. It will make the output less cluttered as you work in this section:

```
//ExploreIf();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler does not generate any executable code from comments.

Use loops to repeat operations

In this section you use **loops** to repeat statements. Try this code in your `Main` method:

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

The `while` statement checks a condition and executes the statement or statement block following the `while`. It repeatedly checks the condition and executing those statements until the condition is false.

There's one other new operator in this example. The `++` after the `counter` variable is the **increment** operator. It adds 1 to the value of `counter` and stores that value in the `counter` variable.

> **IMPORTANT**
> Make sure that the `while` loop condition changes to false as you execute the code. Otherwise, you create an **infinite loop** where your program never ends. That is not demonstrated in this sample, because you have to force your program to quit using **CTRL-C** or other means.

The `while` loop tests the condition before executing the code following the `while`. The `do` ... `while` loop executes the code first, and then checks the condition. The do while loop is shown in the following code:

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

This `do` loop and the earlier `while` loop produce the same output.

Work with the for loop

The **for** loop is commonly used in C#. Try this code in your Main() method:

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

This does the same work as the `while` loop and the `do` loop you've already used. The `for` statement has three parts that control how it works.

The first part is the **for initializer**: `int index = 0;` declares that `index` is the loop variable, and sets its initial value to `0`.

The middle part is the **for condition**: `index < 10` declares that this `for` loop continues to execute as long as the value of counter is less than 10.

The final part is the **for iterator**: `index++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `index` should be incremented by 1 each time the block executes.

Experiment with these yourself. Try each of the following:

- Change the initializer to start at a different value.

- Change the condition to stop at a different value.

When you're done, let's move on to write some code yourself to use what you've learned.

Combine branches and loops

Now that you've seen the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. You should get 63 for an answer. You can see one possible answer by viewing the completed code on GitHub.

You've completed the "branches and loops" tutorial.

You can continue with the Arrays and collections tutorial in your own development environment.

You can learn more about these concepts in these topics:

- If and else statement
- While statement
- Do statement
- For statement

Learn to manage data collections using the generic list type

8/19/2019 • 5 minutes to read • Edit Online

This introductory tutorial provides an introduction to the C# language and the basics of the List<T> class.

This tutorial expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in Become familiar with the development tools, with links to more details.

A basic list example

Create a directory named **list-tutorial**. Make that the current directory and run `dotnet new console`.

Open **Program.cs** in your favorite editor, and replace the existing code with the following:

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Replace `<name>` with your name. Save **Program.cs**. Type `dotnet run` in your console window to try it.

You've just created a list of strings, added three names to that list, and printed out the names in all CAPS. You're using concepts that you've learned in earlier tutorials to loop through the list.

The code to display names makes use of the string interpolation feature. When you precede a `string` with the `$` character, you can embed C# code in the string declaration. The actual string replaces that C# code with the value it generates. In this example, it replaces the `{name.ToUpper()}` with each name, converted to capital letters, because you called the ToUpper method.

Let's keep exploring.

Modify list contents

The collection you created uses the List<T> type. This type stores sequences of elements. You specify the type of the elements between the angle brackets.

One important aspect of this List<T> type is that it can grow or shrink, enabling you to add or remove elements. Add this code before the closing `}` in the `Main` method:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

You've added two more names to the end of the list. You've also removed one as well. Save the file, and type `dotnet run` to try it.

The List<T> enables you to reference individual items by **index** as well. You place the index between `[` and `]` tokens following the list name. C# uses 0 for the first index. Add this code directly below the code you just added and try it:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

You cannot access an index beyond the end of the list. Remember that indices start at 0, so the largest valid index is one less than the number of items in the list. You can check how long the list is using the Count property. Add the following code at the end of the Main method:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Save the file, and type `dotnet run` again to see the results.

Search and sort lists

Our samples use relatively small lists, but your applications may often create lists with many more elements, sometimes numbering in the thousands. To find elements in these larger collections, you need to search the list for different items. The IndexOf method searches for an item and returns the index of the item. Add this code to the bottom of your `Main` method:

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

The items in your list can be sorted as well. The Sort method sorts all the items in the list in their normal order (alphabetically in the case of strings). Add this code to the bottom of our `Main` method:

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Save the file and type `dotnet run` to try this latest version.

Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `WorkingWithStrings` and write a new `Main` method that calls `WorkingWithStrings`. When you have finished, your code should look like this:

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkingWithStrings();
        }

        public static void WorkingWithStrings()
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine();
            names.Add("Maria");
            names.Add("Bill");
            names.Remove("Ana");
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine($"My name is {names[0]}");
            Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

            Console.WriteLine($"The list has {names.Count} people in it");

            var index = names.IndexOf("Felipe");
            Console.WriteLine($"The name {names[index]} is at index {index}");

            var notFound = names.IndexOf("Not Found");
            Console.WriteLine($"When an item is not found, IndexOf returns {notFound}");

            names.Sort();
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Lists of other types

You've been using the `string` type in lists so far. Let's make a List<T> using a different type. Let's build a set of numbers.

Add the following to the bottom of your new `Main` method:

```
var fibonacciNumbers = new List<int> {1, 1};
```

That creates a list of integers, and sets the first two integers to the value 1. These are the first two values of a *Fibonacci Sequence*, a sequence of numbers. Each next Fibonacci number is found by taking the sum of the previous two numbers. Add this code:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Save the file and type `dotnet run` to see the results.

TIP

To concentrate on just this section, you can comment out the code that calls `WorkingWithStrings();`. Just put two `/` characters in front of the call like this: `// WorkingWithStrings();`.

Challenge

See if you can put together some of the concepts from this and earlier lessons. Expand on what you've built so far with Fibonacci Numbers. Try to write the code to generate the first 20 numbers in the sequence. (As a hint, the 20th Fibonacci number is 6765.)

Complete challenge

You can see an example solution by looking at the finished sample code on GitHub.

With each iteration of the loop, you're taking the last two integers in the list, summing them, and adding that value to the list. The loop repeats until you've added 20 items to the list.

Congratulations, you've completed the list tutorial. You can continue with the Introduction to classes tutorial in your own development environment.

You can learn more about working with the `List` type in the .NET Guide topic on collections. You'll also learn about many other collection types.

Explore object oriented programming with classes and objects

6/26/2019 • 8 minutes to read • Edit Online

This tutorial expects that you have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the Become familiar with the development tools with links to more details.

Create your application

Using a terminal window, create a directory named **classes**. You'll build your application there. Change to that directory and type `dotnet new console` in the console window. This command creates your application. Open **Program.cs**. It should look like this:

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

In this tutorial, you're going to create new types that represent a bank account. Typically developers define each class in a different text file. That makes it easier to manage as a program grows in size. Create a new file named **BankAccount.cs** in the **classes** directory.

This file will contain the definition of a ***bank account***. Object Oriented programming organizes code by creating types in the form of ***classes***. These classes contain the code that represents a specific entity. The `BankAccount` class represents a bank account. The code implements specific operations through methods and properties. In this tutorial, the bank account supports this behavior:

1. It has a 10-digit number that uniquely identifies the bank account.
2. It has a string that stores the name or names of the owners.
3. The balance can be retrieved.
4. It accepts deposits.
5. It accepts withdrawals.
6. The initial balance must be positive.
7. Withdrawals cannot result in a negative balance.

Define the bank account type

You can start by creating the basics of a class that defines that behavior. It would look like this:

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Before going on, let's take a look at what you've built. The `namespace` declaration provides a way to logically organize your code. This tutorial is relatively small, so you'll put all the code in one namespace.

`public class BankAccount` defines the class, or type, you are creating. Everything inside the `{` and `}` that follows the class declaration defines the behavior of the class. There are five ***members*** of the `BankAccount` class. The first three are ***properties***. Properties are data elements and can have code that enforces validation or other rules. The last two are ***methods***. Methods are blocks of code that perform a single function. Reading the names of each of the members should provide enough information for you or another developer to understand what the class does.

Open a new account

The first feature to implement is to open a bank account. When a customer opens an account, they must supply an initial balance, and information about the owner or owners of that account.

Creating a new object of the `BankAccount` type means defining a ***constructor*** that assigns those values. A ***constructor*** is a member that has the same name as the class. It is used to initialize objects of that class type. Add the following constructor to the `BankAccount` type:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Constructors are called when you create an object using `new`. Replace the line `Console.WriteLine("Hello World!");` in ***program.cs*** with the following line (replace `<name>` with your name):

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial
balance.");
```

Type `dotnet run` to see what happens.

Did you notice that the account number is blank? It's time to fix that. The account number should be assigned when the object is constructed. But it shouldn't be the responsibility of the caller to create it. The `BankAccount` class code should know how to assign new account numbers. A simple way to do this is to start with a 10-digit number. Increment it when each new account is created. Finally, store the current account number when an object is

constructed.

Add the following member declaration to the `BankAccount` class:

```
private static int accountNumberSeed = 1234567890;
```

This is a data member. It's `private`, which means it can only be accessed by code inside the `BankAccount` class. It's a way of separating the public responsibilities (like having an account number) from the private implementation (how account numbers are generated.) It is also `static`, which means it is shared by all of the `BankAccount` objects. The value of a non-static variable is unique to each instance of the `BankAccount` object. Add the following two lines to the constructor to assign the account number:

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Type `dotnet run` to see the results.

Create deposits and withdrawals

Your bank account class needs to accept deposits and withdrawals to work correctly. Let's implement deposits and withdrawals by creating a journal of every transaction for the account. That has a few advantages over simply updating the balance on each transaction. The history can be used to audit all transactions and manage daily balances. By computing the balance from the history of all transactions when needed, any errors in a single transaction that are fixed will be correctly reflected in the balance on the next computation.

Let's start by creating a new type to represent a transaction. This is a simple type that doesn't have any responsibilities. It needs a few properties. Create a new file named ***Transaction.cs***. Add the following code to it:

```
using System;

namespace classes
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Notes { get; }

        public Transaction(decimal amount, DateTime date, string note)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

Now, let's add a List<T> of `Transaction` objects to the `BankAccount` class. Add the following declaration:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

The List<T> class requires you to import a different namespace. Add the following at the beginning of **BankAccount.cs**:

```
using System.Collections.Generic;
```

Now, let's change how the `Balance` is reported. It can be found by summing the values of all transactions. Modify the declaration of `Balance` in the `BankAccount` class to the following:

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

This example shows an important aspect of ***properties***. You're now computing the balance when another programmer asks for the value. Your computation enumerates all transactions, and provides the sum as the current balance.

Next, implement the `MakeDeposit` and `MakeWithdrawal` methods. These methods will enforce the final two rules: that the initial balance must be positive, and that any withdrawal must not create a negative balance.

This introduces the concept of ***exceptions***. The standard way of indicating that a method cannot complete its work successfully is to throw an exception. The type of exception and the message associated with it describe the error. Here, the `MakeDeposit` method throws an exception if the amount of the deposit is negative. The `MakeWithdrawal` method throws an exception if the withdrawal amount is negative, or if applying the withdrawal results in a negative balance:

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

The `throw` statement **throws** an exception. Execution of the current block ends, and control transfers to the first matching `catch` block found in the call stack. You'll add a `catch` block to test this code a little later on.

The constructor should get one change so that it adds an initial transaction, rather than updating the balance directly. Since you already wrote the `MakeDeposit` method, call it from your constructor. The finished constructor should look like this:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now is a property that returns the current date and time. Test this by adding a few deposits and withdrawals in your `Main` method:

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Next, test that you are catching error conditions by trying to create an account with a negative balance:

```
// Test that the initial balances must be positive.
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

You use the `try` and `catch` statements to mark a block of code that may throw exceptions and to catch those errors that you expect. You can use the same technique to test the code that throws an exception for a negative balance:

```
// Test for a negative balance:
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Save the file and type `dotnet run` to try it.

Challenge - log all transactions

To finish this tutorial, you can write the `GetAccountHistory` method that creates a `string` for the transaction history. Add this method to the `BankAccount` type:

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    report.AppendLine("Date\tAmount\tNote");
    foreach (var item in allTransactions)
    {
        report.AppendLine($"{item.Date.ToShortDateString()}\t{item.Amount}\t{item.Notes}");
    }

    return report.ToString();
}
```

This uses the StringBuilder class to format a string that contains one line for each transaction. You've seen the string formatting code earlier in these tutorials. One new character is `\t`. That inserts a tab to format the output.

Add this line to test it in **Program.cs**:

```
Console.WriteLine(account.GetAccountHistory());
```

Type `dotnet run` to see the results.

Next Steps

If you got stuck, you can see the source for this tutorial in our GitHub repo

Congratulations, you've finished all our introduction to C# tutorials. If you're eager to learn more, try more of our tutorials

Use string interpolation to construct formatted strings

6/26/2019 • 7 minutes to read • Edit Online

This tutorial teaches you how to use C# string interpolation to insert values into a single result string. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that show you how to insert values into a string and format those values in different ways.

This tutorial expects that you have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. You also can complete the interactive version of this tutorial in your browser.

Create an interpolated string

Create a directory named **interpolated**. Make it the current directory and run the following command from a console window:

```
dotnet new console
```

This command creates a new .NET Core console application in the current directory.

Open **Program.cs** in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code, where you replace `<name>` with your name:

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Try this code by typing `dotnet run` in your console window. When you run the program, it displays a single string that includes your name in the greeting. The string included in the WriteLine method call is an *interpolated string expression*. It's a kind of template that lets you construct a single string (called the *result string*) from a string that includes embedded code. Interpolated strings are particularly useful for inserting values into a string or concatenating (joining together) strings.

This simple example contains the two elements that every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character. (If you'd like to see what happens if you include one, insert a space after the `$` character, save the file, and run the program again by typing `dotnet run` in the console window. The C# compiler displays an error message, "error CS1056: Unexpected character '$'".)
- One or more *interpolation expressions*. An interpolation expression is indicated by an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value (including `null`) inside the braces.

Let's try a few more string interpolation examples with some other data types.

Include different data types

In the previous section, you used string interpolation to insert one string inside of another. The result of an interpolation expression can be of any data type, though. Let's include values of various data types in an interpolated string.

In the following example, we first define a class data type `Vegetable` that has a `Name` property and a `ToString` method, which overrides the behavior of the Object.ToString() method. The `public` access modifier makes that method available to any client code to get the string representation of a `Vegetable` instance. In the example the `Vegetable.ToString` method returns the value of the `Name` property that is initialized at the `Vegetable` constructor:

```
public Vegetable(string name) => Name = name;
```

Then we create an instance of the `Vegetable` class named `item` by using the `new` operator and providing a name for the constructor `Vegetable`:

```
var item = new Vegetable("eggplant");
```

Finally, we include the `item` variable into an interpolated string that also contains a DateTime value, a Decimal value, and a `Unit` enumeration value. Replace all of the C# code in your editor with the following code, and then use the `dotnet run` command to run it:

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
   }
}
```

Note that the interpolation expression `item` in the interpolated string resolves to the text "eggplant" in the result string. That's because, when the type of the expression result is not a string, the result is resolved to a string in the following way:

- If the interpolation expression evaluates to `null`, an empty string ("", or String.Empty) is used.
- If the interpolation expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called. You can test this by updating the implementation of the `Vegetable.ToString` method. You might not even need to implement the `ToString` method since every type has some implementation of this method. To test this, comment out the definition of the `Vegetable.ToString` method in the example (to do that, put a comment symbol, `//`, in front of it). In the output, the string "eggplant" is replaced by the fully qualified type name ("Vegetable" in this example), which is the default behavior of the Object.ToString() method. The default behavior of the `ToString` method for an enumeration value is to return the string representation of the value.

In the output from this example, the date is too precise (the price of eggplant doesn't change every second), and

the price value doesn't indicate a unit of currency. In the next section, you'll learn how to fix those issues by controlling the format of string representations of the expression results.

Control the formatting of interpolation expressions

In the previous section, two poorly formatted strings were inserted into the result string. One was a date and time value for which only the date was appropriate. The second was a price that didn't indicate its unit of currency. Both issues are easy to address. String interpolation lets you specify *format strings* that control the formatting of particular types. Modify the call to `Console.WriteLine` from the previous example to include the format strings for the date and price expressions as shown in the following line:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

You specify a format string by following the interpolation expression with a colon (":") and the format string. "d" is a standard date and time format string that represents the short date format. "C2" is a standard numeric format string that represents a number as a currency value with two digits after the decimal point.

A number of types in the .NET libraries support a predefined set of format strings. These include all the numeric types and the date and time types. For a complete list of types that support format strings, see Format Strings and .NET Class Library Types in the Formatting Types in .NET article.

Try modifying the format strings in your text editor and, each time you make a change, rerun the program to see how the changes affect the formatting of the date and time and the numeric value. Change the "d" in `{date:d}` to "t" (to display the short time format), "y" (to display the year and month), and "yyyy" (to display the year as a four-digit number). Change the "C2" in `{price:C2}` to "e" (for exponential notation) and "F3" (for a numeric value with three digits after the decimal point).

In addition to controlling formatting, you can also control the field width and alignment of the formatted strings that are included in the result string. In the next section, you'll learn how to do this.

Control the field width and alignment of interpolation expressions

Ordinarily, when the result of an interpolation expression is formatted to string, that string is included in a result string without leading or trailing spaces. Particularly when you work with a set of data, being able to control a field width and text alignment helps to produce a more readable output. To see this, replace all the code in your text editor with the following code, then type `dotnet run` to execute the program:

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

The names of authors are left-aligned, and the titles they wrote are right-aligned. You specify the alignment by adding a comma (",") after an interpolation expression and designating the *minimum* field width. If the specified value is a positive number, the field is right-aligned. If it is a negative number, the field is left-aligned.

Try removing the negative signs from the `{"Author",-25}` and `{title.Key,-25}` code and run the example again, as the following code does:

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

This time, the author information is right-aligned.

You can combine an alignment specifier and a format string for a single interpolation expression. To do that, specify the alignment first, followed by a colon and the format string. Replace all of the code inside the `Main` method with the following code, which displays three formatted strings with defined field widths. Then run the program by entering the `dotnet run` command.

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

The output looks something like the following:

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

You've completed the string interpolation tutorial.

For more information, see the String interpolation topic and the String interpolation in C# tutorial.

String interpolation in C#

5/28/2019 • 5 minutes to read • Edit Online

This tutorial shows you how to use string interpolation to format and include expression results in a result string. The examples assume that you are familiar with basic C# concepts and .NET type formatting. If you are new to string interpolation or .NET type formatting, check out the interactive string interpolation tutorial first. For more information about formatting types in .NET, see the Formatting Types in .NET topic.

> **NOTE**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

Introduction

The string interpolation feature is built on top of the composite formatting feature and provides a more readable and convenient syntax to include formatted expression results in a result string.

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can embed any valid C# expression that returns a value in an interpolated string. In the following example, as soon as an expression is evaluated, its result is converted into a string and included in a result string:

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is
{CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

As the example shows, you include an expression in an interpolated string by enclosing it with braces:

```
{<interpolationExpression>}
```

Interpolated strings support all the capabilities of the string composite formatting feature. That makes them a more readable alternative to the use of the String.Format method.

How to specify a format string for an interpolation expression

You specify a format string that is supported by the type of the expression result by following the interpolation expression with a colon (":") and the format string:

```
{<interpolationExpression>:<formatString>}
```

The following example shows how to specify standard and custom format strings for expressions that produce date and time or numeric results:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote {Math.E:F5}
in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

For more information, see the Format String Component section of the Composite Formatting topic. That section provides links to the topics that describe standard and custom format strings supported by .NET base types.

How to control the field width and alignment of the formatted interpolation expression

You specify the minimum field width and the alignment of the formatted expression result by following the interpolation expression with a comma (",") and the constant expression:

```
{<interpolationExpression>,<alignment>}
```

If the *alignment* value is positive, the formatted expression result is right-aligned; if negative, it's left-aligned.

If you need to specify both alignment and a format string, start with the alignment component:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

The following example shows how to specify alignment and uses pipe characters ("|") to delimit text fields:

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|   3.500|
// |Geometric|   3.464|
// |Harmonic |   3.429|
```

As the example output shows, if the length of the formatted expression result exceeds specified field width, the *alignment* value is ignored.

For more information, see the Alignment Component section of the Composite Formatting topic.

How to use escape sequences in an interpolated string

Interpolated strings support all escape sequences that can be used in ordinary string literals. For more information, see String escape sequences.

To interpret escape sequences literally, use a verbatim string literal. A verbatim interpolated string starts with the `$` character followed by the `@` character.

To include a brace, "{" or "}", in a result string, use two braces, "{{" or "}}". For more information, see the Escaping Braces section of the Composite Formatting topic.

The following example shows how to include braces in a result string and construct a verbatim interpolated string:

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}}
sets.");

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

How to use a ternary conditional operator `?:` in an interpolation expression

As the colon (":") has special meaning in an item with an interpolation expression, in order to use a conditional operator in an expression, enclose it in parentheses, as the following example shows:

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

How to create a culture-specific result string with string interpolation

By default, an interpolated string uses the current culture defined by the CultureInfo.CurrentCulture property for all formatting operations. Use implicit conversion of an interpolated string to a System.FormattableString instance and call its ToString(IFormatProvider) method to create a culture-specific result string. The following example shows how to do that:

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US       5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL        17-05-18 15:44:55      31.415.926,536
//            05/17/2018 15:44:55      31,415,926.536
```

As the example shows, you can use one FormattableString instance to generate multiple result strings for various cultures.

How to create a result string using the invariant culture

Along with the FormattableString.ToString(IFormatProvider) method, you can use the static FormattableString.Invariant method to resolve an interpolated string to a result string for the InvariantCulture. The following example shows how to do that:

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusion

This tutorial describes common scenarios of string interpolation usage. For more information about string interpolation, see the String interpolation topic. For more information about formatting types in .NET, see the Formatting Types in .NET and Composite formatting topics.

See also

- String.Format
- System.FormattableString
- System.IFormattable
- Strings

Tutorial: Update interfaces with default interface members in C# 8.0

8/13/2019 • 6 minutes to read • Edit Online

Beginning with C# 8.0 on .NET Core 3.0, you can define an implementation when you declare a member of an interface. The most common scenario is to safely add members to an interface already released and used by innumerable clients.

In this tutorial, you'll learn how to:

- Extend interfaces safely by adding methods with implementations.
- Create parameterized implementations to provide greater flexibility.
- Enable implementers to provide a more specific implementation in the form of an override.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 preview compiler. The C# 8.0 preview compiler is available starting with Visual Studio 2019, or the latest .NET Core 3.0 preview SDK. Default interface members are available beginning with .NET Core 3.0 preview 4.

Scenario overview

This tutorial starts with version 1 of a customer relationship library. You can get the starter application on our samples repo on GitHub. The company that built this library intended customers with existing applications to adopt their library. They provided minimal interface definitions for users of their library to implement. Here's the interface definition for a customer:

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

They defined a second interface that represents an order:

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

From those interfaces, the team could build a library for their users to create a better experience for their customers. Their goal was to create a deeper relationship with existing customers and improve their relationships with new customers.

Now, it's time to upgrade the library for the next release. One of the requested features enables a loyalty discount for customers that have lots of orders. This new loyalty discount gets applied whenever a customer makes an order.

The specific discount is a property of each individual customer. Each implementation of ICustomer can set different rules for the loyalty discount.

The most natural way to add this functionality is to enhance the `ICustomer` interface with a method to apply any loyalty discount. This design suggestion caused concern among experienced developers: "Interfaces are immutable once they've been released! This is a breaking change!" C# 8.0 adds *default interface implementations* for upgrading interfaces. The library authors can add new members to the interface and provide a default implementation for those members.

Default interface implementations enable developers to upgrade an interface while still enabling any implementors to override that implementation. Users of the library can accept the default implementation as a non-breaking change. If their business rules are different, they can override.

Upgrade with default interface members

The team agreed on the most likely default implementation: a loyalty discount for customers.

The upgrade should provide the functionality to set two properties: the number of orders needed to be eligible for the discount, and the percentage of the discount. This makes it a perfect scenario for default interface members. You can add a method to the ICustomer interface, and provide the most likely implementation. All existing, and any new implementations can use the default implementation, or provide their own.

First, add the new method to the implementation:

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

The library author wrote a first test to check the implementation:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};


SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Notice the following portion of the test:

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

That cast from `SampleCustomer` to `ICustomer` is necessary. The `SampleCustomer` class doesn't need to provide an implementation for `ComputeLoyaltyDiscount`; that's provided by the `ICustomer` interface. However, the `SampleCustomer` class doesn't inherit members from its interfaces. That rule hasn't changed. In order to call any method declared and implemented in the interface, the variable must be the type of the interface, `ICustomer` in this example.

Provide parameterization

That's a good start. But, the default implementation is too restrictive. Many consumers of this system may choose different thresholds for number of purchases, a different length of membership, or a different percentage discount. You can provide a better upgrade experience for more customers by providing a way to set those parameters. Let's add a static method that sets those three parameters controlling the default implementation:

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

There's many new language capabilities shown in that small code fragment. Interfaces can now include static members, including fields and methods. Different access modifiers are also enabled. The additional fields are private, the new method is public. Any of the modifiers are allowed on interface members.

Applications that use the general formula for computing the loyalty discount, but different parameters, don't need to provide a custom implementation; they can set the arguments through a static method. For example, the following code sets a "customer appreciation" that rewards any customer with more than one month's membership:

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Extend the default implementation

The code you've added so far has provided a convenient implementation for those scenarios where users want

something like the default implementation, or to provide an unrelated set of rules. For a final feature, let's refactor the code a bit to enable scenarios where users may want to build on the default implementation.

Consider a startup that wants to attract new customers. They offer a 50% discount off a new customer's first order. Otherwise, existing customers get the standard discount. The library author needs to move the default implementation into a `protected static` method so that any class implementing this interface can reuse the code in their implementation. The default implementation of the interface member calls this shared method as well:

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

In an implementation of a class that implements this interface, the override can call the static helper method, and extend that logic to provide the "new customer" discount:

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

You can see the entire finished code in our samples repo on GitHub. You can get the starter application on our samples repo on GitHub.

These new features mean that interfaces can be updated safely when there's a reasonable default implementation for those new members. Carefully design interfaces to express single functional ideas that can be implemented by multiple classes. That makes it easier to upgrade those interface definitions when new requirements are discovered for that same functional idea.

Indices and ranges

5/13/2019 • 4 minutes to read • Edit Online

Ranges and indices provide a succinct syntax for accessing single elements or ranges in an Array, Span<T>, or ReadOnlySpan<T>. These features enable more concise, clear syntax to access single elements or ranges of elements in a sequence.

In this tutorial, you'll learn how to:

- Use the syntax for ranges in a sequence.
- Understand the design decisions for the start and end of each sequence.
- Learn scenarios for the Index and Range types.

Language support for indices and ranges

This language support relies on two new types and two new operators.

- System.Index represents an index into a sequence.
- The `^` operator, which specifies that an index is relative to the end of a sequence.
- System.Range represents a sub range of a sequence.
- The Range operator (`..`), which specifies the start and end of a range as its operands.

Let's start with the rules for indices. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. Note that `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index. Add the following code below the initialization:

```
Console.WriteLine($"The last word is {words[^1]}");
```

A range specifies the *start* and *end* of a range. Ranges are exclusive, meaning the *end* is not included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range. Add the following code to the same method. Copy and paste it at the bottom of the interactive window.

```
var quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following code creates a subrange with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included. Add the following code as well:

```
var lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following examples create ranges that are open ended for the start, end, or both:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the, "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

You can also declare ranges or indices as variables. The variable can then be used inside the `[` and `]` characters:

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
var text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following sample shows many of the reasons for those choices. Modify `x`, `y`, and `z` to try different combinations. When you experiment, use values where `x` is less than `y`, and `y` is less than `z` for valid combinations. Add the following code in a new method. Try different combinations:

```
var numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through
{y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");
```

Scenarios for Indices and Ranges

You'll often use ranges and indices when you want to perform some analysis on subrange of an entire sequence. The new syntax is clearer in reading exactly what subrange is involved. The local function `MovingAverage` takes a Range as its argument. The method then enumerates just that range when calculating the min, max, and average. Try the following code in your project:

```
int[] sequence = Sequence(1000);


for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..start+10;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();
```

You can download the completed code from the dotnet/samples repository.

Tutorial: Express your design intent more clearly with nullable and non-nullable reference types

5/24/2019 • 10 minutes to read • Edit Online

C# 8 introduces **nullable reference types**, which complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*.

In this tutorial, you'll learn how to:

- Incorporate nullable and non-nullable reference types into your designs
- Enable nullable reference type checks throughout your code.
- Write code where the compiler enforces those design decisions.
- Use the nullable reference feature in your own designs

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 beta compiler. The C# 8 beta compiler is available with Visual Studio 2019, or the latest .NET Core 3.0 preview.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Incorporate nullable reference types into your designs

In this tutorial, you'll build a library that models running a survey. The code uses both nullable reference types and non-nullable reference types to represent the real-world concepts. The survey questions can never be null. A respondent might prefer not to answer a question. The responses might be null in this case.

The code you'll write for this sample expresses that intent, and the compiler enforces that intent.

Create the application and enable nullable reference types

Create a new console application either in Visual Studio or from the command line using `dotnet new console`. Name the application `NullableIntroduction`. Once you've created the application, you'll need to enable C# 8 beta features. Open the `csproj` file and add a `LangVersion` element to the `PropertyGroup` element. You must opt into the **nullable reference types** feature, even in C# 8 projects. That's because once the feature is turned on, existing reference variable declarations become **non-nullable reference types**. While that decision will help find issues where existing code may not have proper null-checks, it may not accurately reflect your original design intent. You turn on the feature by setting the `Nullable` element to `enable`:

```
<LangVersion>8.0</LangVersion>
<Nullable>enable</Nullable>
```

> **IMPORTANT**
>
> The `Nullable` element was previously named `NullableContextOptions`. The rename ships with Visual Studio 2019, 16.2-p1. The .NET Core SDK 3.0.100-preview5-011568 does not have this change. If you are using the .NET Core CLI, you'll need to use `NullableContextOptions` until the next preview is available.

> **NOTE**
>
> When C# 8 is released (not in preview mode), the `Nullable` element will be added by new project templates. Until then, you'll need to add it manually.

Design the types for the application

This survey application requires creating a number of classes:

- A class that models the list of questions.
- A class that models a list of people contacted for the survey.
- A class that models the answers from a person that took the survey.

These types will make use of both nullable and non-nullable reference types to express which members are required and which members are optional. Nullable reference types communicate that design intent clearly:

- The questions that are part of the survey can never be null: It makes no sense to ask an empty question.
- The respondents can never be null. You'll want to track people you contacted, even respondents that declined to participate.
- Any response to a question may be null. Respondents can decline to answer some or all questions.

If you've programmed in C#, you may be so accustomed to reference types that allow null values that you may have missed other opportunities to declare non-nullable instances:

- The collection of questions should be non-nullable.
- The collection of respondents should be non-nullable.

As you write the code, you'll see that a non-nullable reference type as the default for references avoids common mistakes that could lead to null reference exceptions. One lesson from this tutorial is that you made decisions about which variables could or could not be null. The language didn't provide syntax to express those decisions. Now it does.

The app you'll build will do the following steps:

1. Create a survey and add questions to it.
2. Create a pseudo-random set of respondents for the survey.
3. Contact respondents until the completed survey size reaches the goal number.
4. Write out important statistics on the survey responses.

Build the survey with nullable and non-nullable types

The first code you'll write creates the survey. You'll write classes to model a survey question and a survey run. Your survey has three types of questions, distinguished by the format of the answer: Yes/No answers, number answers, and text answers. Create a `public` `SurveyQuestion` class:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

The compiler interprets every reference type variable declaration as a **non-nullable** reference type for code in a nullable enabled context. You can see your first warning by adding properties for the question text and the type of question, as shown in the following code:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Because you haven't initialized `QuestionText`, the compiler issues a warning that a non-nullable property hasn't been initialized. Your design requires the question text to be non-null, so you add a constructor to initialize it and the `QuestionType` value as well. The finished class definition looks like the following code:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}
```

Adding the constructor removes the warning. The constructor argument is also a non-nullable reference type, so the compiler doesn't issue any warnings.

Next, create a `public` class named `SurveyRun`. This class contains a list of `SurveyQuestion` objects and methods to add questions to the survey, as shown in the following code:

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}
```

As before, you must initialize the list object to a non-null value or the compiler issues a warning. There are no null checks in the second overload of `AddQuestion` because they aren't needed: You've declared that variable to be non-nullable. Its value can't be `null`.

Switch to `Program.cs` in your editor and replace the contents of `Main` with the following lines of code:

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that
happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Because the entire project is in a nullable enabled context, you'll get warnings when you pass `null` to any method expecting a non-nullable reference type. Try it by adding the following line to `Main`:

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Create respondents and get answers to the survey

Next, write the code that generates answers to the survey. This process involves several small tasks:

1. Build a method that generates respondent objects. These represent people asked to fill out the survey.
2. Build logic to simulate asking the questions to a respondent and collecting answers or noting that a respondent didn't answer.
3. Repeat until enough respondents have answered the survey.

You'll need a class to represent a survey response, so add that now. Enable nullable support. Add an `Id` property and a constructor that initializes it, as shown in the following code:

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Next, add a `static` method to create new participants by generating a random ID:

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

The main responsibility of this class is to generate the responses for a participant to the questions in the survey. This responsibility has a few steps:

1. Ask for participation in the survey. If the person doesn't consent, return a missing (or null) response.
2. Ask each question and record the answer. Each answer may also be missing (or null).

Add the following code to your `SurveyResponse` class:

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

The storage for the survey answers is a `Dictionary<int, string>?`, indicating that it may be null. You're using the new language feature to declare your design intent, both to the compiler and to anyone reading your code later. If you ever dereference `surveyResponses` without checking for the null value first, you'll get a compiler warning. You

don't get a warning in the `AnswerSurvey` method because the compiler can determine the `surveyResponses` variable was set to a non-null value above.

Using `null` for missing answers highlights a key point for working with nullable reference types: your goal isn't to remove all `null` values from your program. Rather, your goal is to ensure that the code you write expresses the intent of your design. Missing values are a necessary concept to express in your code. The `null` value is a clear way to express those missing values. Trying to remove all `null` values only leads to defining some other way to express those missing values without `null`.

Next, you need to write the `PerformSurvey` method in the `SurveyRun` class. Add the following code in the `SurveyRun` class:

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int repondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (repondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            repondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Here again, your choice of a nullable `List<SurveyResponse>?` indicates the response may be null. That indicates the survey hasn't been given to any respondents yet. Notice that respondents are added until enough have consented.

The last step to run the survey is to add a call to perform the survey at the end of the `Main` method:

```
surveyRun.PerformSurvey(50);
```

Examine survey responses

The last step is to display survey results. You'll add code to many of the classes you've written. This code demonstrates the value of distinguishing nullable and non-nullable reference types. Start by adding the following two expression-bodied members to the `SurveyResponse` class:

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses.GetValueOrDefault(index, "No answer")!;
```

Because `surveyResponses` is a non-nullable reference type, no checks are necessary before de-referencing it. The `Answer` method returns a non-nullable string, so choose the overload of `GetValueOrDefault` that takes a second argument for the default value.

Next, add these three expression-bodied members to the `SurveyRun` class:

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

The `AllParticipants` member must take into account that the `respondents` variable might be null, but the return value can't be null. If you change that expression by removing the `??` and the empty sequence that follows, the compiler warns you the method might return `null` and its return signature returns a non-nullable type.

Finally, add the following loop at the bottom of the `Main` method:

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i)} : {answer}");
        }
    }
    else
        Console.WriteLine("\tNo responses");
}
```

You don't need any `null` checks in this code because you've designed the underlying interfaces so that they all return non-nullable reference types.

Get the code

You can get the code for the finished tutorial from our samples repository in the csharp/NullableIntroduction folder.

Experiment by changing the type declarations between nullable and non-nullable reference types. See how that generates different warnings to ensure you don't accidentally dereference a `null`.

Next steps

Learn more by migrating an existing application to use nullable reference types:

Upgrade an application to use nullable reference types

Tutorial: Migrate existing code with nullable reference types

5/24/2019 • 13 minutes to read • Edit Online

C# 8 introduces **nullable reference types**, which complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value*, others *may be missing a value*. Any existing variables of a reference type would be interpreted as a non-nullable reference type.

In this tutorial, you'll learn how to:

- Enable null reference checks as you work with code.
- Diagnose and correct different warnings related to null values.
- Manage the interface between nullable enabled and nullable disabled contexts.
- Control nullable annotation contexts.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 beta compiler. The C# 8 beta compiler is available with Visual Studio 2019, or the latest .NET Core 3.0 preview.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Explore the sample application

The sample application that you'll migrate is an RSS feed reader web app. It reads from a single RSS feed and displays summaries for the most recent articles. You can click on any of the articles to visit the site. The application is relatively new but was written before nullable reference types were available. The design decisions for the application represented sound principles, but don't take advantage of this important language feature.

The sample application includes a unit test library that validates the major functionality of the app. That project will make it easier to upgrade safely, if you change any of the implementation based on the warnings generated. You can download the starter code from the dotnet/samples GitHub repository.

Your goal migrating a project should be to leverage the new language features so that you clearly express your intent on the nullability of variables, and do so in such a way that the compiler doesn't generate warnings when you have the nullable annotation context and nullable warning context set to `enabled`.

Upgrade the projects to C# 8

A good first step is to determine the scope of the migration task. Start by upgrading the project to C# 8.0 (or newer). Add the `LangVersion` element to both csproj files for the web project and the unit test project:

```
<LangVersion>8.0</LangVersion>
```

Upgrading the language version selects C# 8.0, but does not enable the nullable annotation context or the nullable warning context. Rebuild the project to ensure that it builds without warnings.

A good next step is to turn on the nullable annotation context and see how many warnings are generated. Add the

following element to both csproj files in the solution, directly under the `LangVersion` element:

```
<Nullable>enable</Nullable>
```

> **IMPORTANT**
> The `Nullable` element was previously named `NullableContextOptions`. The rename ships with Visual Studio 2019, 16.2-p1. The .NET Core SDK 3.0.100-preview5-011568 does not have this change. If you are using the .NET Core CLI, you'll need to use `NullableContextOptions` until the next preview is available.

Do a test build, and notice the warning list. In this small application, the compiler generates five warnings, so it's likely you'd leave the nullable annotation context enabled and start fixing warnings for the entire project.

That strategy works only for smaller projects. For any larger projects, the number of warnings generated by enabling the nullable annotation context for the entire codebase makes it harder to fix the warnings systematically. For larger enterprise projects, you'll often want to migrate one project at a time. In each project, migrate one class or file at a time.

Warnings help discover original design intent

There are two classes that generate multiple warnings. Start with the `NewsStoryViewModel` class. Remove the `Nullable` element from both csproj files so that you can limit the scope of warnings to the sections of code you're working with. Open the *NewsStoryViewModel.cs* file and add the following directives to enable the nullable annotation context for the `NewsStoryViewModel` and restore it following that class definition:

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```

These two directives help you focus your migration efforts. The nullable warnings are generated for the area of code you're actively working on. You'll leave them on until you're ready to turn on the warnings for the entire project. You should use the `restore` rather than `disable` value so that you don't accidentally disable the context later when you've turned on nullable annotations for the entire project. Once you've turned on the nullable annotation context for the entire project, you can remove all the `#nullable` pragmas from that project.

The `NewsStoryViewModel` class is a data transfer object (DTO) and two of the properties are read/write strings:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

These two properties cause `CS8618`, "Non-nullable property is uninitialized". That's clear enough: both `string` properties have the default value of `null` when a `NewsStoryViewModel` is constructed. What's important to discover is how `NewsStoryViewModel` objects are constructed. Looking at this class, you can't tell if the `null` value is part of the design, or if these objects are set to non-null values whenever one is created. The news stories are created in the `GetNews` method of the `NewsService` class:

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

There's quite a bit going on in the preceding block of code. This application uses the AutoMapper NuGet package to construct a news item from an `ISyndicationItem`. You've discovered that the news story items are constructed and the properties are set in that one statement. That means the design for the `NewsStoryViewModel` indicates that these properties should never have the `null` value. These properties should be **nonnullable reference types**. That best expresses the original design intent. In fact, any `NewsStoryViewModel` *is* correctly instantiated with non-null values. That makes the following initialization code a valid fix:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

The assignment of `Title` and `Uri` to `default` which is `null` for the `string` type doesn't change the runtime behavior of the program. The `NewsStoryViewModel` is still constructed with null values, but now the compiler reports no warnings. The **null-forgiving operator**, the `!` character following the `default` expression tells the compiler that the preceding expression is not null. This technique may be expedient when other changes force much larger changes to a code base, but in this application there is a relatively quick and better solution: Make the `NewsStoryViewModel` an immutable type where all the properties are set in the constructor. Make the following changes to the `NewsStoryViewModel`:

```
#nullable enable
    public class NewsStoryViewModel
    {
        public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
            (Published, Title, Uri) = (published, title, uri);

        public DateTimeOffset Published { get; }
        public string Title { get; }
        public string Uri { get; }
    }
#nullable restore
```

Once that's done, you need to update the code that configures the AutoMapper so that it uses the constructor rather than setting properties. Open `NewsService.cs` and look for the following code at the bottom of the file:

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

That code maps properties of the `ISyndicationItem` object to the `NewsStoryViewModel` properties. You want the AutoMapper to provide the mapping using a constructor instead. Replace the above code with the following automapper configuration:

```
#nullable enable
    public class NewsStoryProfile : Profile
    {
        public NewsStoryProfile()
        {
            // Create the AutoMapper mapping profile between the 2 objects.
            // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
            CreateMap<ISyndicationItem, NewsStoryViewModel>()
                .ForCtorParam("published", opt => opt.MapFrom(src => src.Published))
                .ForCtorParam("title", opt => opt.MapFrom(src => src.Title))
                .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
        }
```

Notice that because this class is small, and you've examined carefully, you should turn on the `#nullable enable` directive above this class declaration. The change to the constructor could have broken something, so it's worthwhile to run all the tests and test the application before moving on.

The first set of changes showed you how to discover when the original design indicated that variables shouldn't be set to `null`. The technique is referred to as **correct by construction**. You declare that an object and its properties cannot be `null` when it's constructed. The compiler's flow analysis provides assurance that those properties aren't set to `null` after construction. Note that this constructor is called by external code, and that code is **nullable oblivious**. The new syntax doesn't provide runtime checking. External code might circumvent the compiler's flow analysis.

Other times, the structure of a class provides different clues to the intent. Open the *Error.cshtml.cs* file in the *Pages* folder. The `ErrorViewModel` contains the following code:

```
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
```

Add the `#nullable enable` directive before the class declaration, and a `#nullable restore` directive after it. You'll get one warning that `RequestId` is not initialized. By looking at the class, you should decide that the `RequestId` property should be null in some cases. The existence of the `ShowRequestId` property indicates that missing values are possible. Because `null` is valid, add the `?` on the `string` type to indicate the `RequestId` property is a *nullable reference type*. The final class looks like the following example:

```
#nullable enable
    public class ErrorModel : PageModel
    {
        public string? RequestId { get; set; }

        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

        public void OnGet()
        {
            RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        }
    }
#nullable restore
```

Check for the uses of the property, and you see that in the associated page, the property is checked for null before rendering it in markup. That's a safe use of a nullable reference type, so you're done with this class.

Fixing nulls causes change

Frequently, the fix for one set of warnings creates new warnings in related code. Let's see the warnings in action by fixing the `index.cshtml.cs` class. Open the `index.cshtml.cs` file and examine the code. This file contains the code behind for the index page:

```
public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a
syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}
```

Add the `#nullable enable` directive and you'll see two warnings. Neither the `ErrorText` property nor the `NewsItems` property is initialized. An examination of this class would lead you to believe that both properties should be nullable reference types: Both have private setters. Exactly one is assigned in the `OnGet` method. Before making changes, look at the consumers of both properties. In the page itself, the `ErrorText` is checked against null before generating markup for any errors. The `NewsItems` collection is checked against `null`, and checked to ensure the collection has items. A quick fix would be to make both properties nullable reference types. A better fix would be to make the collection a nonnullable reference type, and add items to the existing collection when retrieving news. The first fix is to add the `?` to the `string` type for the `ErrorText`:

```
public string? ErrorText { get; private set; }
```

That change won't ripple through other code, because any access to the `ErrorText` property was already guarded by null checks. Next, initialize the `NewsItems` list and remove the property setter, making it a readonly property:

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

That fixed the warning but introduced an error. The `NewsItems` list is now **correct by construction**, but the code that sets the list in `OnGet` must change to match the new API. Instead of an assignment, call `AddRange` to add the news items to the existing list:

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

Using `AddRange` instead of an assignment means that the `GetNews` method can return an `IEnumerable` instead of a `List`. That saves one allocation. Change the signature of the method, and remove the `ToList` call, as shown in the following code sample:

```
public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
           new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}
```

Changing the signature breaks one of tests as well. Open the `NewsServiceTests.cs` file in the `Services` folder of the `SimpleFeedReader.Tests` project. Navigate to the `Returns_News_Stories_Given_Valid_Uri` test and change the type of the `result` variable to `IEnumerable<NewsItem>`. Changing the type means the `Count` property is no longer available, so replace the `Count` property in the `Assert` with a call to `Any()`:

```
// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());
```

You'll need to add a `using System.Linq` statement to the beginning of the file as well.

This set of changes highlights special consideration when updating code that includes generic instantiations. Both the list and the elements in the list of non-nullable types. Either or both could be nullable types. All the following declarations are allowed:

- `List<NewsStoryViewModel>`: nonnullable list of nonullable view models.
- `List<NewsStoryViewModel?>`: nonnullable list of nullable view models.
- `List<NewsStoryViewModel>?`: nullable list of nonnullable view models.
- `List<NewsStoryViewModel?>?`: nullable list of nullable view models.

Interfaces with external code

You've made changes to the `NewsService` class, so turn on the `#nullable enable` annotation for that class. This won't generate any new warnings. However, careful examination of the class helps to illustrate some of the limitations of the compiler's flow analysis. Examine the constructor:

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

The `IMapper` parameter is typed as a nonnullable reference. It's called by ASP.NET Core infrastructure code, so the compiler doesn't really know that the `IMapper` will never be null. The default ASP.NET Core dependency injection (DI) container throws an exception if it can't resolve a necessary service, so the code is correct. The compiler can't validate all calls to your public APIs, even if your code is compiled with nullable annotation contexts enabled. Furthermore, your libraries may be consumed by projects that have not yet opted into using nullable reference types. Validate inputs to public APIs even though you've declared them as nonnullable types.

Get the code

You've fixed the warnings you identified in the initial test compile, so now you can turn on the nullable annotation context for both projects. Rebuild the projects; the compiler reports no warnings. You can get the code for the finished project in the dotnet/samples GitHub repository.

The new features that support nullable reference types help you find and fix potential errors in how you handle `null` values in your code. Enabling the nullable annotation context allows you to express your design intent: some variables should never be null, other variables may contain null values. These features make it easier for you to declare your design intent. Similarly, the nullable warning context instructs the compiler to issue warnings when you have violated that intent. Those warnings guide you to make updates that make your code more resilient and less likely to throw a `NullReferenceException` during execution. You can control the scope of these contexts so that you can focus on local areas of code to migrate while the remaining codebase is untouched. In practice, you can make this migration task a part of regular maintenance to your classes. This tutorial demonstrated the process to migrate an application to use nullable reference types. You can explore a larger real-world example of this process by examining the PR Jon Skeet made to incorporate nullable reference types into NodaTime.

Tutorial: Generate and consume async streams using C# 8.0 and .NET Core 3.0

4/9/2019 • 8 minutes to read • Edit Online

C# 8.0 introduces **async streams**, which model a streaming source of data when the elements in the data stream may be retrieved or generated asynchronously. Async streams rely on new interfaces introduced in .NET Standard 2.1 and implemented in .NET Core 3.0 to provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- Create a data source that generates a sequence of data elements asynchronously.
- Consume that data source asynchronously.
- Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 beta compiler. The C# 8 beta compiler is available starting with Visual Studio 2019, or the latest .NET Core 3.0 preview SDK. Async streams are first available in .NET Core 3.0 preview 1.

You'll need to create a GitHub access token so that you can access the GitHub GraphQL endpoint. Select the following permissions for your GitHub Access Token:

- repo:status
- public_repo

Save the access token in a safe place so you can use it to gain access to the GitHub API endpoint.

> **WARNING**
> Keep your personal access token secure. Any software with your personal access token could make GitHub API calls using your access rights.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Run the starter application

You can get the code for the starter application used in this tutorial from our dotnet/samples repository in the csharp/tutorials/AsyncStreams folder.

The starter application is a console application that uses the GitHub GraphQL interface to retrieve recent issues written in the dotnet/docs repository. Start by looking at the following code for the starter app `Main` method:

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-
token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

You can either set a `GitHubKey` environment variable to your personal access token, or you can replace the last argument in the call to `GenEnvVariable` with your personal access token. Don't put your access code in source code if you'll be saving the source with others, or putting it in a shared source repository.

After creating the GitHub client, the code in `Main` creates a progress reporting object and a cancellation token. Once those objects are created, `Main` calls `runPagedQueryAsync` to retrieve the most recent 250 created issues. After that task has finished, the results are displayed.

When you run the starter application, you can make some important observations about how this application runs. You'll see progress reported for each page returned from GitHub. You can observe a noticeable pause before GitHub returns each new page of issues. Finally, the issues are displayed only after all 10 pages have been retrieved from GitHub.

Examine the implementation

The implementation reveals why you observed the behavior discussed in the previous section. Examine the code for `runPagedQueryAsync`:

```
private static async Task<JArray> runPagedQueryAsync(GitHubClient client, string queryText, string repoName,
CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Let's concentrate on the paging algorithm and async structure of the preceding code. (You can consult the GitHub GraphQL documentation for details on the GitHub GraphQL API.) The `runPagedQueryAsync` method enumerates the issues from most recent to oldest. It requests 25 issues per page and examines the `pageInfo` structure of the response to continue with the previous page. That follows GraphQL's standard paging support for multi-page responses. The response includes a `pageInfo` object that includes a `hasPreviousPages` value and a `startCursor` value used to request the previous page. The issues are in the `nodes` array. The `runPagedQueryAsync` method appends these nodes to an array that contains all the results from all pages.

After retrieving and restoring a page of results, `runPagedQueryAsync` reports progress and checks for cancellation. If cancellation has been requested, `runPagedQueryAsync` throws an OperationCanceledException.

There are several elements in this code that can be improved. Most importantly, `runPagedQueryAsync` must allocate storage for all the issues returned. This sample stops at 250 issues because retrieving all open issues would require much more memory to store all the retrieved issues. In addition, the protocols for supporting progress and supporting cancellation make the algorithm harder to understand on its first reading. You must look for the progress class to find where progress is reported. You also have to trace the communications through the CancellationTokenSource and its associated CancellationToken to understand where cancellation is requested and where it's granted.

Async streams provide a better way

Async streams and the associated language support address all those concerns. The code that generates the sequence can now use `yield return` to return elements in a method that was declared with the `async` modifier.

You can consume an async stream using an `await foreach` loop just as you consume any sequence using a `foreach` loop.

These new language features depend on three new interfaces added to .NET Standard 2.1 and implemented in .NET Core 3.0:

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        T Current { get; }

        ValueTask<bool> MoveNextAsync();
    }
}

namespace System
{
    public interface IAsyncDisposable
    {
        ValueTask DisposeAsync();
    }
}
```

These three interfaces should be familiar to most C# developers. They behave in a manner similar to their synchronous counterparts:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

One type that may be unfamiliar is System.Threading.Tasks.ValueTask. The `ValueTask` struct provides a similar API to the System.Threading.Tasks.Task class. `ValueTask` is used in these interfaces for performance reasons.

Convert to async streams

Next, convert the `runPagedQueryAsync` method to generate an async stream. First, change the signature of `runPagedQueryAsync` to return an `IAsyncEnumerable<JToken>`, and remove the cancellation token and progress objects from the parameter list as shown in the following code:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

The starter code processes each page as the page is retrieved, as shown in the following code:

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Replace those three lines with the following code:

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

You can also remove the declaration of `finalResults` earlier in this method and the `return` statement that follows the loop you modified.

You've finished the changes to generate an async stream. The finished method should resemble the code below:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Next, you change the code that consumes the collection to consume the async stream. Find the following code in `Main` that processes the collection of issues:

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Replace that code with the following `await foreach` loop:

```
await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs"))
    Console.WriteLine(issue);
```

You can get the code for the finished tutorial from the dotnet/samples repository in the csharp/tutorials/AsyncStreams folder.

Run the finished application

Run the application again. Contrast its behavior with the behavior of the starter application. The first page of results is enumerated as soon as it's available. There's an observable pause as each new page is requested and retrieved, then the next page's results are quickly enumerated. The `try` / `catch` block isn't needed to handle cancellation: the caller can stop enumerating the collection. Progress is clearly reported because the async stream generates results as each page is downloaded.

You can see improvements in memory use by examining the code. You no longer need to allocate a collection to store all the results before they're enumerated. The caller can determine how to consume the results and if a storage collection is needed.

Run both the starter and finished applications and you can observe the differences between the implementations for yourself. You can delete the GitHub access token you created when you started this tutorial after you've finished. If an attacker gained access to that token, they could access GitHub APIs using your credentials.

Tutorial: Using pattern matching features to extend data types

6/26/2019 • 14 minutes to read • Edit Online

C# 7 introduced basic pattern matching features. Those features are extended in C# 8 with new expressions and patterns. You can write functionality that behaves as though you extended types that may be in other libraries. Another use for patterns is to create functionality your application requires that isn't a fundamental feature of the type being extended.

In this tutorial, you'll learn how to:

- Recognize situations where pattern matching should be used.
- Use pattern matching expressions to implement behavior based on types and property values.
- Combine pattern matching with other techniques to create complete algorithms.

Prerequisites

You'll need to set up your machine to run .NET Core, including the C# 8.0 preview compiler. The C# 8 preview compiler is available with the latest Visual Studio 2019, or the latest .NET Core 3.0 preview.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET Core CLI.

Scenarios for pattern matching

Modern development often includes integrating data from multiple sources and presenting information and insights from that data in a single cohesive application. You and your team won't have control or access for all the types that represent the incoming data.

The classic object-oriented design would call for creating types in your application that represent each data type from those multiple data sources. Then, your application would work with those new types, build inheritance hierarchies, create virtual methods, and implement abstractions. Those techniques work, and sometimes they are the best tools. Other times you can write less code. You can write more clear code using techniques that separate the data from the operations that manipulate that data.

In this tutorial, you'll create and explore an application that takes incoming data from several external sources for a single scenario. You'll see how **pattern matching** provides an efficient way to consume and process that data in ways that weren't part of the original system.

Consider a major metropolitan area that is using tolls and peak time pricing to manage traffic. You write an application that calculates tolls for a vehicle based on its type. Later enhancements incorporate pricing based on the number of occupants in the vehicle. Further enhancements add pricing based on the time and the day of the week.

From that brief description, you may have quickly sketched out an object hierarchy to model this system. However, your data is coming from multiple sources like other vehicle registration management systems. These systems provide different classes to model that data and you don't have a single object model you can use. In this tutorial, you'll use these simplified classes to model for the vehicle data from these external systems, as shown in the following code:

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

You can download the starter code from the dotnet/samples GitHub repository. You can see that the vehicle classes are from different systems, and are in different namespaces. No common base class, other than `System.Object` can be leveraged.

Pattern matching designs

The scenario used in this tutorial highlights the kinds of problems that pattern matching is well-suited to solve:

- The objects you need to work with aren't in an object hierarchy that matches your goals. You may be working with classes that are part of unrelated systems.
- The functionality you're adding isn't part of the core abstraction for these classes. The toll paid by a vehicle *changes* for different types of vehicles, but the toll isn't a core function of the vehicle.

When the *shape* of the data and the *operations* on that data are not described together, the pattern matching features in C# make it easier to work with.

Implement the basic toll calculations

The most basic toll calculation relies only on the vehicle type:

- A `Car` is $2.00.
- A `Taxi` is $3.50.
- A `Bus` is $5.00.
- A `DeliveryTruck` is $10.00

Create a new `TollCalculator` class, and implement pattern matching on the vehicle type to get the toll amount. The following code shows the initial implementation of the `TollCalculator`.

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    public class TollCalculator
    {
        public decimal CalculateToll(object vehicle) =>
            vehicle switch
        {
            Car c           => 2.00m,
            Taxi t          => 3.50m,
            Bus b           => 5.00m,
            DeliveryTruck t => 10.00m,
            { }             => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
            null            => throw new ArgumentNullException(nameof(vehicle))
        };
    }
}
```

The preceding code uses a **switch expression** (not the same as a `switch` statement) that tests the **type pattern**. A **switch expression** begins with the variable, `vehicle` in the preceding code, followed by the `switch` keyword. Next comes all the **switch arms** inside curly braces. The `switch` expression makes other refinements to the syntax that surrounds the `switch` statement. The `case` keyword is omitted, and the result of each arm is an expression. The last two arms show a new language feature. The `{ }` case matches any non-null object that didn't match an earlier arm. This arm catches any incorrect types passed to this method. The `{ }` case must follow the cases for each vehicle type. If the order were reversed, the `{ }` case would take precedence. Finally, the `null` pattern detects when a `null` is passed to this method. The `null` pattern can be last because the other type patterns match only a non-null object of the correct type.

You can test this code using the following code in `Program.cs`:

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            var tollCalc = new TollCalculator();

            var car = new Car();
            var taxi = new Taxi();
            var bus = new Bus();
            var truck = new DeliveryTruck();

            Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
            Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
            Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
            Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

            try
            {
                tollCalc.CalculateToll("this will fail");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Caught an argument exception when using the wrong type");
            }
            try
            {
                tollCalc.CalculateToll(null);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("Caught an argument exception when using null");
            }
        }
    }
}
```

That code is included in the starter project, but is commented out. Remove the comments, and you can test what you've written.

You're starting to see how patterns can help you create algorithms where the code and the data are separate. The `switch` expression tests the type and produces different values based on the results. That's only the beginning.

Add occupancy pricing

The toll authority wants to encourage vehicles to travel at maximum capacity. They've decided to charge more when vehicles have fewer passengers, and encourage full vehicles by offering lower pricing:

- Cars and taxis with no passengers pay an extra $0.50.
- Cars and taxis with two passengers get a $0.50 discount.
- Cars and taxis with three or more passengers get a $1.00 discount.
- Buses that are less than 50% full pay an extra $2.00.
- Buses that are more than 90% full get a $1.00 discount.

These rules can be implemented using the **property pattern** in the same switch expression. The property pattern examines properties of the object once the type has been determined. The single case for a `Car` expands to four

different cases:

```
vehicle switch
{
    Car { Passengers: 0}        => 2.00m + 0.50m,
    Car { Passengers: 1 }       => 2.0m,
    Car { Passengers: 2}        => 2.0m - 0.50m,
    Car c                       => 2.00m - 1.0m,

    // ...
};
```

The first three cases test the type as a `Car`, then check the value of the `Passengers` property. If both match, that expression is evaluated and returned.

You would also expand the cases for taxis in a similar manner:

```
vehicle switch
{
    // ...

    Taxi { Fares: 0}  => 3.50m + 1.00m,
    Taxi { Fares: 1 } => 3.50m,
    Taxi { Fares: 2}  => 3.50m - 0.50m,
    Taxi t            => 3.50m - 1.00m,

    // ...
};
```

In the preceding example, the `when` clause was omitted on the final case.

Next, implement the occupancy rules by expanding the cases for buses, as shown in the following example:

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    // ...
};
```

The toll authority isn't concerned with the number of passengers in the delivery trucks. Instead, they adjust the toll amount based on the weight class of the trucks as follows:

- Trucks over 5000 lbs are charged an extra $5.00.
- Light trucks under 3000 lbs are given a $2.00 discount.

That rule is implemented with the following code:

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};
```

The preceding code shows the `when` clause of a switch arm. You use the `when` clause to test conditions other than equality on a property. When you've finished, you'll have a method that looks much like the following:

```
vehicle switch
{
    Car { Passengers: 0}        => 2.00m + 0.50m,
    Car { Passengers: 1}        => 2.0m,
    Car { Passengers: 2}        => 2.0m - 0.50m,
    Car c                       => 2.00m - 1.0m,

    Taxi { Fares: 0}  => 3.50m + 1.00m,
    Taxi { Fares: 1 } => 3.50m,
    Taxi { Fares: 2}  => 3.50m - 0.50m,
    Taxi t            => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};
```

Many of these switch arms are examples of **recursive patterns**. For example, `Car { Passengers: 1}` shows a constant pattern inside a property pattern.

You can make this code less repetitive by using nested switches. The `Car` and `Taxi` both have four different arms in the preceding examples. In both cases, you can create a type pattern that feeds into a property pattern. This technique is shown in the following code:

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

In the preceding sample, using a recursive expression means you don't repeat the `Car` and `Taxi` arms containing child arms that test the property value. This technique isn't used for the `Bus` and `DeliveryTruck` arms because those arms are testing ranges for the property, not discrete values.

Add peak pricing

For the final feature, the toll authority wants to add time sensitive peak pricing. During the morning and evening rush hours, the tolls are doubled. That rule only affects traffic in one direction: inbound to the city in the morning, and outbound in the evening rush hour. During other times during the workday, tolls increase by 50%. Late night and early morning, tolls are reduced by 25%. During the weekend, it's the normal rate, regardless of the time.

You'll use pattern matching for this feature, but you'll integrate it with other techniques. You could build a single pattern match expression that would account for all the combinations of direction, day of the week, and time. The result would be a complicated expression. It would be hard to read and difficult to understand. That makes it hard to ensure correctness. Instead, combine those methods to build a tuple of values that concisely describes all those states. Then use pattern matching to calculate a multiplier for the toll. The tuple contains three discrete conditions:

- The day is either a weekday or a weekend.
- The band of time when the toll is collected.
- The direction is into the city or out of the city

The following table shows the combinations of input values and the peak pricing multiplier:

DAY	TIME	DIRECTION	PREMIUM
Weekday	morning rush	inbound	x 2.00
Weekday	morning rush	outbound	x 1.00

DAY	TIME	DIRECTION	PREMIUM
Weekday	daytime	inbound	x 1.50
Weekday	daytime	outbound	x 1.50
Weekday	evening rush	inbound	x 1.00
Weekday	evening rush	outbound	x 2.00
Weekday	overnight	inbound	x 0.75
Weekday	overnight	outbound	x 0.75
Weekend	morning rush	inbound	x 1.00
Weekend	morning rush	outbound	x 1.00
Weekend	daytime	inbound	x 1.00
Weekend	daytime	outbound	x 1.00
Weekend	evening rush	inbound	x 1.00
Weekend	evening rush	outbound	x 1.00
Weekend	overnight	inbound	x 1.00
Weekend	overnight	outbound	x 1.00

There are 16 different combinations of the three variables. By combining some of the conditions, you'll simplify the final switch expression.

The system that collects the tolls uses a DateTime structure for the time when the toll was collected. Build member methods that create the variables from the preceding table. The following function uses a pattern matching switch expression to express whether a DateTime represents a weekend or a weekday:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

That method works, but it's repetitious. You can simplify it, as shown in the following code:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday   => false,
        _                  => true
    };
```

Next, add a similar function to categorize the time into the blocks:

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll)
{
    int hour = timeOfToll.Hour;
    if (hour < 6)
        return TimeBand.Overnight;
    else if (hour < 10)
        return TimeBand.MorningRush;
    else if (hour < 16)
        return TimeBand.Daytime;
    else if (hour < 20)
        return TimeBand.EveningRush;
    else
        return TimeBand.Overnight;
}
```

The previous method doesn't use pattern matching. It's clearer using a familiar cascade of `if` statements. You do add a private `enum` to convert each range of time to a discrete value.

After you create those methods, you can use another `switch` expression with the **tuple pattern** to calculate the pricing premium. You could build a `switch` expression with all 16 arms:

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true,  TimeBand.MorningRush, true)  => 2.00m,
        (true,  TimeBand.MorningRush, false) => 1.00m,
        (true,  TimeBand.Daytime,     true)  => 1.50m,
        (true,  TimeBand.Daytime,     false) => 1.50m,
        (true,  TimeBand.EveningRush, true)  => 1.00m,
        (true,  TimeBand.EveningRush, false) => 2.00m,
        (true,  TimeBand.Overnight,   true)  => 0.75m,
        (true,  TimeBand.Overnight,   false) => 0.75m,
        (false, TimeBand.MorningRush, true)  => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime,     true)  => 1.00m,
        (false, TimeBand.Daytime,     false) => 1.00m,
        (false, TimeBand.EveningRush, true)  => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight,   true)  => 1.00m,
        (false, TimeBand.Overnight,   false) => 1.00m,
    };
```

The above code works, but it can be simplified. All eight combinations for the weekend have the same toll. You can replace all eight with the following line:

```
(false, _, _) => 1.0m,
```

Both inbound and outbound traffic have the same multiplier during the weekday daytime and overnight hours. Those four switch arms can be replaced with the following two lines:

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

The code should look like the following code after those two changes:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Finally, you can remove the two rush hour times that pay the regular price. Once you remove those arms, you can replace the `false` with a discard (`_`) in the final switch arm. You'll have the following finished method:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (true, TimeBand.Daytime,     _)     => 1.5m,
        (true, TimeBand.MorningRush, true)  => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        (_,    _,                    _)     => 1.0m,
    };
```

This example highlights one of the advantages of pattern matching: the pattern branches are evaluated in order. If you rearrange them so that an earlier branch handles one of your later cases, the compiler warns you about the unreachable code. Those language rules made it easier to do the preceding simplifications with confidence that the code didn't change.

Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes. The cloud is causing data and functionality to live apart. The *shape* of the data and the *operations* on it aren't necessarily described together. In this tutorial, you consumed existing data in entirely different ways from its original function. Pattern matching gave you the ability to write functionality that overrode those types, even though you couldn't extend them.

Next steps

You can download the finished code from the dotnet/samples GitHub repository. Explore patterns on your own and add this technique into your regular coding activities. Learning these techniques gives you another way to approach problems and create new functionality.

Console Application

3/22/2019 • 12 minutes to read • Edit Online

This tutorial teaches you a number of features in .NET Core and the C# language. You'll learn:

- The basics of the .NET Core Command Line Interface (CLI)
- The structure of a C# Console Application
- Console I/O
- The basics of File I/O APIs in .NET
- The basics of the Task-based Asynchronous Programming in .NET

You'll build an application that reads a text file, and echoes the contents of that text file to the console. The output to the console is paced to match reading it aloud. You can speed up or slow down the pace by pressing the '<' (less than) or '>' (greater than) keys.

There are a lot of features in this tutorial. Let's build them one by one.

Prerequisites

You'll need to setup your machine to run .NET Core. You can find the installation instructions on the .NET Core page. You can run this application on Windows, Linux, macOS or in a Docker container. You'll need to install your favorite code editor.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

Before you start making modifications, let's go through the steps to run the simple Hello World application. After creating the application, type `dotnet restore` at the command prompt. This command runs the NuGet package restore process. NuGet is a .NET package manager. This command downloads any of the missing dependencies for your project. As this is a new project, none of the dependencies are in place, so the first run will download the .NET Core framework. After this initial step, you will only need to run `dotnet restore` when you add new dependent packages, or update the versions of any of your dependencies.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

After restoring packages, you run `dotnet build`. This executes the build engine and creates your application executable. Finally, you execute `dotnet run` to run your application.

The simple Hello World application code is all in Program.cs. Open that file with your favorite text editor. We're about to make our first changes. At the top of the file, see a using statement:

```
using System;
```

This statement tells the compiler that any types from the `System` namespace are in scope. Like other Object Oriented languages you may have used, C# uses namespaces to organize types. This Hello World program is no different. You can see that the program is enclosed in the namespace with the name based on the name of the current directory. For this tutorial, let's change the name of the namespace to `TeleprompterConsole`:

```
namespace TeleprompterConsole
```

Reading and Echoing the File

The first feature to add is the ability to read a text file and display all that text to the console. First, let's add a text file. Copy the sampleQuotes.txt file from the GitHub repository for this sample into your project directory. This will serve as the script for your application. If you would like information on how to download the sample app for this topic, see the instructions in the Samples and Tutorials topic.

Next, add the following method in your `Program` class (right below the `Main` method):

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

This method uses types from two new namespaces. For this to compile you'll need to add the following two lines to the top of the file:

```
using System.Collections.Generic;
using System.IO;
```

The IEnumerable<T> interface is defined in the System.Collections.Generic namespace. The File class is defined in the System.IO namespace.

This method is a special type of C# method called an *Iterator method*. Enumerator methods return sequences that are evaluated lazily. That means each item in the sequence is generated as it is requested by the code consuming the sequence. Enumerator methods are methods that contain one or more `yield return` statements. The object returned by the `ReadFrom` method contains the code to generate each item in the sequence. In this example, that involves reading the next line of text from the source file, and returning that string. Each time the calling code requests the next item from the sequence, the code reads the next line of text from the file and returns it. When the file is completely read, the sequence indicates that there are no more items.

There are two other C# syntax elements that may be new to you. The `using` statement in this method manages resource cleanup. The variable that is initialized in the `using` statement (`reader`, in this example) must implement the IDisposable interface. That interface defines a single method, `Dispose`, that should be called when the resource should be released. The compiler generates that call when execution reaches the closing brace of the `using` statement. The compiler-generated code ensures that the resource is released even if an exception is thrown from the code in the block defined by the using statement.

The `reader` variable is defined using the `var` keyword. `var` defines an *implicitly typed local variable*. That means the type of the variable is determined by the compile-time type of the object assigned to the variable. Here, that is the return value from the OpenText(String) method, which is a StreamReader object.

Now, let's fill in the code to read the file in the `Main` method:

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Run the program (using `dotnet run`) and you can see every line printed out to the console.

Adding Delays and Formatting output

What you have is being displayed far too fast to read aloud. Now you need to add the delays in the output. As you start, you'll be building some of the core code that enables asynchronous processing. However, these first steps will follow a few anti-patterns. The anti-patterns are pointed out in comments as you add the code, and the code will be updated in later steps.

There are two steps to this section. First, you'll update the iterator method to return single words instead of entire lines. That's done with these modifications. Replace the `yield return line;` statement with the following code:

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Next, you need to modify how you consume the lines of the file, and add a delay after writing each word. Replace the `Console.WriteLine(line)` statement in the `Main` method with the following block:

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

The Task class is in the System.Threading.Tasks namespace, so you need to add that `using` statement at the top of file:

```
using System.Threading.Tasks;
```

Run the sample, and check the output. Now, each single word is printed, followed by a 200 ms delay. However, the displayed output shows some issues because the source text file has several lines that have more than 80 characters without a line break. That can be hard to read while it's scrolling by. That's easy to fix. You'll just keep track of the length of each line, and generate a new line whenever the line length reaches a certain threshold. Declare a local variable after the declaration of `words` in the `ReadFrom` method that holds the line length:

```
var lineLength = 0;
```

Then, add the following code after the `yield return word + " ";` statement (before the closing brace):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Run the sample, and you'll be able to read aloud at its pre-configured pace.

Async Tasks

In this final step, you'll add the code to write the output asynchronously in one task, while also running another task to read input from the user if they want to speed up or slow down the text display, or stop the text display altogether. This has a few steps in it and by the end, you'll have all the updates that you need. The first step is to create an asynchronous Task returning method that represents the code you've created so far to read and display the file.

Add this method to your `Program` class (it's taken from the body of your `Main` method):

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

You'll notice two changes. First, in the body of the method, instead of calling Wait() to synchronously wait for a task to finish, this version uses the `await` keyword. In order to do that, you need to add the `async` modifier to the method signature. This method returns a `Task`. Notice that there are no return statements that return a `Task` object. Instead, that `Task` object is created by code the compiler generates when you use the `await` operator. You can imagine that this method returns when it reaches an `await`. The returned `Task` indicates that the work has not completed. The method resumes when the awaited task completes. When it has executed to completion, the returned `Task` indicates that it is complete. Calling code can monitor that returned `Task` to determine when it has completed.

You can call this new method in your `Main` method:

```
ShowTeleprompter().Wait();
```

Here, in `Main`, the code does synchronously wait. You should use the `await` operator instead of synchronously waiting whenever possible. But, in a console application's `Main` method, you cannot use the `await` operator. That would result in the application exiting before all tasks have completed.

> **NOTE**
>
> If you use C# 7.1 or later, you can create console applications with `async` `Main` method.

Next, you need to write the second asynchronous method to read from the Console and watch for the '<' (less than), '>' (greater than) and 'X' or 'x' keys. Here's the method you add for that task:

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

This creates a lambda expression to represent an Action delegate that reads a key from the Console and modifies a local variable representing the delay when the user presses the '<' (less than) or '>' (greater than) keys. The delegate method finishes when user presses the 'X' or 'x' keys, which allow the user to stop the text display at any time. This method uses ReadKey() to block and wait for the user to press a key.

To finish this feature, you need to create a new `async Task` returning method that starts both of these tasks (`GetInput` and `ShowTeleprompter`), and also manages the shared data between these two tasks.

It's time to create a class that can handle the shared data between these two tasks. This class contains two public properties: the delay, and a flag `Done` to indicate that the file has been completely read:

```
namespace TeleprompterConsole
{
    internal class TelePrompterConfig
    {
        public int DelayInMilliseconds { get; private set; } = 200;

        public void UpdateDelay(int increment) // negative to speed up
        {
            var newDelay = Min(DelayInMilliseconds + increment, 1000);
            newDelay = Max(newDelay, 20);
            DelayInMilliseconds = newDelay;
        }

        public bool Done { get; private set; }

        public void SetDone()
        {
            Done = true;
        }
    }
}
```

Put that class in a new file, and enclose that class in the `TeleprompterConsole` namespace as shown above. You'll also need to add a `using static` statement so that you can reference the `Min` and `Max` methods without the enclosing class or namespace names. A `using static` statement imports the methods from one class. This is in contrast with the `using` statements used up to this point that have imported all classes from a namespace.

```
using static System.Math;
```

Next, you need to update the `ShowTeleprompter` and `GetInput` methods to use the new `config` object. Write one final `Task` returning `async` method to start both tasks and exit when the first task finishes:

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

The one new method here is the WhenAny(Task[]) call. That creates a `Task` that finishes as soon as any of the tasks in its argument list completes.

Next, you need to update both the `ShowTeleprompter` and `GetInput` methods to use the `config` object for the delay:

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

This new version of `ShowTeleprompter` calls a new method in the `TelepromperConfig` class. Now, you need to update `Main` to call `RunTeleprompter` instead of `ShowTeleprompter`:

```
RunTeleprompter().Wait();
```

Conclusion

This tutorial showed you a number of the features around the C# language and the .NET Core libraries related to working in Console applications. You can build on this knowledge to explore more about the language, and the classes introduced here. You've seen the basics of File and Console I/O, blocking and non-blocking use of the Task-based asynchronous programming, a tour of the C# language and how C# programs are organized and the .NET Core Command Line Interface and tools.

For more information about File I/O, see the File and Stream I/O topic. For more information about asynchronous programming model used in this tutorial, see the Task-based Asynchronous Programming topic and the Asynchronous programming topic.

REST client

5/6/2019 • 14 minutes to read • Edit Online

Introduction

This tutorial teaches you a number of features in .NET Core and the C# language. You'll learn:

- The basics of the .NET Core Command Line Interface (CLI).
- An overview of C# Language features.
- Managing dependencies with NuGet
- HTTP Communications
- Processing JSON information
- Managing configuration with Attributes.

You'll build an application that issues HTTP Requests to a REST service on GitHub. You'll read information in JSON format, and convert that JSON packet into C# objects. Finally, you'll see how to work with C# objects.

There are a lot of features in this tutorial. Let's build them one by one.

If you prefer to follow along with the final sample for this topic, you can download it. For download instructions, see Samples and Tutorials.

Prerequisites

You'll need to set up your machine to run .NET core. You can find the installation instructions on the .NET Core page. You can run this application on Windows, Linux, macOS or in a Docker container. You'll need to install your favorite code editor. The descriptions below use Visual Studio Code, which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application. As this is a new project, none of the dependencies are in place, so the first run will download the .NET Core framework, install a development certificate and run the NuGet package manager to restore missing dependencies.

Before you start making modifications, type `dotnet run` (see note) at the command prompt to run your application. `dotnet run` automatically performs `dotnet restore` if your environment is missing dependencies. It also performs `dotnet build` if your application needs to be rebuilt. After your initial setup, you will only need to run `dotnet restore` or `dotnet build` when it makes sense for your project.

Adding New Dependencies

One of the key design goals for .NET Core is to minimize the size of the .NET installation. If an application needs additional libraries for some of its features, you add those dependencies into your C# project (*.csproj) file. For our example, you'll need to add the `System.Runtime.Serialization.Json` package so your application can process JSON responses.

Open your `csproj` project file. The first line of the file should appear as:

```
<Project Sdk="Microsoft.NET.Sdk">
```

Add the following immediately after this line:

```
  <ItemGroup>
    <PackageReference Include="System.Runtime.Serialization.Json" Version="4.3.0" />
  </ItemGroup>
```

Most code editors will provide completion for different versions of these libraries. You'll usually want to use the latest version of any package that you add. However, it is important to make sure that the versions of all packages match, and that they also match the version of the .NET Core Application framework.

After you've made these changes, execute `dotnet restore` (see note) so that the package is installed on your system.

Making Web Requests

Now you're ready to start retrieving data from the web. In this application, you'll read information from the GitHub API. Let's read information about the projects under the .NET Foundation umbrella. You'll start by making the request to the GitHub API to retrieve information on the projects. The endpoint you'll use is: https://api.github.com/orgs/dotnet/repos. You want to retrieve all the information about these projects, so you'll use an HTTP GET request. Your browser also uses HTTP GET requests, so you can paste that URL into your browser to see what information you'll be receiving and processing.

You use the HttpClient class to make web requests. Like all modern .NET APIs, HttpClient supports only async methods for its long-running APIs. Start by making an async method. You'll fill in the implementation as you build the functionality of the application. Start by opening the `program.cs` file in your project directory and adding the following method to the `Program` class:

```
private static async Task ProcessRepositories()
{
}
```

You'll need to add a `using` statement at the top of your `Main` method so that the C# compiler recognizes the Task type:

```
using System.Threading.Tasks;
```

If you build your project at this point, you'll get a warning generated for this method, because it does not contain any `await` operators and will run synchronously. Ignore that for now; you'll add `await` operators as you fill in the method.

Next, rename the namespace defined in the `namespace` statement from its default of `ConsoleApp` to `WebAPIClient`. We'll later define a `repo` class in this namespace.

Next, update the `Main` method to call this method. The `ProcessRepositories` method returns a Task, and you shouldn't exit the program before that task finishes. Therefore, you must use the `Wait` method to block and wait for the task to finish:

```
static void Main(string[] args)
{
    ProcessRepositories().Wait();
}
```

Now, you have a program that does nothing, but does it asynchronously. Let's improve it.

First you need an object that is capable to retrieve data from the web; you can use a HttpClient to do that. This object handles the request and the responses. Instantiate a single instance of that type in the `Program` class inside the Program.cs file.

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static void Main(string[] args)
        {
            //...
        }
    }
}
```

Let's go back to the `ProcessRepositories` method and fill in a first version of it:

```
private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

You'll need to also add two new using statements at the top of the file for this to compile:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

This first version makes a web request to read the list of all repositories under the dotnet foundation organization. (The gitHub ID for the .NET Foundation is 'dotnet'). The first few lines set up the HttpClient for this request. First, it is configured to accept the GitHub JSON responses. This format is simply JSON. The next line adds a User Agent header to all requests from this object. These two headers are checked by the GitHub server code, and are necessary to retrieve information from GitHub.

After you've configured the HttpClient, you make a web request and retrieve the response. In this first version, you use the HttpClient.GetStringAsync(String) convenience method. This convenience method starts a task that makes the web request, and then when the request returns, it reads the response stream and extracts the content from the stream. The body of the response is returned as a String. The string is available when the task completes.

The final two lines of this method await that task, and then print the response to the console. Build the app, and run it. The build warning is gone now, because the `ProcessRepositories` now does contain an `await` operator.

You'll see a long display of JSON formatted text.

Processing the JSON Result

At this point, you've written the code to retrieve a response from a web server, and display the text that is contained in that response. Next, let's convert that JSON response into C# objects.

The JSON Serializer converts JSON data into C# Objects. Your first task is to define a C# class type to contain the information you use from this response. Let's build this slowly, so start with a simple C# type that contains the name of the repository:

```
using System;

namespace WebAPIClient
{
    public class repo
    {
        public string name;
    }
}
```

Put the above code in a new file called 'repo.cs'. This version of the class represents the simplest path to process JSON data. The class name and the member name match the names used in the JSON packet, instead of following C# conventions. You'll fix that by providing some configuration attributes later. This class demonstrates another important feature of JSON serialization and deserialization: Not all the fields in the JSON packet are part of this class. The JSON serializer will ignore information that is not included in the class type being used. This feature makes it easier to create types that work with only a subset of the fields in the JSON packet.

Now that you've created the type, let's deserialize it. You'll need to create a DataContractJsonSerializer object. This object must know the CLR type expected for the JSON packet it retrieves. The packet from GitHub contains a sequence of repositories, so a `List<repo>` is the correct type. Add the following line to your `ProcessRepositories` method:

```
var serializer = new DataContractJsonSerializer(typeof(List<repo>));
```

You're using two new namespaces, so you'll need to add those as well:

```
using System.Collections.Generic;
using System.Runtime.Serialization.Json;
```

Next, you'll use the serializer to convert JSON into C# objects. Replace the call to GetStringAsync(String) in your `ProcessRepositories` method with the following two lines:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = serializer.ReadObject(await streamTask) as List<repo>;
```

Notice that you're now using GetStreamAsync(String) instead of GetStringAsync(String). The serializer uses a stream instead of a string as its source. Let's explain a couple features of the C# language that are being used in the second line above. The argument to ReadObject(Stream) is an `await` expression. Await expressions can appear almost anywhere in your code, even though up to now, you've only seen them as part of an assignment statement.

Secondly, the `as` operator converts from the compile time type of `object` to `List<repo>`. The declaration of ReadObject(Stream) declares that it returns an object of type System.Object. ReadObject(Stream) will return the type you specified when you constructed it (`List<repo>` in this tutorial). If the conversion does not succeed, the

`as` operator evaluates to `null`, instead of throwing an exception.

You're almost done with this section. Now that you've converted the JSON to C# objects, let's display the name of each repository. Replace the lines that read:

```
var msg = await stringTask;   //**Deleted this
Console.Write(msg);
```

with the following:

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

Compile and run the application. It will print out the names of the repositories that are part of the .NET Foundation.

Controlling Serialization

Before you add more features, let's address the `repo` type and make it follow more standard C# conventions. You'll do this by annotating the `repo` type with *attributes* that control how the JSON Serializer works. In your case, you'll use these attributes to define a mapping between the JSON key names and the C# class and member names. The two attributes used are the DataContractAttribute and DataMemberAttribute attributes. By convention, all Attribute classes end in the suffix `Attribute`. However, you do not need to use that suffix when you apply an attribute.

The DataContractAttribute and DataMemberAttribute attributes are in a different library, so you'll need to add that library to your C# project file as a dependency. Add the following line to the `<ItemGroup>` section of your project file:

```
<PackageReference Include="System.Runtime.Serialization.Primitives" Version="4.3.0" />
```

After you save the file, run `dotnet restore` (see note) to retrieve this package.

Next, open the `repo.cs` file. Let's change the name to use Pascal Case, and fully spell out the name `Repository`. We still want to map JSON 'repo' nodes to this type, so you'll need to add the DataContractAttribute attribute to the class declaration. You'll set the `Name` property of the attribute to the name of the JSON nodes that map to this type:

```
[DataContract(Name="repo")]
public class Repository
```

The DataContractAttribute is a member of the System.Runtime.Serialization namespace, so you'll need to add the appropriate `using` statement at the top of the file:

```
using System.Runtime.Serialization;
```

You changed the name of the `repo` class to `Repository`, so you'll need to make the same name change in Program.cs (some editors may support a rename refactoring that will make this change automatically:)

```
var serializer = new DataContractJsonSerializer(typeof(List<Repository>));

// ...

var repositories = serializer.ReadObject(await streamTask) as List<Repository>;
```

Next, let's make the same change with the `name` field by using the DataMemberAttribute class. Make the following changes to the declaration of the `name` field in repo.cs:

```
[DataMember(Name="name")]
public string Name;
```

This change means you need to change the code that writes the name of each repository in program.cs:

```
Console.WriteLine(repo.Name);
```

Do a `dotnet build` followed by a `dotnet run` to make sure you've got the mappings correct. You should see the same output as before. Before we process more properties from the web server, let's make one more change to the `Repository` class. The `Name` member is a publicly accessible field. That's not a good object-oriented practice, so let's change it to a property. For our purposes, we don't need any specific code to run when getting or setting the property, but changing to a property makes it easier to add those changes later without breaking any code that uses the `Repository` class.

Remove the field definition, and replace it with an auto-implemented property:

```
public string Name { get; set; }
```

The compiler generates the body of the `get` and `set` accessors, as well as a private field to store the name. It would be similar to the following code that you could type by hand:

```
public string Name
{
    get { return this._name; }
    set { this._name = value; }
}
private string _name;
```

Let's make one more change before adding new features. The `ProcessRepositories` method can do the async work and return a collection of the repositories. Let's return the `List<Repository>` from that method, and move the code that writes the information into the `Main` method.

Change the signature of `ProcessRepositories` to return a task whose result is a list of `Repository` objects:

```
private static async Task<List<Repository>> ProcessRepositories()
```

Then, just return the repositories after processing the JSON response:

```
var repositories = serializer.ReadObject(await streamTask) as List<Repository>;
return repositories;
```

The compiler generates the `Task<T>` object for the return because you've marked this method as `async`. Then, let's modify the `Main` method so that it captures those results and writes each repository name to the console.

Your `Main` method now looks like this:

```
public static void Main(string[] args)
{
    var repositories = ProcessRepositories().Result;

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

Accessing the `Result` property of a Task blocks until the task has completed. Normally, you would prefer to `await` the completion of the task, as in the `ProcessRepositories` method, but that isn't allowed in the `Main` method.

Reading More Information

Let's finish this by processing a few more of the properties in the JSON packet that gets sent from the GitHub API. You won't want to grab everything, but adding a few properties will demonstrate a few more features of the C# language.

Let's start by adding a few more simple types to the `Repository` class definition. Add these properties to that class:

```
[DataMember(Name="description")]
public string Description { get; set; }

[DataMember(Name="html_url")]
public Uri GitHubHomeUrl { get; set; }

[DataMember(Name="homepage")]
public Uri Homepage { get; set; }

[DataMember(Name="watchers")]
public int Watchers { get; set; }
```

These properties have built-in conversions from the string type (which is what the JSON packets contain) to the target type. The Uri type may be new to you. It represents a URI, or in this case, a URL. In the case of the `Uri` and `int` types, if the JSON packet contains data that does not convert to the target type, the serialization action will throw an exception.

Once you've added these, update the `Main` method to display those elements:

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

As a final step, let's add the information for the last push operation. This information is formatted in this fashion in the JSON response:

```
2016-02-08T21:27:00Z
```

That format does not follow any of the standard .NET DateTime formats. Because of that, you'll need to write a

custom conversion method. You also probably don't want the raw string exposed to users of the `Repository` class. Attributes can help control that as well. First, define a `private` property that will hold the string representation of the date time in your `Repository` class:

```
[DataMember(Name="pushed_at")]
private string JsonDate { get; set; }
```

The DataMemberAttribute attribute informs the serializer that this should be processed, even though it is not a public member. Next, you need to write a public read-only property that converts the string to a valid DateTime object, and returns that DateTime:

```
[IgnoreDataMember]
public DateTime LastPush
{
    get
    {
        return DateTime.ParseExact(JsonDate, "yyyy-MM-ddTHH:mm:ssZ", CultureInfo.InvariantCulture);
    }
}
```

Let's go over the new constructs above. The `IgnoreDataMember` attribute instructs the serializer that this type should not be read to or written from any JSON object. This property contains only a `get` accessor. There is no `set` accessor. That's how you define a *read-only* property in C#. (Yes, you can create *write-only* properties in C#, but their value is limited.) The ParseExact(String, String, IFormatProvider) method parses a string and creates a DateTime object using a provided date format, and adds additional metadata to the `DateTime` using a `CultureInfo` object. If the parse operation fails, the property accessor throws an exception.

To use InvariantCulture, you will need to add the System.Globalization namespace to the `using` statements in `repo.cs`:

```
using System.Globalization;
```

Finally, add one more output statement in the console, and you're ready to build and run this app again:

```
Console.WriteLine(repo.LastPush);
```

Your version should now match the finished sample.

Conclusion

This tutorial showed you how to make web requests, parse the result, and display properties of those results. You've also added new packages as dependencies in your project. You've seen some of the features of the C# language that support object-oriented techniques.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Inheritance in C# and .NET

3/26/2019 • 26 minutes to read • Edit Online

This tutorial introduces you to inheritance in C#. Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

Prerequisites

This tutorial assumes that you've installed .NET Core. For installation instructions, see .NET Core installation guide. You also need a code editor. This tutorial uses Visual Studio Code, although you can use any code editor of your choice.

Running the examples

To create and run the examples in this tutorial, you use the dotnet utility from the command line. Follow these steps for each example:

1. Create a directory to store the example.
2. Enter the dotnet new console command at a command prompt to create a new .NET Core project.
3. Copy and paste the code from the example into your code editor.
4. Enter the dotnet restore command from the command line to load or restore the project's dependencies.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

1. Enter the dotnet run command to compile and execute the example.

Background: What is inheritance?

Inheritance is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the *base class*. The class that inherits the members of the base class is called the *derived class*.

C# and .NET support *single inheritance* only. That is, a class can only inherit from a single class. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types. In other words, type `D` can inherit from type `C`, which inherits from type `B`, which inherits from the base class type `A`. Because inheritance is transitive, the members of type `A` are available to type `D`.

Not all members of a base class are inherited by derived classes. The following members are not inherited:

- Static constructors, which initialize the static data of a class.
- Instance constructors, which you call to create a new instance of the class. Each class must define its own constructors.
- Finalizers, which are called by the runtime's garbage collector to destroy instances of a class.

While all other members of a base class are inherited by derived classes, whether they are visible or not depends on their accessibility. A member's accessibility affects its visibility for derived classes as follows:

- Private members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes. In the following example, `A.B` is a nested class that derives from `A`, and `C` derives from `A`. The private `A.value` field is visible in A.B. However, if you remove the comments from the `C.GetValue` method and attempt to compile the example, it produces compiler error CS0122: "'A.value' is inaccessible due to its protection level."

```
using System;

public class A
{
   private int value = 10;

   public class B : A
   {
       public int GetValue()
       {
           return this.value;
       }
   }
}

public class C : A
{
//    public int GetValue()
//    {
//        return this.value;
//    }
}

public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- Protected members are visible only in derived classes.
- Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
- Public members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class. In the following example, class `A` defines a method named `Method1`, and class `B` inherits from class `A`. The example then calls `Method1` as if it were an instance method on `B`.

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }


public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}
```

Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the virtual keyword. By default, base class members are not marked as `virtual` and cannot be overridden. Attempting to override a non-virtual member, as the following example does, generates compiler error CS0506: "<member> cannot override inherited member <member> because it is not marked virtual, abstract, or override.

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

In some cases, a derived class *must* override the base class implementation. Base class members marked with the abstract keyword require that derived classes override them. Attempting to compile the following example generates compiler error CS0534, "<class> does not implement inherited abstract member <member>", because class `B` provides no implementation for `A.Method1`.

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Because of these rules, attempting to compile code like the following example produces compiler error CS0527: "Type 'ValueType' in interface list is not an interface." The error message indicates that, although you can define the interfaces that a struct implements, inheritance is not supported.

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Implicit inheritance

Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from Object or a type derived from it. The common functionality of Object is available to any type.

To see what implicit inheritance means, let's define a new class, `SimpleClass`, that is simply an empty class definition:

```
public class SimpleClass
{ }
```

You can then use reflection (which lets you inspect a type's metadata to get information about that type) to get a list of the members that belong to the `SimpleClass` type. Although you haven't defined any members in your `SimpleClass` class, output from the example indicates that it actually has nine members. One of these members is a parameterless (or default) constructor that is automatically supplied for the `SimpleClass` type by the C# compiler. The remaining eight are members of Object, the type from which all classes and interfaces in the .NET type system ultimately implicitly inherit.

```
using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
                             BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by
{member.DeclaringType}";
            Console.WriteLine(output);

        }
    }
}
// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method):  Public, Declared by System.Object
// Equals (Method):  Public, Declared by System.Object
// Equals (Method):  Public Static, Declared by System.Object
// ReferenceEquals (Method):  Public Static, Declared by System.Object
// GetHashCode (Method):  Public, Declared by System.Object
// GetType (Method):  Public, Declared by System.Object
// Finalize (Method):  Internal, Declared by System.Object
// MemberwiseClone (Method):  Internal, Declared by System.Object
// .ctor (Constructor):  Public, Declared by SimpleClass
```

Implicit inheritance from the Object class makes these methods available to the `SimpleClass` class:

- The public `ToString` method, which converts a `SimpleClass` object to its string representation, returns the fully qualified type name. In this case, the `ToString` method returns the string "SimpleClass".
- Three methods that test for equality of two objects: the public instance `Equals(Object)` method, the public static `Equals(Object, Object)` method, and the public static `ReferenceEquals(Object, Object)` method. By default, these methods test for reference equality; that is, to be equal, two object variables must refer to the same object.
- The public `GetHashCode` method, which computes a value that allows an instance of the type to be used in hashed collections.

- The public `GetType` method, which returns a Type object that represents the `SimpleClass` type.
- The protected Finalize method, which is designed to release unmanaged resources before an object's memory is reclaimed by the garbage collector.
- The protected MemberwiseClone method, which creates a shallow clone of the current object.

Because of implicit inheritance, you can call any inherited member from a `SimpleClass` object just as if it was actually a member defined in the `SimpleClass` class. For instance, the following example calls the `SimpleClass.ToString` method, which `SimpleClass` inherits from Object.

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        SimpleClass
```

The following table lists the categories of types that you can create in C# and the types from which they implicitly inherit. Each base type makes a different set of members available through inheritance to implicitly derived types.

TYPE CATEGORY	IMPLICITLY INHERITS FROM
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
delegate	MulticastDelegate, Delegate, Object

Inheritance and an "is a" relationship

Ordinarily, inheritance is used to express an "is a" relationship between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the `Publication` class represents a publication of any kind, and the `Book` and `Magazine` classes represent specific types of publications.

> **NOTE**
> A class or struct can implement one or more interfaces. While interface implementation is often presented as a workaround for single inheritance or as a way of using inheritance with structs, it is intended to express a different relationship (a "can do" relationship) between an interface and its implementing type than inheritance. An interface defines a subset of functionality (such as the ability to test for equality, to compare or sort objects, or to support culture-sensitive parsing and formatting) that the interface makes available to its implementing types.

Note that "is a" also expresses the relationship between a type and a specific instantiation of that type. In the

following example, `Automobile` is a class that has three unique read-only properties: `Make`, the manufacturer of the automobile; `Model`, the kind of automobile; and `Year`, its year of manufacture. Your `Automobile` class also has a constructor whose arguments are assigned to the property values, and it overrides the Object.ToString method to produce a string that uniquely identifies the `Automobile` instance rather than the `Automobile` class.

```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
           throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrWhiteSpace(make))
           throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
           throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrWhiteSpace(model))
           throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
           throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

In this case, you shouldn't rely on inheritance to represent specific car makes and models. For example, you don't need to define a `Packard` type to represent automobiles manufactured by the Packard Motor Car Company. Instead, you can represent them by creating an `Automobile` object with the appropriate values passed to its class constructor, as the following example does.

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//        1948 Packard Custom Eight
```

An is-a relationship based on inheritance is best applied to a base class and to derived classes that add additional members to the base class or that require additional functionality not present in the base class.

Designing the base class and derived classes

Let's look at the process of designing a base class and its derived classes. In this section, you'll define a base class,

`Publication`, which represents a publication of any kind, such as a book, a magazine, a newspaper, a journal, an article, etc. You'll also define a `Book` class that derives from `Publication`. You could easily extend the example to define other derived classes, such as `Magazine`, `Journal`, `Newspaper`, and `Article`.

The base Publication class

In designing your `Publication` class, you need to make several design decisions:

- What members to include in your base `Publication` class, and whether the `Publication` members provide method implementations or whether `Publication` is an abstract base class that serves as a template for its derived classes.

  In this case, the `Publication` class will provide method implementations. The Designing abstract base classes and their derived classes section contains an example that uses an abstract base class to define the methods that derived classes must override. Derived classes are free to provide any implementation that is suitable for the derived type.

  The ability to reuse code (that is, multiple derived classes share the declaration and implementation of base class methods and do not need to override them) is an advantage of non-abstract base classes. Therefore, you should add members to `Publication` if their code is likely to be shared by some or most specialized `Publication` types. If you fail to provide base class implementations efficiently, you'll end up having to provide largely identical member implementations in derived classes rather a single implementation in the base class. The need to maintain duplicated code in multiple locations is a potential source of bugs.

  Both to maximize code reuse and to create a logical and intuitive inheritance hierarchy, you want to be sure that you include in the `Publication` class only the data and functionality that is common to all or to most publications. Derived classes then implement members that are unique to the particular kinds of publication that they represent.

- How far to extend your class hierarchy. Do you want to develop a hierarchy of three or more classes, rather than simply a base class and one or more derived classes? For example, `Publication` could be a base class of `Periodical`, which in turn is a base class of `Magazine`, `Journal` and `Newspaper`.

  For your example, you'll use the small hierarchy of a `Publication` class and a single derived class, `Book`. You could easily extend the example to create a number of additional classes that derive from `Publication`, such as `Magazine` and `Article`.

- Whether it makes sense to instantiate the base class. If it does not, you should apply the abstract keyword to the class. Otherwise, your `Publication` class can be instantiated by calling its class constructor. If an attempt is made to instantiate a class marked with the `abstract` keyword by a direct call to its class constructor, the C# compiler generates error CS0144, "Cannot create an instance of the abstract class or interface." If an attempt is made to instantiate the class by using reflection, the reflection method throws a MemberAccessException.

  By default, a base class can be instantiated by calling its class constructor. You do not have to explicitly define a class constructor. If one is not present in the base class' source code, the C# compiler automatically provides a default (parameterless) constructor.

  For your example, you'll mark the `Publication` class as abstract so that it cannot be instantiated. An `abstract` class without any `abstract` methods indicates that this class represents an abstract concept that is shared among several concrete classes (like a `Book`, `Journal`).

- Whether derived classes must inherit the base class implementation of particular members, whether they have the option to override the base class implementation, or whether they must provide an implementation. You use the abstract keyword to force derived classes to provide an implementation. You use the virtual keyword to allow derived classes to override a base class method. By default, methods defined in the base class are *not* overridable.

The `Publication` class does not have any `abstract` methods, but the class itself is `abstract`.

- Whether a derived class represents the final class in the inheritance hierarchy and cannot itself be used as a base class for additional derived classes. By default, any class can serve as a base class. You can apply the sealed keyword to indicate that a class cannot serve as a base class for any additional classes. Attempting to derive from a sealed class generated compiler error CS0509, "cannot derive from sealed type <typeName>".

  For your example, you'll mark your derived class as `sealed`.

The following example shows the source code for the `Publication` class, as well as a `PublicationType` enumeration that is returned by the `Publication.PublicationType` property. In addition to the members that it inherits from Object, the `Publication` class defines the following unique members and member overrides:

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
   private bool published = false;
   private DateTime datePublished;
   private int totalPages;

   public Publication(string title, string publisher, PublicationType type)
   {
      if (publisher == null)
         throw new ArgumentNullException("The publisher cannot be null.");
      else if (String.IsNullOrWhiteSpace(publisher))
         throw new ArgumentException("The publisher cannot consist only of white space.");
      Publisher = publisher;

      if (title == null)
         throw new ArgumentNullException("The title cannot be null.");
      else if (String.IsNullOrWhiteSpace(title))
         throw new ArgumentException("The title cannot consist only of white space.");
      Title = title;

      Type = type;
   }

   public string Publisher { get; }

   public string Title { get; }

   public PublicationType Type { get; }

   public string CopyrightName { get; private set; }

   public int CopyrightDate { get; private set; }

   public int Pages
   {
     get { return totalPages; }
     set
     {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
         totalPages = value;
     }
   }

   public string GetPublicationDate()
   {
      if (!published)
```

```
            return "NYP";
         else
            return datePublished.ToString("d");
      }

      public void Publish(DateTime datePublished)
      {
         published = true;
         this.datePublished = datePublished;
      }

      public void Copyright(string copyrightName, int copyrightDate)
      {
         if (copyrightName == null)
            throw new ArgumentNullException("The name of the copyright holder cannot be null.");
         else if (String.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder cannot consist only of white space.");
         CopyrightName = copyrightName;

         int currentYear = DateTime.Now.Year;
         if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
            throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and
{currentYear + 1}");
         CopyrightDate = copyrightDate;
      }

      public override string ToString() => Title;
   }
```

- A constructor

  Because the `Publication` class is `abstract`, it cannot be instantiated directly from code like the following example:

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                    PublicationType.Book);
  ```

  However, its instance constructor can be called directly from derived class constructors, as the source code for the `Book` class shows.

- Two publication-related properties

  `Title` is a read-only String property whose value is supplied by calling the `Publication` constructor.

  `Pages` is a read-write Int32 property that indicates how many total pages the publication has. The value is stored in a private field named `totalPages`. It must be a positive number or an ArgumentOutOfRangeException is thrown.

- Publisher-related members

  Two read-only properties, `Publisher` and `Type`. The values are originally supplied by the call to the `Publication` class constructor.

- Publishing-related members

  Two methods, `Publish` and `GetPublicationDate`, set and return the publication date. The `Publish` method sets a private `published` flag to `true` when it is called and assigns the date passed to it as an argument to the private `datePublished` field. The `GetPublicationDate` method returns the string "NYP" if the `published` flag is `false`, and the value of the `datePublished` field if it is `true`.

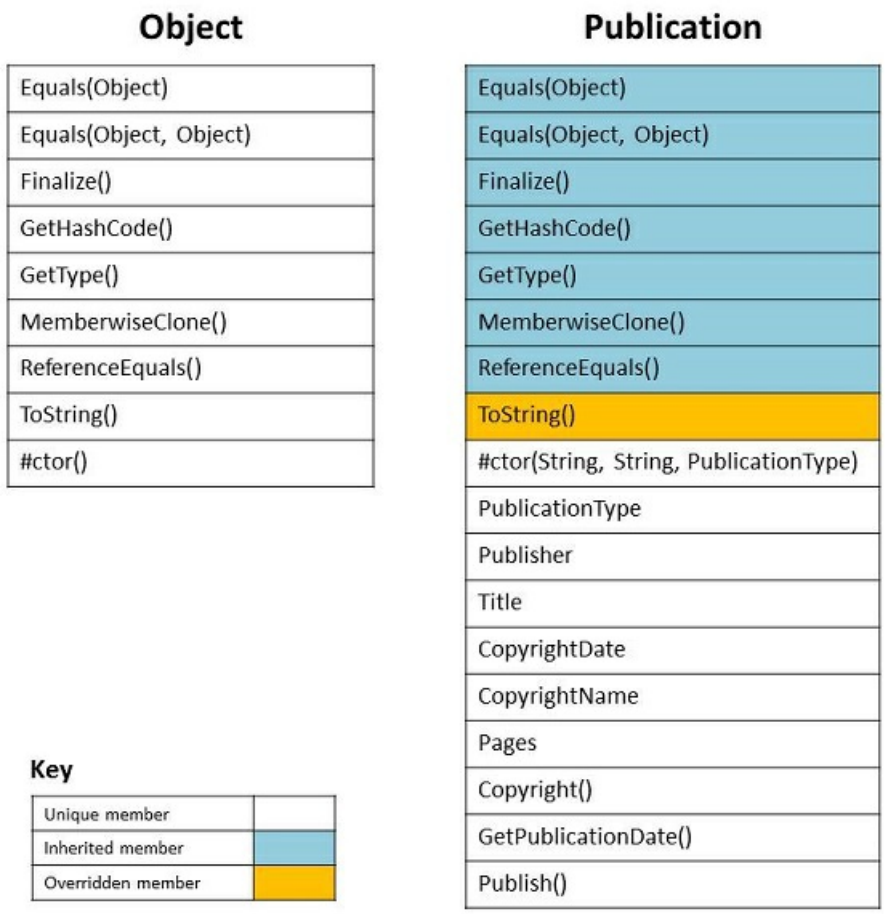
- Copyright-related members

The `Copyright` method takes the name of the copyright holder and the year of the copyright as arguments and assigns them to the `CopyrightName` and `CopyrightDate` properties.

- An override of the `ToString` method

  If a type does not override the Object.ToString method, it returns the fully qualified name of the type, which is of little use in differentiating one instance from another. The `Publication` class overrides Object.ToString to return the value of the `Title` property.

The following figure illustrates the relationship between your base `Publication` class and its implicitly inherited Object class.



The `Book` class

The `Book` class represents a book as a specialized type of publication. The following example shows the source code for the `Book` class.

```
using System;

public sealed class Book : Publication
{
   public Book(string title, string author, string publisher) :
          this(title, String.Empty, author, publisher)
   { }

   public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
   {
      // isbn argument must be a 10- or 13-character numeric string without "-" characters.
      // We could also determine whether the ISBN is valid by comparing its checksum digit
      // with a computed checksum.
      //
      if (! String.IsNullOrEmpty(isbn)) {
        // Determine if ISBN length is correct.
        if (! (isbn.Length == 10 | isbn.Length == 13))
            throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
        ulong nISBN = 0;
        if (! UInt64.TryParse(isbn, out nISBN))
            throw new ArgumentException("The ISBN can consist of numeric characters only.");
      }
      ISBN = isbn;

      Author = author;
   }

   public string ISBN { get; }

   public string Author { get; }

   public Decimal Price { get; private set; }

   // A three-digit ISO currency symbol.
   public string Currency { get; private set; }


   // Returns the old price, and sets a new price.
   public Decimal SetPrice(Decimal price, string currency)
   {
       if (price < 0)
          throw new ArgumentOutOfRangeException("The price cannot be negative.");
       Decimal oldValue = Price;
       Price = price;

       if (currency.Length != 3)
          throw new ArgumentException("The ISO currency symbol is a 3-character string.");
       Currency = currency;

       return oldValue;
   }

   public override bool Equals(object obj)
   {
      Book book = obj as Book;
      if (book == null)
         return false;
      else
         return ISBN == book.ISBN;
   }

   public override int GetHashCode() => ISBN.GetHashCode();

   public override string ToString() => $"{(String.IsNullOrEmpty(Author) ? "" : Author + ", ")}{Title}";
}
```

In addition to the members that it inherits from `Publication`, the `Book` class defines the following unique members and member overrides:

- Two constructors

  The two `Book` constructors share three common parameters. Two, *title* and *publisher*, correspond to parameters of the `Publication` constructor. The third is *author*, which is stored to a public immutable `Author` property. One constructor includes an *isbn* parameter, which is stored in the `ISBN` auto-property.

  The first constructor uses the this keyword to call the other constructor. Constructor chaining is a common pattern in defining constructors. Constructors with fewer parameters provide default values when calling the constructor with the greatest number of parameters.

  The second constructor uses the base keyword to pass the title and publisher name to the base class constructor. If you don't make an explicit call to a base class constructor in your source code, the C# compiler automatically supplies a call to the base class' default or parameterless constructor.

- A read-only `ISBN` property, which returns the `Book` object's International Standard Book Number, a unique 10- or 13-digit number. The ISBN is supplied as an argument to one of the `Book` constructors. The ISBN is stored in a private backing field, which is auto-generated by the compiler.

- A read-only `Author` property. The author name is supplied as an argument to both `Book` constructors and is stored in the property.

- Two read-only price-related properties, `Price` and `Currency`. Their values are provided as arguments in a `SetPrice` method call. The `Currency` property is the three-digit ISO currency symbol (for example, USD for the U.S. dollar). ISO currency symbols can be retrieved from the ISOCurrencySymbol property. Both of these properties are externally read-only, but both can be set by code in the `Book` class.

- A `SetPrice` method, which sets the values of the `Price` and `Currency` properties. Those values are returned by those same properties.

- Overrides to the `ToString` method (inherited from `Publication`) and the Object.Equals(Object) and GetHashCode methods (inherited from Object).

  Unless it is overridden, the Object.Equals(Object) method tests for reference equality. That is, two object variables are considered to be equal if they refer to the same object. In the `Book` class, on the other hand, two `Book` objects should be equal if they have the same ISBN.

  When you override the Object.Equals(Object) method, you must also override the GetHashCode method, which returns a value that the runtime uses to store items in hashed collections for efficient retrieval. The hash code should return a value that's consistent with the test for equality. Since you've overridden Object.Equals(Object) to return `true` if the ISBN properties of two `Book` objects are equal, you return the hash code computed by calling the GetHashCode method of the string returned by the `ISBN` property.

The following figure illustrates the relationship between the `Book` class and `Publication`, its base class.

You can now instantiate a `Book` object, invoke both its unique and inherited members, and pass it as an argument to a method that expects a parameter of type `Publication` or of type `Book`, as the following example shows.

```
using System;
using static System.Console;

public class Example
{
   public static void Main()
   {
      var book = new Book("The Tempest",  "0971655819", "Shakespeare, William",
                          "Public Domain Press");
      ShowPublicationInfo(book);
      book.Publish(new DateTime(2016, 8, 18));
      ShowPublicationInfo(book);

      var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
      Write($"{book.Title} and {book2.Title} are the same publication: " +
            $"{((Publication) book).Equals(book2)}");
   }

   public static void ShowPublicationInfo(Publication pub)
   {
       string pubDate = pub.GetPublicationDate();
       WriteLine($"{pub.Title}, " +
                 $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}");
   }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Designing abstract base classes and their derived classes

In the previous example, you defined a base class that provided an implementation for a number of methods to allow derived classes to share code. In many cases, however, the base class is not expected to provide an implementation. Instead, the base class is an *abstract class* that declares *abstract methods*; it serves as a template that defines the members that each derived class must implement. Typically in an abstract base class, the implementation of each derived type is unique to that type. You marked the class with the abstract keyword because it made no sense to instantiate a `Publication` object, although the class did provide implementations of functionality common to publications.

For example, each closed two-dimensional geometric shape includes two properties: area, the inner extent of the shape; and perimeter, or the distance along the edges of the shape. The way in which these properties are calculated, however, depends completely on the specific shape. The formula for calculating the perimeter (or circumference) of a circle, for example, is different from that of a triangle. The `Shape` class is an `abstract` class with `abstract` methods. That indicates derived classes share the same functionality, but those derived classes implement that functionality differently.

The following example defines an abstract base class named `Shape` that defines two properties: `Area` and `Perimeter`. In addition to marking the class with the abstract keyword, each instance member is also marked with the abstract keyword. In this case, `Shape` also overrides the Object.ToString method to return the name of the type, rather than its fully qualified name. And it defines two static members, `GetArea` and `GetPerimeter`, that allow callers to easily retrieve the area and perimeter of an instance of any derived class. When you pass an instance of a derived class to either of these methods, the runtime calls the method override of the derived class.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }

   public abstract double Perimeter { get; }

   public override string ToString() => GetType().Name;

   public static double GetArea(Shape shape) => shape.Area;

   public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

You can then derive some classes from `Shape` that represent specific shapes. The following example defines three classes, `Triangle`, `Rectangle`, and `Circle`. Each uses a formula unique for that particular shape to compute the area and perimeter. Some of the derived classes also define properties, such as `Rectangle.Diagonal` and `Circle.Diameter`, that are unique to the shape that they represent.

```
using System;

public class Square : Shape
{
   public Square(double length)
   {
      Side = length;
   }

   public double Side { get; }

   public override double Area => Math.Pow(Side, 2);

   public override double Perimeter => Side * 4;

   public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; }

   public double Width { get; }

   public override double Area => Length * Width;

   public override double Perimeter => 2 * Length + 2 * Width;

   public bool IsSquare() => Length == Width;

   public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

   public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

   // Define a circumference, since it's the more familiar term.
   public double Circumference => Perimeter;

   public double Radius { get; }

   public double Diameter => Radius * 2;
}
```

The following example uses objects derived from `Shape`. It instantiates an array of objects derived from `Shape` and calls the static methods of the `Shape` class, which wraps return `Shape` property values. The runtime retrieves values from the overridden properties of the derived types. The example also casts each `Shape` object in the array to its derived type and, if the cast succeeds, retrieves properties of that particular subclass of `Shape`.

```
using System;

public class Example
{
   public static void Main()
   {
      Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                         new Circle(3) };
      foreach (var shape in shapes) {
         Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                           $"perimeter, {Shape.GetPerimeter(shape)}");
         var rect = shape as Rectangle;
         if (rect != null) {
            Console.WriteLine($"   Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
            continue;
         }
         var sq = shape as Square;
         if (sq != null) {
            Console.WriteLine($"   Diagonal: {sq.Diagonal}");
            continue;
         }
      }
   }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

See also

- Classes and objects
- Inheritance (C# Programming Guide)

Working with LINQ

8/19/2019 • 15 minutes to read • Edit Online

Introduction

This tutorial teaches you features in .NET Core and the C# language. You'll learn:

- How to generate sequences with LINQ.
- How to write methods that can be easily used in LINQ queries.
- How to distinguish between eager and lazy evaluation.

You'll learn these techniques by building an application that demonstrates one of the basic skills of any magician: the faro shuffle. Briefly, a faro shuffle is a technique where you split a card deck exactly in half, then the shuffle interleaves each one card from each half to rebuild the original deck.

Magicians use this technique because every card is in a known location after each shuffle, and the order is a repeating pattern.

For your purposes, it is a light hearted look at manipulating sequences of data. The application you'll build will construct a card deck, and then perform a sequence of shuffles, writing the sequence out each time. You'll also compare the updated order to the original order.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also see the completed sample in the dotnet/samples GitHub repository. For download instructions, see Samples and Tutorials.

Prerequisites

You'll need to setup your machine to run .NET core. You can find the installation instructions on the .NET Core page. You can run this application on Windows, Ubuntu Linux, OS X or in a Docker container. You'll need to install your favorite code editor. The descriptions below use Visual Studio Code which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

If you've never used C# before, this tutorial explains the structure of a C# program. You can read that and then return here to learn more about LINQ.

Creating the Data Set

Before you begin, make sure that the following lines are at the top of the `Program.cs` file generated by `dotnet new console`:

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

If these three lines (`using` statements) aren't at the top of the file, our program will not compile.

Now that you have all of the references that you'll need, consider what constitutes a deck of cards. Commonly, a deck of playing cards has four suits, and each suit has thirteen values. Normally, you might consider creating a `Card` class right off the bat and populating a collection of `Card` objects by hand. With LINQ, you can be more concise than the usual way of dealing with creating a deck of cards. Instead of creating a `Card` class, you can create two sequences to represent suits and ranks, respectively. You'll create a really simple pair of *iterator methods* that will generate the ranks and suits as IEnumerable<T>s of strings:

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Place these underneath the `Main` method in your `Program.cs` file. These two methods both utilize the `yield return` syntax to produce a sequence as they run. The compiler builds an object that implements IEnumerable<T> and generates the sequence of strings as they are requested.

Now, use these iterator methods to create the deck of cards. You'll place the LINQ query in our `Main` method. Here's a look at it:

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

The multiple `from` clauses produce a SelectMany, which creates a single sequence from combining each element in the first sequence with each element in the second sequence. The order is important for our purposes. The first element in the first source sequence (Suits) is combined with every element in the second sequence (Ranks). This

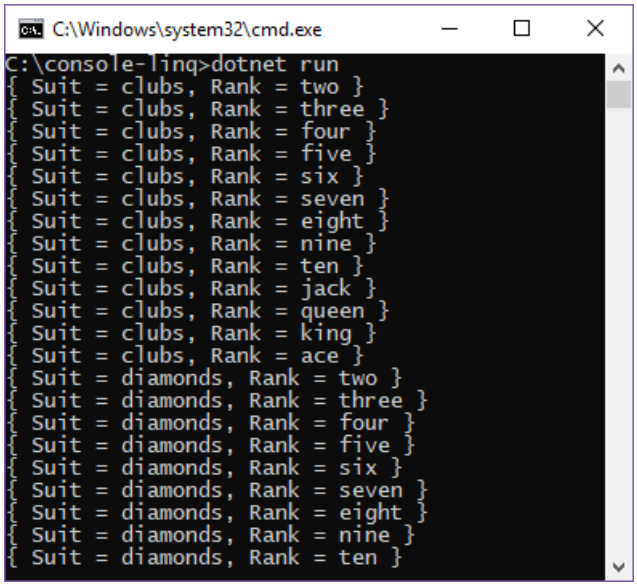
produces all thirteen cards of first suit. That process is repeated with each element in the first sequence (Suits). The end result is a deck of cards ordered by suits, followed by values.

It's important to keep in mind that whether you choose to write your LINQ in the query syntax used above or use method syntax instead, it's always possible to go from one form of syntax to the other. The above query written in query syntax can be written in method syntax as:

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

The compiler translates LINQ statements written with query syntax into the equivalent method call syntax. Therefore, regardless of your syntax choice, the two versions of the query produce the same result. Choose which syntax works best for your situation: for instance, if you're working in a team where some of the members have difficulty with method syntax, try to prefer using query syntax.

Go ahead and run the sample you've built at this point. It will display all 52 cards in the deck. You may find it very helpful to run this sample under a debugger to observe how the `Suits()` and `Ranks()` methods execute. You can clearly see that each string in each sequence is generated only as it is needed.



Manipulating the Order

Next, focus on how you're going to shuffle the cards in the deck. The first step in any good shuffle is to split the deck in two. The Take and Skip methods that are part of the LINQ APIs provide that feature for you. Place them underneath the `foreach` loop:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

However, there's no shuffle method to take advantage of in the standard library, so you'll have to write your own. The shuffle method you'll be creating illustrates several techniques that you'll use with LINQ-based programs, so each part of this process will be explained in steps.

In order to add some functionality to how you interact with the IEnumerable<T> you'll get back from LINQ queries, you'll need to write some special kinds of methods called extension methods. Briefly, an extension method is a special purpose *static method* that adds new functionality to an already-existing type without having to modify the original type you want to add functionality to.

Give your extension methods a new home by adding a new *static* class file to your program called `Extensions.cs`, and then start building out the first extension method:

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T>
second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Look at the method signature for a moment, specifically the parameters:

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

You can see the addition of the `this` modifier on the first argument to the method. That means you call the method as though it were a member method of the type of the first argument. This method declaration also follows a standard idiom where the input and output types are `IEnumerable<T>`. That practice enables LINQ methods to be chained together to perform more complex queries.

Naturally, since you split the deck into halves, you'll need to join those halves together. In code, this means you'll be enumerating both of the sequences you acquired through Take and Skip at once, *`interleaving`* the elements, and creating one sequence: your now-shuffled deck of cards. Writing a LINQ method that works with two sequences requires that you understand how IEnumerable<T> works.

The IEnumerable<T> interface has one method: GetEnumerator. The object returned by GetEnumerator has a method to move to the next element, and a property that retrieves the current element in the sequence. You will use those two members to enumerate the collection and return the elements. This Interleave method will be an iterator method, so instead of building a collection and returning the collection, you'll use the `yield return` syntax shown above.

Here's the implementation of that method:

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Now that you've written this method, go back to the `Main` method and shuffle the deck once:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Comparisons

How many shuffles it takes to set the deck back to its original order? To find out, you'll need to write a method that determines if two sequences are equal. After you have that method, you'll need to place the code that shuffles the deck in a loop, and check to see when the deck is back in order.

Writing a method to determine if the two sequences are equal should be straightforward. It's a similar structure to the method you wrote to shuffle the deck. Only this time, instead of `yield return`ing each element, you'll compare the matching elements of each sequence. When the entire sequence has been enumerated, if every element matches, the sequences are the same:

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

This shows a second LINQ idiom: terminal methods. They take a sequence as input (or in this case, two sequences), and return a single scalar value. When using terminal methods, they are always the final method in a chain of methods for a LINQ query, hence the name "terminal".

You can see this in action when you use it to determine when the deck is back in its original order. Put the shuffle code inside a loop, and stop when the sequence is back in its original order by applying the `SequenceEquals()` method. You can see it would always be the final method in any query, because it returns a single value instead of a sequence:

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Run the code you've got so far and take note of how the deck rearranges on each shuffle. After 8 shuffles (iterations of the do-while loop), the deck returns to the original configuration it was in when you first created it from the starting LINQ query.

Optimizations

The sample you've built so far executes an *out shuffle*, where the top and bottom cards stay the same on each run. Let's make one change: we'll use an *in shuffle* instead, where all 52 cards change position. For an in shuffle, you interleave the deck so that the first card in the bottom half becomes the first card in the deck. That means the last

card in the top half becomes the bottom card. This is a simple change to a singular line of code. Update the current shuffle query by switching the positions of Take and Skip. This will change the order of the top and bottom halves of the deck:

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Run the program again, and you'll see that it takes 52 iterations for the deck to reorder itself. You'll also start to notice some serious performance degradations as the program continues to run.

There are a number of reasons for this. You can tackle one of the major causes of this performance drop: inefficient use of *lazy evaluation*.

Briefly, lazy evaluation states that the evaluation of a statement is not performed until its value is needed. LINQ queries are statements that are evaluated lazily. The sequences are generated only as the elements are requested. Usually, that's a major benefit of LINQ. However, in a use such as this program, this causes exponential growth in execution time.

Remember that we generated the original deck using a LINQ query. Each shuffle is generated by performing three LINQ queries on the previous deck. All these are performed lazily. That also means they are performed again each time the sequence is requested. By the time you get to the 52nd iteration, you're regenerating the original deck many, many times. Let's write a log to demonstrate this behavior. Then, you'll fix it.

In your `Extensions.cs` file, type in or copy the method below. This extension method creates a new file called `debug.log` within your project directory and records what query is currently being executed to the log file. This extension method can be appended to any query to mark that the query executed.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

You will see a red squiggle under `File`, meaning it doesn't exist. It won't compile, since the compiler doesn't know what `File` is. To solve this problem, make sure to add the following line of code under the very first line in `Extensions.cs`:

```
using System.IO;
```

This should solve the issue and the red error disappears.

Next, instrument the definition of each query with a log message:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Notice that you don't log every time you access a query. You log only when you create the original query. The program still takes a long time to run, but now you can see why. If you run out of patience running the in shuffle with logging turned on, switch back to the out shuffle. You'll still see the lazy evaluation effects. In one run, it executes 2592 queries, including all the value and suit generation.

You can improve the performance of the code here to reduce the number of executions you make. A simple fix you can make is to *cache* the results of the original LINQ query that constructs the deck of cards. Currently, you're executing the queries again and again every time the do-while loop goes through an iteration, re-constructing the deck of cards and reshuffling it every time. To cache the deck of cards, you can leverage the LINQ methods ToArray and ToList; when you append them to the queries, they'll perform the same actions you've told them to, but now they'll store the results in an array or a list, depending on which method you choose to call. Append the LINQ method ToArray to both queries and run the program again:

```
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Now the out shuffle is down to 30 queries. Run again with the in shuffle and you'll see similar improvements: it now executes 162 queries.

Please note that this example is **designed** to highlight the use cases where lazy evaluation can cause performance difficulties. While it's important to see where lazy evaluation can impact code performance, it's equally important to understand that not all queries should run eagerly. The performance hit you incur without using ToArray is because each new arrangement of the deck of cards is built from the previous arrangement. Using lazy evaluation means each new deck configuration is built from the original deck, even executing the code that built the `startingDeck`. That causes a large amount of extra work.

In practice, some algorithms run well using eager evaluation, and others run well using lazy evaluation. For daily usage, lazy evaluation is usually a better choice when the data source is a separate process, like a database engine. For databases, lazy evaluation allows more complex queries to execute only one round trip to the database process and back to the rest of your code. LINQ is flexible whether you choose to utilize lazy or eager evaluation, so measure your processes and pick whichever kind of evaluation gives you the best performance.

Conclusion

In this project, you covered:

- using LINQ queries to aggregate data into a meaningful sequence
- writing Extension methods to add our own custom functionality to LINQ queries
- locating areas in our code where our LINQ queries might run into performance issues like degraded speed
- lazy and eager evaluation in regards to LINQ queries and the implications they might have on query performance

Aside from LINQ, you learned a bit about a technique magicians use for card tricks. Magicians use the Faro shuffle because they can control where every card moves in the deck. Now that you know, don't spoil it for everyone else!

For more information on LINQ, see:

- Language Integrated Query (LINQ)
  - Introduction to LINQ
  - Basic LINQ Query Operations (C#)
  - Data Transformations With LINQ (C#)
  - Query Syntax and Method Syntax in LINQ (C#)
  - C# Features That Support LINQ

Using Attributes in C#

4/11/2019 • 7 minutes to read • Edit Online

Attributes provide a way of associating information with code in a declarative way. They can also provide a reusable element that can be applied to a variety of targets.

Consider the `[Obsolete]` attribute. It can be applied to classes, structs, methods, constructors, and more. It *declares* that the element is obsolete. It's then up to the C# compiler to look for this attribute, and do some action in response.

In this tutorial, you'll be introduced to how to add attributes to your code, how to create and use your own attributes, and how to use some attributes that are built into .NET Core.

Prerequisites

You'll need to setup your machine to run .NET core. You can find the installation instructions on the .NET Core page. You can run this application on Windows, Ubuntu Linux, macOS or in a Docker container. You'll need to install your favorite code editor. The descriptions below use Visual Studio Code which is an open source, cross platform editor. However, you can use whatever tools you are comfortable with.

Create the Application

Now that you've installed all the tools, create a new .NET Core application. To use the command line generator, execute the following command in your favorite shell:

```
dotnet new console
```

This command will create barebones .NET core project files. You will need to execute `dotnet restore` to restore the dependencies needed to compile this project.

> **NOTE**
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

To execute the program, use `dotnet run`. You should see "Hello, World" output to the console.

How to add attributes to code

In C#, attributes are classes that inherit from the `Attribute` base class. Any class that inherits from `Attribute` can be used as a sort of "tag" on other pieces of code. For instance, there is an attribute called `ObsoleteAttribute`. This is used to signal that code is obsolete and shouldn't be used anymore. You can place this attribute on a class, for instance, by using square brackets.

```
[Obsolete]
public class MyClass
{

}
```

Note that while the class is called `ObsoleteAttribute`, it's only necessary to use `[Obsolete]` in the code. This is a convention that C# follows. You can use the full name `[ObsoleteAttribute]` if you choose.

When marking a class obsolete, it's a good idea to provide some information as to *why* it's obsolete, and/or *what* to use instead. Do this by passing a string parameter to the Obsolete attribute.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{

}
```

The string is being passed as an argument to an `ObsoleteAttribute` constructor, just as if you were writing `var attr = new ObsoleteAttribute("some string")`.

Parameters to an attribute constructor are limited to simple types/literals: `bool, int, double, string, Type, enums, etc` and arrays of those types. You can not use an expression or a variable. You are free to use positional or named parameters.

How to create your own attribute

Creating an attribute is as simple as inheriting from the `Attribute` base class.

```
public class MySpecialAttribute : Attribute
{

}
```

With the above, I can now use `[MySpecial]` (or `[MySpecialAttribute]`) as an attribute elsewhere in the code base.

```
[MySpecial]
public class SomeOtherClass
{

}
```

Attributes in the .NET base class library like `ObsoleteAttribute` trigger certain behaviors within the compiler. However, any attribute you create acts only as metadata, and doesn't result in any code within the attribute class being executed. It's up to you to act on that metadata elsewhere in your code (more on that later in the tutorial).

There is a 'gotcha' here to watch out for. As mentioned above, only certain types are allowed to be passed as arguments when using attributes. However, when creating an attribute type, the C# compiler won't stop you from creating those parameters. In the below example, I've created an attribute with a constructor that compiles just fine.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {

    }
}
```

However, you will be unable to use this constructor with attribute syntax.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{

}
```

The above will cause a compiler error like

`Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

How to restrict attribute usage

Attributes can be used on a number of "targets". The above examples show them on classes, but they can also be used on:

- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

When you create an attribute class, by default, C# will allow you to use that attribute on any of the possible attribute targets. If you want to restrict your attribute to certain targets, you can do so by using the `AttributeUsageAttribute` on your attribute class. That's right, an attribute on an attribute!

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{

}
```

If you attempt to put the above attribute on something that's not a class or a struct, you will get a compiler error like

`Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

How to use attributes attached to a code element

Attributes act as metadata. Without some outward force, they won't actually do anything.

To find and act on attributes, Reflection is generally needed. I won't cover Reflection in-depth in this tutorial, but the basic idea is that Reflection allows you to write code in C# that examines other code.

For instance, you can use Reflection to get information about a class(add `using System.Reflection;` at the head of your code):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

That will print out something like:

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

Once you have a `TypeInfo` object (or a `MemberInfo`, `FieldInfo`, etc), you can use the `GetCustomAttributes` method. This will return a collection of `Attribute` objects. You can also use `GetCustomAttribute` and specify an Attribute type.

Here's an example of using `GetCustomAttributes` on a `MemberInfo` instance for `MyClass` (which we saw earlier has an `[Obsolete]` attribute on it).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

That will print to console: `Attribute on MyClass: ObsoleteAttribute`. Try adding other attributes to `MyClass`.

It's important to note that these `Attribute` objects are instantiated lazily. That is, they won't be instantiated until you use `GetCustomAttribute` or `GetCustomAttributes`. They are also instantiated each time. Calling `GetCustomAttributes` twice in a row will return two different instances of `ObsoleteAttribute`.

Common attributes in the base class library (BCL)

Attributes are used by many tools and frameworks. NUnit uses attributes like `[Test]` and `[TestFixture]` that are used by the NUnit test runner. ASP.NET MVC uses attributes like `[Authorize]` and provides an action filter framework to perform cross-cutting concerns on MVC actions. PostSharp uses the attribute syntax to allow aspect-oriented programming in C#.

Here are a few notable attributes built into the .NET Core base class libraries:

- `[Obsolete]`. This one was used in the above examples, and it lives in the `System` namespace. It is useful to provide declarative documentation about a changing code base. A message can be provided in the form of a string, and another boolean parameter can be used to escalate from a compiler warning to a compiler error.
- `[Conditional]`. This attribute is in the `System.Diagnostics` namespace. This attribute can be applied to

methods (or attribute classes). You must pass a string to the constructor. If that string doesn't match a `#define` directive, then any calls to that method (but not the method itself) will be removed by the C# compiler. Typically this is used for debugging (diagnostics) purposes.

- `[CallerMemberName]`. This attribute can be used on parameters, and lives in the `System.Runtime.CompilerServices` namespace. This is an attribute that is used to inject the name of the method that is calling another method. This is typically used as a way to eliminate 'magic strings' when implementing INotifyPropertyChanged in various UI frameworks. As an example:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

In the above code, you don't have to have a literal `"Name"` string. This can help prevent typo-related bugs and also makes for smoother refactoring/renaming.

Summary

Attributes bring declarative power to C#, but they are a meta-data form of code and don't act by themselves.

A Tour of the C# Language

4/11/2019 • 4 minutes to read • Edit Online

C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language. If you want to explore the language through interactive examples, try our introduction to C# tutorials.

C# is an object-oriented language, but C# further includes support for ***component-oriented*** programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to support directly these concepts, making C# a very natural language in which to create and use software components.

Several C# features aid in the construction of robust and durable applications: ***Garbage collection*** automatically reclaims memory occupied by unreachable unused objects; ***exception handling*** provides a structured and extensible approach to error detection and recovery; and the ***type-safe*** design of the language makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# has a ***unified type system***. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. Thus, all types share a set of common operations, and values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on ***versioning*** in C#'s design. Many programming languages pay little attention to this issue, and, as a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program might be compiled using the command line:

```
csc hello.cs
```

which produces an executable assembly named hello.exe. The output produced by this application when it is run is:

```
Hello, World
```

> **IMPORTANT**
> The `csc` command compiles for the full framework, and may not be available on all platforms.

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the static modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the standard class libraries, which, by default, are automatically referenced by the compiler.

There's a lot more to learn about C#. The following topics provide an overview of the elements of the C# language. These overviews will provide basic information about all elements of the language and give you the information necessary to dive deeper into elements of the C# language:

- Program Structure
  - Learn the key organizational concepts in the C# language: ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***.
- Types and Variables
  - Learn about ***value types***, ***reference types***, and ***variables*** in the C# language.
- Expressions
  - ***Expressions*** are constructed from ***operands*** and ***operators***. Expressions produce a value.
- Statements
  - You use ***statements*** to express the actions of a program.
- Classes and objects
  - ***Classes*** are the most fundamental of C#'s types. ***Objects*** are instances of a class. Classes are built using ***members***, which are also covered in this topic.
- Structs
  - ***Structs*** are data structures that, unlike classes, are value types.
- Arrays
  - An ***array*** is a data structure that contains a number of variables that are accessed through computed indices.
- Interfaces
  - An ***interface*** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the

members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

- Enums
  - An ***enum type*** is a distinct value type with a set of named constants.
- Delegates
  - A ***delegate type*** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.
- Attributes
  - ***Attributes*** enable programs to specify additional declarative information about types, members, and other entities.

NEXT

What's new in C# 8.0

7/20/2019 • 13 minutes to read • Edit Online

There are many enhancements to the C# language that you can try out already.

- Readonly members
- Default interface members
- Pattern matching enhancements:
  - Switch expressions
  - Property patterns
  - Tuple patterns
  - Positional patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams
- Indices and ranges

> **NOTE**
> This article was last updated for C# 8.0 preview 5.

The remainder of this article briefly describes these features. Where in-depth articles are available, links to those tutorials and overviews are provided. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the dotnet-try global tool.
2. Clone the dotnet/try-samples repository.
3. Set the current directory to the *csharp8* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Readonly members

You can apply the `readonly` modifier to any member of a struct. It indicates that the member does not modify state. It's more granular than applying the `readonly` modifier to a `struct` declaration. Consider the following mutable struct:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Like most structs, the `ToString()` method does not modify state. You could indicate that by adding the `readonly`

modifier to the declaration of `ToString()`:

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

The preceding change generates a compiler warning, because `ToString` accesses the `Distance` property, which is not marked `readonly`:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

The compiler warns you when it needs to create a defensive copy. The `Distance` property does not change state, so you can fix this warning by adding the `readonly` modifier to the declaration:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notice that the `readonly` modifier is necessary on a read only property. The compiler doesn't assume `get` accessors do not modify state; you must declare `readonly` explicitly. The compiler does enforce the rule that `readonly` members do not modify state. The following method will not compile unless you remove the `readonly` modifier:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

This feature lets you specify your design intent so the compiler can enforce it, and make optimizations based on that intent.

Default interface members

You can now add members to interfaces and provide an implementation for those members. This language feature enables API authors to add methods to an interface in later versions without breaking source or binary compatibility with existing implementations of that interface. Existing implementations *inherit* the default implementation. This feature also enables C# to interoperate with APIs that target Android or Swift, which support similar features. Default interface members also enable scenarios similar to a "traits" language feature.

Default interface members affects many scenarios and language elements. Our first tutorial covers updating an interface with default implementations. Other tutorials and reference updates are coming in time for general release.

More patterns in more places

Pattern matching gives tools to provide shape-dependent functionality across related but different kinds of data. C# 7.0 introduced syntax for type patterns and constant patterns by using the `is` expression and the `switch` statement. These features represented the first tentative steps toward supporting programming paradigms where data and functionality live apart. As the industry moves toward more microservices and other cloud-based architectures, other language tools are needed.

C# 8.0 expands this vocabulary so you can use more pattern expressions in more places in your code. Consider these features when your data and functionality are separate. Consider pattern matching when your algorithms depend on a fact other than the runtime type of an object. These techniques provide another way to express

designs.

In addition to new patterns in new places, C# 8.0 adds **recursive patterns**. The result of any pattern expression is an expression. A recursive pattern is simply a pattern expression applied to the output of another pattern expression.

switch expressions

Often, a `switch` statement produces a value in each of its `case` blocks. **Switch expressions** enable you to use more concise expression syntax. There are fewer repetitive `case` and `break` keywords, and fewer curly braces. As an example, consider the following enum that lists the colors of the rainbow:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

If your application defined an `RGBColor` type that is constructed from the `R`, `G` and `B` components, you could convert a `Rainbow` value to its RGB values using the following method containing a switch expression:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

There are several syntax improvements here:

- The variable comes before the `switch` keyword. The different order makes it visually easy to distinguish the switch expression from the switch statement.
- The `case` and `:` elements are replaced with `=>`. It's more concise and intuitive.
- The `default` case is replaced with a `_` discard.
- The bodies are expressions, not statements.

Contrast that with the equivalent code using the classic `switch` statement:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Property patterns

The **property pattern** enables you to match on properties of the object examined. Consider an eCommerce site that must compute sales tax based on the buyer's address. That computation is not a core responsibility of an `Address` class. It will change over time, likely more often than address format changes. The amount of sales tax depends on the `State` property of the address. The following method uses the property pattern to compute the sales tax from the address and the price:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.75M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

Pattern matching creates a concise syntax for expressing this algorithm.

Tuple patterns

Some algorithms depend on multiple inputs. **Tuple patterns** allow you to switch based on multiple values expressed as a tuple. The following code shows a switch expression for the game *rock, paper, scissors*:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

The messages indicate the winner. The discard case represents the three combinations for ties, or other text inputs.

Positional patterns

Some types include a `Deconstruct` method that deconstructs its properties into discrete variables. When a `Deconstruct` method is accessible, you can use **positional patterns** to inspect properties of the object and use those properties for a pattern. Consider the following `Point` class that includes a `Deconstruct` method to create discrete variables for `X` and `Y`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

Additionally, consider the following enum that represents various positions of a quadrant:

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

The following method uses the **positional pattern** to extract the values of `x` and `y`. Then, it uses a `when` clause to determine the `Quadrant` of the point:

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

The discard pattern in the preceding switch matches when either `x` or `y` is 0, but not both. A switch expression must either produce a value or throw an exception. If none of the cases match, the switch expression throws an exception. The compiler generates a warning for you if you do not cover all possible cases in your switch expression.

You can explore pattern matching techniques in this advanced tutorial on pattern matching.

using declarations

A **using declaration** is a variable declaration preceded by the `using` keyword. It tells the compiler that the variable being declared should be disposed at the end of the enclosing scope. For example, consider the following code that writes a text file:

```
static void WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
    }
// file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace for the method is reached. That's the end of the scope in which `file` is declared. The preceding code is equivalent to the following code that uses the classic using statement:

```
static void WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
        }
    } // file is disposed here
}
```

In the preceding example, the file is disposed when the closing brace associated with the `using` statement is reached.

In both cases, the compiler generates the call to `Dispose()`. The compiler generates an error if the expression in the `using` statement is not disposable.

Static local functions

You can now add the `static` modifier to local functions to ensure that local function doesn't capture (reference) any variables from the enclosing scope. Doing so generates `CS8421`, "A static local function can't contain a reference to <variable>."

Consider the following code. The local function `LocalFunction` accesses the variable `y`, declared in the enclosing scope (the method `M`). Therefore, `LocalFunction` can't be declared with the `static` modifier:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The following code contains a static local function. It can be static because it doesn't access any variables in the enclosing scope:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Disposable ref structs

A `struct` declared with the `ref` modifier may not implement any interfaces and so cannot implement IDisposable. Therefore, to enable a `ref struct` to be disposed, it must have an accessible `void Dispose()` method. This also applies to `readonly ref struct` declarations.

Nullable reference types

Inside a nullable annotation context, any variable of a reference type is considered to be a **nonnullable reference type**. If you want to indicate that a variable may be null, you must append the type name with the `?` to declare the variable as a **nullable reference type**.

For nonnullable reference types, the compiler uses flow analysis to ensure that local variables are initialized to a non-null value when declared. Fields must be initialized during construction. The compiler generates a warning if the variable is not set by a call to any of the available constructors or by an initializer. Furthermore, nonnullable reference types can't be assigned a value that could be null.

Nullable reference types aren't checked to ensure they aren't assigned or initialized to null. However, the compiler uses flow analysis to ensure that any variable of a nullable reference type is checked against null before it's accessed or assigned to a nonnullable reference type.

You can learn more about the feature in the overview of nullable reference types. Try it yourself in a new application in this nullable reference types tutorial. Learn about the steps to migrate an existing codebase to make use of nullable reference types in the migrating an application to use nullable reference types tutorial.

Asynchronous streams

Starting with C# 8.0, you can create and consume streams asynchronously. A method that returns an asynchronous stream has three properties:

1. It's declared with the `async` modifier.
2. It returns an IAsyncEnumerable<T>.
3. The method contains `yield return` statements to return successive elements in the asynchronous stream.

Consuming an asynchronous stream requires you to add the `await` keyword before the `foreach` keyword when you enumerate the elements of the stream. Adding the `await` keyword requires the method that enumerates the asynchronous stream to be declared with the `async` modifier and to return a type allowed for an `async` method. Typically that means returning a Task or Task<TResult>. It can also be a ValueTask or ValueTask<TResult>. A method can both consume and produce an asynchronous stream, which means it would return an IAsyncEnumerable<T>. The following code generates a sequence from 0 to 19, waiting 100 ms between generating each number:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

You would enumerate the sequence using the `await foreach` statement:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

You can try asynchronous streams yourself in our tutorial on creating and consuming async streams.

Indices and ranges

Ranges and indices provide a succinct syntax for specifying subranges in an array, Span<T>, or ReadOnlySpan<T>.

This language support relies on two new types, and two new operators.

- System.Index represents an index into a sequence.
- The `^` operator, which specifies that an index is relative to the end of the sequence.
- System.Range represents a sub range of a sequence.
- The Range operator (`..`), which specifies the start and end of a range as its operands.

Let's start with the rules for indexes. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. Note that `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

A range specifies the *start* and *end* of a range. The start of the range is inclusive, but the end of the range is exclusive, meaning the *start* is included in the range but the *end* is not included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

Let's look at a few examples. Consider the following array, annotated with its index from the start and from the end:

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index:

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` is not in the range.

```
var quickBrownFox = words[1..4];
```

The following code creates a subrange with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` is not included:

```
var lazyDog = words[^2..^0];
```

The following examples create ranges that are open ended for the start, end, or both:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

You can also declare ranges as variables:

```
Range phrase = 1..4;
```

The range can then be used inside the `[` and `]` characters:

```
var text = words[phrase];
```

You can explore more about indices and ranges in the tutorial on indices and ranges.

What's new in C# 7.3

7/24/2019 • 6 minutes to read • Edit Online

There are two main themes to the C# 7.3 release. One theme provides features that enable safe code to be as performant as unsafe code. The second theme provides incremental improvements to existing features. In addition, new compiler options were added in this release.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use additional generic constraints.

The following enhancements were made to existing features:

- You can test `==` and `!=` with tuple types.
- You can use expression variables in more locations.
- You may attach attributes to the backing field of auto-implemented properties.
- Method resolution when arguments differ by `in` has been improved.
- Overload resolution now has fewer ambiguous cases.

The new compiler options are:

- `-publicsign` to enable Open Source Software (OSS) signing of assemblies.
- `-pathmap` to provide a mapping for source directories.

The remainder of this article provides details and links to learn more about each of the improvements. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the dotnet-try global tool.
2. Clone the dotnet/try-samples repository.
3. Set the current directory to the *csharp7* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Enabling more efficient safe code

You should be able to write C# code safely that performs as well as unsafe code. Safe code avoids classes of errors, such as buffer overruns, stray pointers, and other memory access errors. These new features expand the capabilities of verifiable safe code. Strive to write more of your code using safe constructs. These features make that easier.

Indexing `fixed` fields does not require pinning

Consider this struct:

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

In earlier versions of C#, you needed to pin a variable to access one of the integers that are part of `myFixedField`. Now, the following code compiles without pinning the variable `p` inside a separate `fixed` statement:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

The variable `p` accesses one element in `myFixedField`. You don't need to declare a separate `int*` variable. Note that you still need an `unsafe` context. In earlier versions of C#, you need to declare a second fixed pointer:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

For more information, see the article on the `fixed` statement.

`ref` local variables may be reassigned

Now, `ref` locals may be reassigned to refer to different instances after being initialized. The following code now compiles:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

For more information, see the article on `ref` returns and `ref` locals, and the article on `foreach`.

`stackalloc` arrays support initializers

You've been able to specify the values for elements in an array when you initialize it:

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Now, that same syntax can be applied to arrays that are declared with `stackalloc`:

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

For more information, see the `stackalloc` operator article.

More types support the `fixed` statement

The `fixed` statement supported a limited set of types. Starting with C# 7.3, any type that contains a `GetPinnableReference()` method that returns a `ref T` or `ref readonly T` may be `fixed`. Adding this feature means that `fixed` can be used with System.Span<T> and related types.

For more information, see the `fixed` statement article in the language reference.

Enhanced generic constraints

You can now specify the type System.Enum or System.Delegate as base class constraints for a type parameter.

You can also use the new `unmanaged` constraint, to specify that a type parameter must be an unmanaged type.

For more information, see the articles on `where` generic constraints and constraints on type parameters.

Adding these constraints to existing types is an incompatible change. Closed generic types may no longer meet these new constraints.

Make existing features better

The second theme provides improvements to features in the language. These features improve productivity when writing C#.

Tuples support `==` and `!=`

The C# tuple types now support `==` and `!=`. For more information, see the section covering equality in the article on tuples.

Attach attributes to the backing fields for auto-implemented properties

This syntax is now supported:

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

The attribute `SomeThingAboutFieldAttribute` is applied to the compiler generated backing field for `SomeProperty`. For more information, see attributes in the C# programming guide.

`in` method overload resolution tiebreaker

When the `in` argument modifier was added, these two methods would cause an ambiguity:

```
static void M(S arg);
static void M(in S arg);
```

Now, the by value (first in the preceding example) overload is better than the by readonly reference version. To call the version with the readonly reference argument, you must include the `in` modifier when calling the method.

> **NOTE**
> This was implemented as a bug fix. This no longer is ambiguous even with the language version set to "7.2".

For more information, see the article on the `in` parameter modifier.

Extend expression variables in initializers

The syntax added in C# 7.0 to allow `out` variable declarations has been extended to include field initializers, property initializers, constructor initializers, and query clauses. It enables code such as the following example:

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

Improved overload candidates

In every release, the overload resolution rules get updated to address situations where ambiguous method invocations have an "obvious" choice. This release adds three new rules to help the compiler pick the obvious choice:

1. When a method group contains both instance and static members, the compiler discards the instance members if the method was invoked without an instance receiver or context. The compiler discards the static members if the method was invoked with an instance receiver. When there is no receiver, the compiler includes only static members in a static context, otherwise both static and instance members. When the receiver is ambiguously an instance or type, the compiler includes both. A static context, where an implicit `this` instance receiver cannot be used, includes the body of members where no `this` is defined, such as static members, as well as places where `this` cannot be used, such as field initializers and constructor-initializers.
2. When a method group contains some generic methods whose type arguments do not satisfy their constraints, these members are removed from the candidate set.
3. For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

You'll only notice this change because you'll find fewer compiler errors for ambiguous method overloads when you are sure which method is better.

New compiler options

New compiler options support new build and DevOps scenarios for C# programs.

Public or Open Source signing

The `-publicsign` compiler option instructs the compiler to sign the assembly using a public key. The assembly is marked as signed, but the signature is taken from the public key. This option enables you to build signed assemblies from open-source projects using a public key.

For more information, see the -publicsign compiler option article.

pathmap

The `-pathmap` compiler option instructs the compiler to replace source paths from the build environment with mapped source paths. The `-pathmap` option controls the source path written by the compiler to PDB files or for the CallerFilePathAttribute.

For more information, see the -pathmap compiler option article.

What's new in C# 7.2

6/25/2019 • 3 minutes to read • Edit Online

C# 7.2 is another point release that adds a number of useful features. One theme for this release is working more efficiently with value types by avoiding unnecessary copies or allocations.

The remaining features are small, nice-to-have features.

C# 7.2 uses the language version selection configuration element to select the compiler language version.

The new language features in this release are:

- Techniques for writing safe efficient code
  - A combination of syntax improvements that enable working with value types using reference semantics.
- Non-trailing named arguments
  - Named arguments can be followed by positional arguments.
- Leading underscores in numeric literals
  - Numeric literals can now have leading underscores before any printed digits.
- `private protected` access modifier
  - The `private protected` access modifier enables access for derived classes in the same assembly.
- Conditional `ref` expressions
  - The result of a conditional expression (`?:`) can now be a reference.

The remainder of this article provides an overview of each feature. For each feature, you'll learn the reasoning behind it. You'll learn the syntax. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the dotnet-try global tool.
2. Clone the dotnet/try-samples repository.
3. Set the current directory to the *csharp7* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Safe efficient code enhancements

Language features introduced in 7.2 let you work with value types while using reference semantics. They are designed to increase performance by minimizing copying value types without incurring the memory allocations associated with using reference types. The features include:

- The `in` modifier on parameters, to specify that an argument is passed by reference but not modified by the called method. Adding the `in` modifier to an argument is a source compatible change.
- The `ref readonly` modifier on method returns, to indicate that a method returns its value by reference but doesn't allow writes to that object. Adding the `ref readonly` modifier is a source compatible change, if the return is assigned to a value. Adding the `readonly` modifier to an existing `ref` return statement is an incompatible change. It requires callers to update the declaration of `ref` local variables to include the `readonly` modifier.
- The `readonly struct` declaration, to indicate that a struct is immutable and should be passed as an `in` parameter to its member methods. Adding the `readonly` modifier to an existing struct declaration is a binary compatible change.
- The `ref struct` declaration, to indicate that a struct type accesses managed memory directly and must always

be stack allocated. Adding the `ref` modifier to an existing `struct` declaration is an incompatible change. A `ref struct` cannot be a member of a class or used in other locations where it may be allocated on the heap.

You can read more about all these changes in Write safe efficient code.

Non-trailing named arguments

Method calls may now use named arguments that precede positional arguments when those named arguments are in the correct positions. For more information see Named and optional arguments.

Leading underscores in numeric literals

The implementation of support for digit separators in C# 7.0 didn't allow the `_` to be the first character of the literal value. Hex and binary numeric literals may now begin with an `_`.

For example:

```
int binaryValue = 0b_0101_0101;
```

private protected access modifier

A new compound access modifier: `private protected` indicates that a member may be accessed by containing class or derived classes that are declared in the same assembly. While `protected internal` allows access by derived classes or classes that are in the same assembly, `private protected` limits access to derived types declared in the same assembly.

For more information see access modifiers in the language reference.

Conditional `ref` expressions

Finally, the conditional expression may produce a ref result instead of a value result. For example, you would write the following to retrieve a reference to the first element in one of two arrays:

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

The variable `r` is a reference to the first value in either `arr` or `otherArr`.

For more information, see conditional operator (?:) in the language reference.

What's new in C# 7.1

8/5/2019 • 3 minutes to read • Edit Online

C# 7.1 is the first point release to the C# language. It marks an increased release cadence for the language. You can use the new features sooner, ideally when each new feature is ready. C# 7.1 adds the ability to configure the compiler to match a specified version of the language. That enables you to separate the decision to upgrade tools from the decision to upgrade language versions.

C# 7.1 adds the language version selection configuration element, three new language features, and new compiler behavior.

The new language features in this release are:

- `async` `Main` method
  - The entry point for an application can have the `async` modifier.
- `default` literal expressions
  - You can use default literal expressions in default value expressions when the target type can be inferred.
- Inferred tuple element names
  - The names of tuple elements can be inferred from tuple initialization in many cases.
- Pattern matching on generic type parameters
  - You can use pattern match expressions on variables whose type is a generic type parameter.

Finally, the compiler has two options `-refout` and `-refonly` that control reference assembly generation.

To use the latest features in a point release, you need to configure the compiler language version and select the version.

The remainder of this article provides an overview of each feature. For each feature, you'll learn the reasoning behind it. You'll learn the syntax. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the dotnet-try global tool.
2. Clone the dotnet/try-samples repository.
3. Set the current directory to the *csharp7* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Async main

An *async main* method enables you to use `await` in your `Main` method. Previously you would need to write:

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

You can now write:

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

If your program doesn't return an exit code, you can declare a `Main` method that returns a Task:

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

You can read more about the details in the async main article in the programming guide.

Default literal expressions

Default literal expressions are an enhancement to default value expressions. These expressions initialize a variable to the default value. Where you previously would write:

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

You can now omit the type on the right-hand side of the initialization:

```
Func<string, bool> whereClause = default;
```

For more information, see the default literal section of the default operator article.

Inferred tuple element names

This feature is a small enhancement to the tuples feature introduced in C# 7.0. Many times when you initialize a tuple, the variables used for the right side of the assignment are the same as the names you'd like for the tuple elements:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count: count, label: label);
```

The names of tuple elements can be inferred from the variables used to initialize the tuple in C# 7.1:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

You can learn more about this feature in the Tuples article.

Pattern matching on generic type parameters

Beginning with C# 7.1, the pattern expression for `is` and the `switch` type pattern may have the type of a generic type parameter. This can be most useful when checking types that may be either `struct` or `class` types, and you want to avoid boxing.

Reference assembly generation

There are two new compiler options that generate *reference-only assemblies*: -refout and -refonly. The linked articles explain these options and reference assemblies in more detail.

What's new in C# 7.0

6/25/2019 • 14 minutes to read • Edit Online

C# 7.0 adds a number of new features to the C# language:

- `out` variables
  - You can declare `out` values inline as arguments to the method where they're used.
- Tuples
  - You can create lightweight, unnamed types that contain multiple public fields. Compilers and IDE tools understand the semantics of these types.
- Discards
  - Discards are temporary, write-only variables used in assignments when you don't care about the value assigned. They're most useful when deconstructing tuples and user-defined types, as well as when calling methods with `out` parameters.
- Pattern Matching
  - You can create branching logic based on arbitrary types and values of the members of those types.
- `ref` locals and returns
  - Method local variables and return values can be references to other storage.
- Local Functions
  - You can nest functions inside other functions to limit their scope and visibility.
- More expression-bodied members
  - The list of members that can be authored using expressions has grown.
- `throw` Expressions
  - You can throw exceptions in code constructs that previously weren't allowed because `throw` was a statement.
- Generalized async return types
  - Methods declared with the `async` modifier can return other types in addition to `Task` and `Task<T>`.
- Numeric literal syntax improvements
  - New tokens improve readability for numeric constants.

The remainder of this article provides an overview of each feature. For each feature, you'll learn the reasoning behind it. You'll learn the syntax. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the dotnet-try global tool.
2. Clone the dotnet/try-samples repository.
3. Set the current directory to the *csharp7* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

`out` variables

The existing syntax that supports `out` parameters has been improved in this version. You can now declare `out` variables in the argument list of a method call, rather than writing a separate declaration statement:

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

You may want to specify the type of the `out` variable for clarity, as shown above. However, the language does support using an implicitly typed local variable:

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- The code is easier to read.
  - You declare the out variable where you use it, not on another line above.
- No need to assign an initial value.
  - By declaring the `out` variable where it's used in a method call, you can't accidentally use it before it is assigned.

Tuples

C# provides a rich syntax for classes and structs that is used to explain your design intent. But sometimes that rich syntax requires extra work with minimal benefit. You may often write methods that need a simple structure containing more than one data element. To support these scenarios *tuples* were added to C#. Tuples are lightweight data structures that contain multiple fields to represent the data members. The fields aren't validated, and you can't define your own methods

> **NOTE**
> Tuples were available before C# 7.0, but they were inefficient and had no language support. This meant that tuple elements could only be referenced as `Item1`, `Item2` and so on. C# 7.0 introduces language support for tuples, which enables semantic names for the fields of a tuple using new, more efficient tuple types.

You can create a tuple by assigning a value to each member, and optionally providing semantic names to each of the members of the tuple:

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

The `namedLetters` tuple contains fields referred to as `Alpha` and `Beta`. Those names exist only at compile time and aren't preserved, for example when inspecting the tuple using reflection at runtime.

In a tuple assignment, you can also specify the names of the fields on the right-hand side of the assignment:

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

There may be times when you want to unpackage the members of a tuple that were returned from a method. You can do that by declaring separate variables for each of the values in the tuple. This unpackaging is called *deconstructing* the tuple:

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

You can also provide a similar deconstruction for any type in .NET. You write a `Deconstruct` method as a member of the class. That `Deconstruct` method provides a set of `out` arguments for each of the properties you want to extract. Consider this `Point` class that provides a deconstructor method that extracts the `X` and `Y` coordinates:

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

You can extract the individual fields by assigning a `Point` to a tuple:

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

You can learn more in depth about tuples in the tuples article.

Discards

Often when deconstructing a tuple or calling a method with `out` parameters, you're forced to define a variable whose value you don't care about and don't intend to use. C# adds support for *discards* to handle this scenario. A discard is a write-only variable whose name is `_` (the underscore character); you can assign all of the values that you intend to discard to the single variable. A discard is like an unassigned variable; apart from the assignment statement, the discard can't be used in code.

Discards are supported in the following scenarios:

- When deconstructing tuples or user-defined types.
- When calling methods with out parameters.
- In a pattern matching operation with the is and switch statements.
- As a standalone identifier when you want to explicitly identify the value of an assignment as a discard.

The following example defines a `QueryCityDataForYears` method that returns a 6-tuple that contains data for a city for two different years. The method call in the example is concerned only with the two population values returned by the method and so treats the remaining values in the tuple as discards when it deconstructs the tuple.

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
       var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

       Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
   }

   private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int year2)
   {
      int population1 = 0, population2 = 0;
      double area = 0;

      if (name == "New York City") {
         area = 468.48;
         if (year1 == 1960) {
            population1 = 7781984;
         }
         if (year2 == 2010) {
            population2 = 8175133;
         }
      return (name, area, year1, population1, year2, population2);
      }

      return ("", 0, 0, 0, 0, 0);
   }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

For more information, see Discards.

Pattern matching

Pattern matching is a feature that allows you to implement method dispatch on properties other than the type of an object. You're probably already familiar with method dispatch based on the type of an object. In object-oriented programming, virtual and override methods provide language syntax to implement method dispatching based on an object's type. Base and Derived classes provide different implementations. Pattern matching expressions extend this concept so that you can easily implement similar dispatch patterns for types and data elements that aren't related through an inheritance hierarchy.

Pattern matching supports `is` expressions and `switch` expressions. Each enables inspecting an object and its properties to determine if that object satisfies the sought pattern. You use the `when` keyword to specify additional rules to the pattern.

The `is` pattern expression extends the familiar `is` operator to query an object about its type and assign the result in one instruction. The following code checks if a variable is an `int`, and if so, adds it to the current sum:

```
if (input is int count)
    sum += count;
```

The preceding small example demonstrates the enhancements to the `is` expression. You can test against value types as well as reference types, and you can assign the successful result to a new variable of the correct type.

The switch match expression has a familiar syntax, based on the `switch` statement already part of the C# language. The updated switch statement has several new constructs:

- The governing type of a `switch` expression is no longer restricted to integral types, `Enum` types, `string`, or a nullable type corresponding to one of those types. Any type may be used.
- You can test the type of the `switch` expression in each `case` label. As with the `is` expression, you may assign a new variable to that type.
- You may add a `when` clause to further test conditions on that variable.
- The order of `case` labels is now important. The first branch to match is executed; others are skipped.

The following code demonstrates these new features:

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` is the familiar constant pattern.
- `case IEnumerable<int> childSequence:` is a type pattern.
- `case int n when n > 0:` is a type pattern with an additional `when` condition.
- `case null:` is the null pattern.
- `default:` is the familiar default case.

You can learn more about pattern matching in Pattern Matching in C#.

Ref locals and returns

This feature enables algorithms that use and return references to variables defined elsewhere. One example is working with large matrices, and finding a single location with certain characteristics. The following method returns a **reference** to that storage in the matrix:

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

You can declare the return value as a `ref` and modify that value in the matrix, as shown in the following code:

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

The C# language has several rules that protect you from misusing the `ref` locals and returns:

- You must add the `ref` keyword to the method signature and to all `return` statements in a method.
  - That makes it clear the method returns by reference throughout the method.
- A `ref return` may be assigned to a value variable, or a `ref` variable.
  - The caller controls whether the return value is copied or not. Omitting the `ref` modifier when assigning the return value indicates that the caller wants a copy of the value, not a reference to the storage.
- You can't assign a standard method return value to a `ref` local variable.
  - That disallows statements like `ref int i = sequence.Count();`
- You can't return a `ref` to a variable whose lifetime doesn't extend beyond the execution of the method.
  - That means you can't return a reference to a local variable or a variable with a similar scope.
- `ref` locals and returns can't be used with async methods.
  - The compiler can't know if the referenced variable has been set to its final value when the async method returns.

The addition of ref locals and ref returns enables algorithms that are more efficient by avoiding copying values, or performing dereferencing operations multiple times.

Adding `ref` to the return value is a source compatible change. Existing code compiles, but the ref return value is copied when assigned. Callers must update the storage for the return value to a `ref` local variable to store the return as a reference.

For more information, see the ref keyword article.

Local functions

Many designs for classes include methods that are called from only one location. These additional private methods keep each method small and focused. *Local functions* enable you to declare methods inside the context of another method. Local functions make it easier for readers of the class to see that the local method is only called from the context in which it is declared.

There are two common use cases for local functions: public iterator methods and public async methods. Both types of methods generate code that reports errors later than programmers might expect. In iterator methods, any exceptions are observed only when calling code that enumerates the returned sequence. In async methods, any exceptions are only observed when the returned `Task` is awaited. The following example demonstrates separating parameter validation from the iterator implementation using a local function:

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

The same technique can be employed with `async` methods to ensure that exceptions arising from argument validation are thrown before the asynchronous work begins:

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

NOTE

Some of the designs that are supported by local functions could also be accomplished using *lambda expressions*. Those interested can read more about the differences

More expression-bodied members

C# 6 introduced expression-bodied members for member functions, and read-only properties. C# 7.0 expands the allowed members that can be implemented as expressions. In C# 7.0, you can implement *constructors*, *finalizers*, and `get` and `set` accessors on *properties* and *indexers*. The following code shows examples of each:

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

> **NOTE**
> This example does not need a finalizer, but it is shown to demonstrate the syntax. You should not implement a finalizer in your class unless it is necessary to release unmanaged resources. You should also consider using the SafeHandle class instead of managing unmanaged resources directly.

These new locations for expression-bodied members represent an important milestone for the C# language: These features were implemented by community members working on the open-source Roslyn project.

Changing a method to an expression bodied member is a binary compatible change.

Throw expressions

In C#, `throw` has always been a statement. Because `throw` is a statement, not an expression, there were C# constructs where you couldn't use it. These included conditional expressions, null coalescing expressions, and some lambda expressions. The addition of expression-bodied members adds more locations where `throw` expressions would be useful. So that you can write any of these constructs, C# 7.0 introduces *throw expressions*.

This addition makes it easier to write more expression-based code. You don't need additional statements for error checking.

Generalized async return types

Returning a `Task` object from async methods can introduce performance bottlenecks in certain paths. `Task` is a reference type, so using it means allocating an object. In cases where a method declared with the `async` modifier returns a cached result, or completes synchronously, the extra allocations can become a significant time cost in performance critical sections of code. It can become costly if those allocations occur in tight loops.

The new language feature means that async method return types aren't limited to `Task`, `Task<T>`, and `void`. The returned type must still satisfy the async pattern, meaning a `GetAwaiter` method must be accessible. As one concrete example, the `ValueTask` type has been added to the .NET framework to make use of this new language feature:

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

> **NOTE**
>
> You need to add the NuGet package `System.Threading.Tasks.Extensions` in order to use the ValueTask<TResult> type.

This enhancement is most useful for library authors to avoid allocating a `Task` in performance critical code.

Numeric literal syntax improvements

Misreading numeric constants can make it harder to understand code when reading it for the first time. Bit masks or other symbolic values are prone to misunderstanding. C# 7.0 includes two new features to write numbers in the most readable fashion for the intended use: *binary literals*, and *digit separators*.

For those times when you're creating bit masks, or whenever a binary representation of a number makes the most readable code, write that number in binary:

```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

The `0b` at the beginning of the constant indicates that the number is written as a binary number. Binary numbers can get long, so it's often easier to see the bit patterns by introducing the `_` as a digit separator, as shown above in the binary constant. The digit separator can appear anywhere in the constant. For base 10 numbers, it is common to use it as a thousands separator:

```
public const long BillionsAndBillions = 100_000_000_000;
```

The digit separator can be used with `decimal`, `float`, and `double` types as well:

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

Taken together, you can declare numeric constants with much more readability.

What's New in C# 6

8/13/2019 • 9 minutes to read • Edit Online

The 6.0 release of C# contained many features that improve productivity for developers. The overall effect of these features is that you write more concise code that is also more readable. The syntax contains less ceremony for many common practices. It's easier to see the design intent with less ceremony. Learn these features well, and you'll be more productive and write more readable code. You can concentrate more on your features than on the constructs of the language.

The rest of this article provides an overview of each of these features, with a link to explore each feature. You can also explore the features in an interactive exploration on C# 6 in the tutorials section.

Read-only auto-properties

Read-only auto-properties provide a more concise syntax to create immutable types. You declare the auto-property with only a get accessor:

```
public string FirstName { get; }
public string LastName { get;  }
```

The `FirstName` and `LastName` properties can be set only in the body of the constructor of the same class:

```
public Student(string firstName, string lastName)
{
    if (IsNullOrWhiteSpace(lastName))
        throw new ArgumentException(message: "Cannot be blank", paramName: nameof(lastName));
    FirstName = firstName;
    LastName = lastName;
}
```

Trying to set `LastName` in another method generates a `CS0200` compilation error:

```
public class Student
{
    public string LastName { get;  }

    public void ChangeName(string newLastName)
    {
        // Generates CS0200: Property or indexer cannot be assigned to -- it is read only
        LastName = newLastName;
    }
}
```

This feature enables true language support for creating immutable types and uses the more concise and convenient auto-property syntax.

If adding this syntax doesn't remove an accessible method, it's a binary compatible change.

Auto-property initializers

Auto-property initializers let you declare the initial value for an auto-property as part of the property declaration.

```
public ICollection<double> Grades { get; } = new List<double>();
```

The `Grades` member is initialized where it's declared. That makes it easier to perform the initialization exactly once. The initialization is part of the property declaration, making it easier to equate the storage allocation with the public interface for `Student` objects.

Expression-bodied function members

Many members that you write are single statements that could be single expressions. Write an expression-bodied member instead. It works for methods and read-only properties. For example, an override of `ToString()` is often a great candidate:

```
public override string ToString() => $"{LastName}, {FirstName}";
```

You can also use this syntax for read-only properties:

```
public string FullName => $"{FirstName} {LastName}";
```

Changing an existing member to an expression bodied member is a binary compatible change.

using static

The *using static* enhancement enables you to import the static methods of a single class. You specify the class you're using:

```
using static System.Math;
```

The Math does not contain any instance methods. You can also use `using static` to import a class' static methods for a class that has both static and instance methods. One of the most useful examples is String:

```
using static System.String;
```

> **NOTE**
> You must use the fully qualified class name, `System.String` in a static using statement. You cannot use the `string` keyword instead.

When imported from a `static using` statement, extension methods are only in scope when called using the extension method invocation syntax. They aren't in scope when called as a static method. You'll often see this in LINQ queries. You can import the LINQ pattern by importing Enumerable, or Queryable.

```
using static System.Linq.Enumerable;
```

You typically call extension methods using extension method invocation expressions. Adding the class name in the rare case where you call them using static method call syntax resolves ambiguity.

The `static using` directive also imports any nested types. You can reference any nested types without qualification.

Null-conditional operators

The *null conditional operator* makes null checks much easier and fluid. Replace the member access `.` with `?.`:

```
var first = person?.FirstName;
```

In the preceding example, the variable `first` is assigned `null` if the person object is `null`. Otherwise, it is assigned the value of the `FirstName` property. Most importantly, the `?.` means that this line of code doesn't generate a `NullReferenceException` if the `person` variable is `null`. Instead, it short-circuits and returns `null`. You can also use a null conditional operator for array or indexer access. Replace `[]` with `?[]` in the index expression.

The following expression returns a `string`, regardless of the value of `person`. You often use this construct with the *null coalescing* operator to assign default values when one of the properties is `null`. When the expression short-circuits, the `null` value returned is typed to match the full expression.

```
first = person?.FirstName ?? "Unspecified";
```

You can also use `?.` to conditionally invoke methods. The most common use of member functions with the null conditional operator is to safely invoke delegates (or event handlers) that may be `null`. You'll call the delegate's `Invoke` method using the `?.` operator to access the member. You can see an example in the delegate patterns article.

The rules of the `?.` operator ensure that the left-hand side of the operator is evaluated only once. It enables many idioms, including the following example using event handlers:

```
// preferred in C# 6:
this.SomethingHappened?.Invoke(this, eventArgs);
```

Ensuring that the left side is evaluated only once also enables you to use any expression, including method calls, on the left side of the `?.`

String interpolation

With C# 6, the new string interpolation feature enables you to embed expressions in a string. Simply preface the string with `$` and use expressions between `{` and `}` instead of ordinals:

```
public string FullName => $"{FirstName} {LastName}";
```

This example uses properties for the substituted expressions. You can use any expression. For example, you could compute a student's grade point average as part of the interpolation:

```
public string GetGradePointPercentage() =>
    $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Average():F2}";
```

The preceding line of code formats the value for `Grades.Average()` as a floating-point number with two decimal places.

Often, you may need to format the string produced using a specific culture. You use the fact that the object produced by a string interpolation can be implicitly converted to System.FormattableString. The FormattableString instance contains the composite format string and the results of evaluating the expressions before converting them to strings. Use the FormattableString.ToString(IFormatProvider) method to specify the culture when formatting a

string. The following example produces a string using the German (de-DE) culture. (By default, the German culture uses the ',' character for the decimal separator, and the '.' character as the thousands separator.)

```
FormattableString str = $"Average grade is {s.Grades.Average()}";
var gradeStr = str.ToString(new System.Globalization.CultureInfo("de-DE"));
```

To get started with string interpolation, see the String interpolation in C# interactive tutorial, the String interpolation article, and the String interpolation in C# tutorial.

Exception filters

Exception Filters are clauses that determine when a given catch clause should be applied. If the expression used for an exception filter evaluates to `true`, the catch clause performs its normal processing on an exception. If the expression evaluates to `false`, then the `catch` clause is skipped. One use is to examine information about an exception to determine if a `catch` clause can process the exception:

```
public static async Task<string> MakeRequest()
{
    WebRequestHandler webRequestHandler = new WebRequestHandler();
    webRequestHandler.AllowAutoRedirect = false;
    using (HttpClient client = new HttpClient(webRequestHandler))
    {
        var stringTask = client.GetStringAsync("https://docs.microsoft.com/en-us/dotnet/about/");
        try
        {
            var responseText = await stringTask;
            return responseText;
        }
        catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
    }
}
```

The `nameof` expression

The nameof expression evaluates to the name of a symbol. It's a great way to get tools working whenever you need the name of a variable, a property, or a member field. One of the most common uses for `nameof` is to provide the name of a symbol that caused an exception:

```
if (IsNullOrWhiteSpace(lastName))
    throw new ArgumentException(message: "Cannot be blank", paramName: nameof(lastName));
```

Another use is with XAML-based applications that implement the `INotifyPropertyChanged` interface:

```
public string LastName
{
    get { return lastName; }
    set
    {
        if (value != lastName)
        {
            lastName = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(LastName)));
        }
    }
}
private string lastName;
```

Await in Catch and Finally blocks

C# 5 had several limitations around where you could place `await` expressions. With C# 6, you can now use `await` in `catch` or `finally` expressions. This is most often used with logging scenarios:

```
public static async Task<string> MakeRequestAndLogFailures()
{
    await logMethodEntrance();
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:10000");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))
    {
        await logError("Recovered from redirect", e);
        return "Site Moved";
    }
    finally
    {
        await logMethodExit();
        client.Dispose();
    }
}
```

The implementation details for adding `await` support inside `catch` and `finally` clauses ensure that the behavior is consistent with the behavior for synchronous code. When code executed in a `catch` or `finally` clause throws, execution looks for a suitable `catch` clause in the next surrounding block. If there was a current exception, that exception is lost. The same happens with awaited expressions in `catch` and `finally` clauses: a suitable `catch` is searched for, and the current exception, if any, is lost.

> **NOTE**
> This behavior is the reason it's recommended to write `catch` and `finally` clauses carefully, to avoid introducing new exceptions.

Initialize associative collections using indexers

Index Initializers is one of two features that make collection initializers more consistent with index usage. In earlier releases of C#, you could use *collection initializers* with sequence style collections, including Dictionary<TKey,TValue>, by adding braces around key and value pairs:

```
private Dictionary<int, string> messages = new Dictionary<int, string>
{
    { 404, "Page not Found"},
    { 302, "Page moved, but left a forwarding address."},
    { 500, "The web server can't come out to play today."}
};
```

You can use them with Dictionary<TKey,TValue> collections and other types where the accessible `Add` method accepts more than one argument. The new syntax supports assignment using an index into the collection:

```
private Dictionary<int, string> webErrors = new Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

This feature means that associative containers can be initialized using syntax similar to what's been in place for sequence containers for several versions.

Extension `Add` methods in collection initializers

Another feature that makes collection initialization easier is the ability to use an *extension method* for the `Add` method. This feature was added for parity with Visual Basic. The feature is most useful when you have a custom collection class that has a method with a different name to semantically add new items.

Improved overload resolution

This last feature is one you probably won't notice. There were constructs where the previous version of the C# compiler may have found some method calls involving lambda expressions ambiguous. Consider this method:

```
static Task DoThings()
{
     return Task.FromResult(0);
}
```

In earlier versions of C#, calling that method using the method group syntax would fail:

```
Task.Run(DoThings);
```

The earlier compiler couldn't distinguish correctly between `Task.Run(Action)` and `Task.Run(Func<Task>())`. In previous versions, you'd need to use a lambda expression as an argument:

```
Task.Run(() => DoThings());
```

The C# 6 compiler correctly determines that `Task.Run(Func<Task>())` is a better choice.

Deterministic compiler output

The `-deterministic` option instructs the compiler to produce a byte-for-byte identical output assembly for successive compilations of the same source files.

By default, every compilation produces unique output on each compilation. The compiler adds a timestamp, and a GUID generated from random numbers. You use this option if you want to compare the byte-for-byte output to

ensure consistency across builds.

For more information, see the -deterministic compiler option article.

The history of C#

7/20/2019 • 8 minutes to read • Edit Online

This article provides a history of each major release of the C# language. The C# team is continuing to innovate and add new features. Detailed language feature status, including features considered for upcoming releases can be found on the dotnet/roslyn repository on GitHub.

> **IMPORTANT**
>
> The C# language relies on types and methods in what the C# specification defines as a *standard library* for some of the features. The .NET platform delivers those types and methods in a number of packages. One example is exception processing. Every `throw` statement or expression is checked to ensure the object being thrown is derived from Exception. Similarly, every `catch` is checked to ensure that the type being caught is derived from Exception. Each version may add new requirements. To use the latest language features in older environments, you may need to install specific libraries. These dependencies are documented in the page for each specific version. You can learn more about the relationships between language and library for background on this dependency.

The C# build tools consider the latest major language release the default language version. There may be point releases between major releases, detailed in other articles in this section. To use the latest features in a point release, you need to configure the compiler language version and select the version. There have been three point releases since C# 7.0:

- C# 7.3:
  - C# 7.3 is available starting with Visual Studio 2017 version 15.7 and .NET Core 2.1 SDK.
- C# 7.2:
  - C# 7.2 is available starting with Visual Studio 2017 version 15.5, and .NET Core 2.0 SDK.
- C# 7.1:
  - C# 7.1 is available starting with Visual Studio 2017 version 15.3 and .NET Core 2.0 SDK.

C# version 1.0

When you go back and look, C# version 1.0 looked a lot like Java. As part of its stated design goals for ECMA, it sought to be a "simple, modern, general-purpose object-oriented language." At the time, looking like Java meant it achieved those early design goals.

But if you look back on C# 1.0 now, you'd find yourself a little dizzy. It lacked the built-in async capabilities and some of the slick functionality around generics you take for granted. As a matter of fact, it lacked generics altogether. And LINQ? Not available yet. Those additions would take some years to come out.

C# version 1.0 looked stripped of features, compared to today. You'd find yourself writing some verbose code. But yet, you have to start somewhere. C# version 1.0 was a viable alternative to Java on the Windows platform.

The major features of C# 1.0 included:

- Classes
- Structs
- Interfaces
- Events
- Properties
- Delegates

- Expressions
- Statements
- Attributes

C# version 1.2

C# version 1.2 shipped with Visual Studio 2003. It contained a few small enhancements to the language. Most notable is that starting with this version, the code generated in a `foreach` loop called Dispose on an IEnumerator when that IEnumerator implemented IDisposable.

C# version 2.0

Now things start to get interesting. Let's take a look at some major features of C# 2.0, released in 2005, along with Visual Studio 2005:

- Generics
- Partial types
- Anonymous methods
- Nullable types
- Iterators
- Covariance and contravariance

Other C# 2.0 features added capabilities to existing features:

- Getter/setter separate accessibility
- Method group conversions (delegates)
- Static classes
- Delegate inference

While C# may have started as a generic Object-Oriented (OO) language, C# version 2.0 changed that in a hurry. Once they had their feet under them, they went after some serious developer pain points. And they went after them in a significant way.

With generics, types and methods can operate on an arbitrary type while still retaining type safety. For instance, having a List<T> lets you have `List<string>` or `List<int>` and perform type-safe operations on those strings or integers while you iterate through them. Using generics is better than create `ListInt` that derives from `ArrayList` or casting from `Object` for every operation.

C# version 2.0 brought iterators. To put it succinctly, iterators let you examine all the items in a `List` (or other Enumerable types) with a `foreach` loop. Having iterators as a first-class part of the language dramatically enhanced readability of the language and people's ability to reason about the code.

And yet, C# continued to play a bit of catch-up with Java. Java had already released versions that included generics and iterators. But that would soon change as the languages continued to evolve apart.

C# version 3.0

C# version 3.0 came in late 2007, along with Visual Studio 2008, though the full boat of language features would actually come with .NET Framework version 3.5. This version marked a major change in the growth of C#. It established C# as a truly formidable programming language. Let's take a look at some major features in this version:

- Auto-implemented properties
- Anonymous types

- Query expressions
- Lambda expressions
- Expression trees
- Extension methods
- Implicitly typed local variables
- Partial methods
- Object and collection initializers

In retrospect, many of these features seem both inevitable and inseparable. They all fit together strategically. It's generally thought that C# version's killer feature was the query expression, also known as Language-Integrated Query (LINQ).

A more nuanced view examines expression trees, lambda expressions, and anonymous types as the foundation upon which LINQ is constructed. But, in either case, C# 3.0 presented a revolutionary concept. C# 3.0 had begun to lay the groundwork for turning C# into a hybrid Object-Oriented / Functional language.

Specifically, you could now write SQL-style, declarative queries to perform operations on collections, among other things. Instead of writing a `for` loop to compute the average of a list of integers, you could now do that as simply as `list.Average()`. The combination of query expressions and extension methods made it look as though that list of integers had gotten a whole lot smarter.

It took time for people to really grasp and integrate the concept, but they gradually did. And now, years later, code is much more concise, simple, and functional.

C# version 4.0

C# version 4.0 would have had a difficult time living up to the groundbreaking status of version 3.0. With version 3.0, C# had moved the language firmly out from the shadow of Java and into prominence. The language was quickly becoming elegant.

The next version did introduce some interesting new features:

- Dynamic binding
- Named/optional arguments
- Generic covariant and contravariant
- Embedded interop types

Embedded interop types alleviated a deployment pain. Generic covariance and contravariance give you more power to use generics, but they're a bit academic and probably most appreciated by framework and library authors. Named and optional parameters let you eliminate many method overloads and provide convenience. But none of those features are exactly paradigm altering.

The major feature was the introduction of the `dynamic` keyword. The `dynamic` keyword introduced into C# version 4.0 the ability to override the compiler on compile-time typing. By using the dynamic keyword, you can create constructs similar to dynamically typed languages like JavaScript. You can create a `dynamic x = "a string"` and then add six to it, leaving it up to the runtime to sort out what should happen next.

Dynamic binding gives you the potential for errors but also great power within the language.

C# version 5.0

C# version 5.0 was a focused version of the language. Nearly all of the effort for that version went into another groundbreaking language concept: the `async` and `await` model for asynchronous programming. Here is the major features list:

- Asynchronous members
- Caller info attributes

See Also

- Code Project: Caller Info Attributes in C# 5.0

The caller info attribute lets you easily retrieve information about the context in which you're running without resorting to a ton of boilerplate reflection code. It has many uses in diagnostics and logging tasks.

But `async` and `await` are the real stars of this release. When these features came out in 2012, C# changed the game again by baking asynchrony into the language as a first-class participant. If you've ever dealt with long running operations and the implementation of webs of callbacks, you probably loved this language feature.

C# version 6.0

With versions 3.0 and 5.0, C# had added major new features in an object-oriented language. With version 6.0, it would go away from doing a dominant killer feature and instead release many smaller features that made C# programming more productive. Here are some of them:

- Static imports
- Exception filters
- Auto-property initializers
- Expression bodied members
- Null propagator
- String interpolation
- nameof operator
- Index initializers

Other new features include:

- Await in catch/finally blocks
- Default values for getter-only properties

Each of these features is interesting in its own right. But if you look at them altogether, you see an interesting pattern. In this version, C# eliminated language boilerplate to make code more terse and readable. So for fans of clean, simple code, this language version was a huge win.

They did one other thing along with this version, though it's not a traditional language feature in itself. They released Roslyn the compiler as a service. The C# compiler is now written in C#, and you can use the compiler as part of your programming efforts.

C# version 7.0

The most recent major version is C# version 7.0. This version has some evolutionary and cool stuff in the vein of C# 6.0, but without the compiler as a service. Here are some of the new features:

- Out variables
- Tuples and deconstruction
- Pattern matching
- Local functions
- Expanded expression bodied members
- Ref locals and returns

Other features included:

- Discards
- Binary Literals and Digit Separators
- Throw expressions

All of these features offer cool new capabilities for developers and the opportunity to write even cleaner code than ever. A highlight is condensing the declaration of variables to use with the `out` keyword and by allowing multiple return values via tuple.

But C# is being put to ever broader use. .NET Core now targets any operating system and has its eyes firmly on the cloud and on portability. These new capabilities certainly occupy the language designers' thoughts and time, in addition to coming up with new features.

Article originally published on the NDepend blog, courtesy of Erik Dietrich and Patrick Smacchia.

Relationships between language features and library types

5/4/2018 • 2 minutes to read • Edit Online

The C# language definition requires a standard library to have certain types and certain accessible members on those types. The compiler generates code that uses these required types and members for many different language features. When necessary, there are NuGet packages that contain types needed for newer versions of the language when writing code for environments where those types or members have not been deployed yet.

This dependency on standard library functionality has been part of the C# language since its first version. In that version, examples included:

- Exception - used for all compiler generated exceptions.
- String - the C# `string` type is a synonym for String.
- Int32 - synonym of `int`.

That first version was simple: the compiler and the standard library shipped together, and there was only one version of each.

Subsequent versions of C# have occasionally added new types or members to the dependencies. Examples include: INotifyCompletion, CallerFilePathAttribute and CallerMemberNameAttribute. C# 7.0 continues this by adding a dependency on ValueTuple to implement the tuples language feature.

The language design team works to minimize the surface area of the types and members required in a compliant standard library. That goal is balanced against a clean design where new library features are incorporated seamlessly into the language. There will be new features in future versions of C# that require new types and members in a standard library. It's important to understand how to manage those dependencies in your work.

Managing your dependencies

C# compiler tools are now decoupled from the release cycle of the .NET libraries on supported platforms. In fact, different .NET libraries have different release cycles: the .NET Framework on Windows is released as a Windows Update, .NET Core ships on a separate schedule, and the Xamarin versions of library updates ship with the Xamarin tools for each target platform.

The majority of time, you won't notice these changes. However, when you are working with a newer version of the language that requires features not yet in the .NET libraries on that platform, you'll reference the NuGet packages to provide those new types. As the platforms your app supports are updated with new framework installations, you can remove the extra reference.

This separation means you can use new language features even when you are targeting machines that may not have the corresponding framework.

Version and update considerations for C# developers

5/15/2019 • 2 minutes to read • Edit Online

Compatibility is a very important goal as new features are added to the C# language. In almost all cases, existing code can be recompiled with a new compiler version without any issue.

More care may be required when you adopt new language features in a library. You may be creating a new library with features found in the latest version and need to ensure apps built using previous versions of the compiler can use it. Or you may be upgrading an existing library and many of your users may not have upgraded versions yet. As you make decisions on adopting new features, you'll need to consider two variations of compatibility: source compatible and binary compatible.

Binary compatible changes

Changes to your library are **binary compatible** when your updated library can be used without rebuilding applications and libraries that use it. Dependent assemblies are not required to be rebuilt, nor are any source code changes required. Binary compatible changes are also source compatible changes.

Source compatible changes

Changes to your library are **source compatible** when applications and libraries that use your library do not require source code changes, but the source must be recompiled against the new version to work correctly.

Incompatible changes

If a change is neither **source compatible** nor **binary compatible**, source code changes along with recompilation are required in dependent libraries and applications.

Evaluate your library

These compatibility concepts affect the public and protected declarations for your library, not its internal implementation. Adopting any new features internally are always **binary compatible**.

Binary compatible changes provide new syntax that generates the same compiled code for public declarations as the older syntax. For example, changing a method to an expression-bodied member is a **binary compatible** change:

Original code:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

New code:

```
public double CalculateSquare(double value) => value * value;
```

Source compatible changes introduce syntax that changes the compiled code for a public member, but in a way that is compatible with existing call sites. For example, changing a method signature from a by value parameter to

an `in` by reference parameter is source compatible, but not binary compatible:

Original code:

```
public double CalculateSquare(double value) => value * value;
```

New code:

```
public double CalculateSquare(in double value) => value * value;
```

The What's new articles note if introducing a feature that affects public declarations is source compatible or binary compatible.

Types (C# Programming Guide)

8/19/2019 • 11 minutes to read • Edit Online

Types, Variables, and Values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying Types in Variable Declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and Type Conversions.

Built-in Types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These are available for you to use in any C# program. For more information about the built-in types, see Reference Tables for built-in Types.

Custom Types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see .NET Class Library.

The Common Type System

It is important to understand two fundamental points about the type system in .NET:

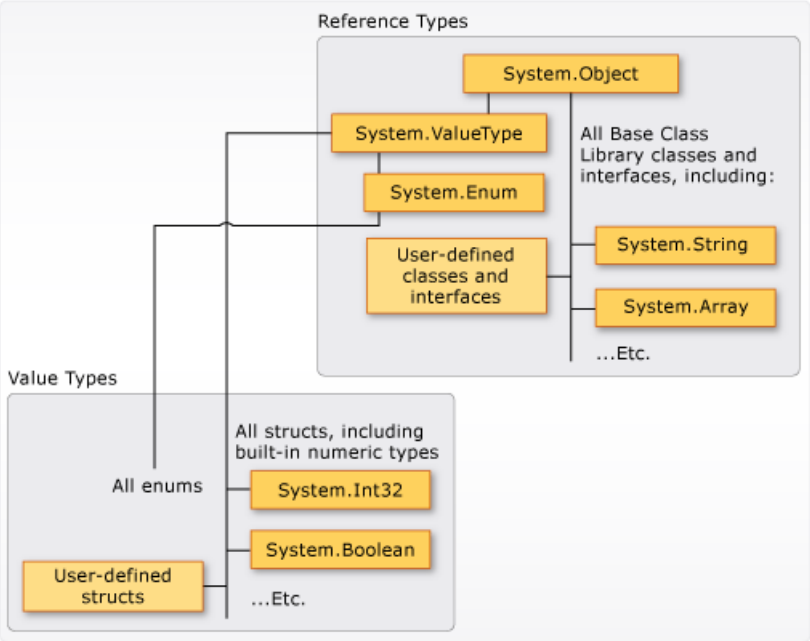
- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of

both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.

The following image shows value types and reference types in the CTS:



> **NOTE**
> You can see that the most commonly used types are all organized in the System namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value Types

Value types derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type byte.
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a System.Object or any interface type as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see Structs. For more information about value types in .NET, see Value Types.

The other category of value types is enum. An enum defines a set of named integral constants. For example, the System.IO.FileMode enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It is defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it is better to use enumerations instead of constant literal numbers. For more information, see System.IO.FileMode.

All enums inherit from System.Enum, which inherits from System.ValueType. All the rules that apply to structs also apply to enums. For more information about enums, see Enumeration Types.

Reference Types

A type that is defined as a class, delegate, array, or interface is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue. For more information about garbage collection, see Automatic Memory Management.

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the System.Array class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see Classes and Structs. For more information about inheritance and virtual methods, see Inheritance.

Types of Literal Values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from System.Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic Types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type System.Collections.Generic.List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see Generics.

Implicit Types, Anonymous Types, and Nullable Types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly Typed Local Variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous Types.

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a `?` after the type. For example, `int?` is an `int` type that can also have the value null. In the CTS, nullable types are instances of the generic struct type System.Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types.

Related Sections

For more information, see the following topics:

- Casting and Type Conversions
- Boxing and Unboxing
- Using Type dynamic
- Value Types
- Reference Types
- Classes and Structs
- Anonymous Types
- Generics

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Conversion of XML Data Types

- Integral types

Nullable reference types

5/24/2019 • 7 minutes to read • Edit Online

C# 8.0 introduces **nullable reference types** and **non-nullable reference types** that enable you to make important statements about the properties for reference type variables:

- **A reference is not supposed to be null**. When variables aren't supposed to be null, the compiler enforces rules that ensure it is safe to dereference these variables without first checking that it isn't null:
  - The variable must be initialized to a non-null value.
  - The variable can never be assigned the value `null`.
- **A reference may be null**. When variables may be null, the compiler enforces different rules to ensure that you've correctly checked for a null reference:
  - The variable may only be dereferenced when the compiler can guarantee that the value isn't null.
  - These variables may be initialized with the default `null` value and may be assigned the value `null` in other code.

This new feature provides significant benefits over the handling of reference variables in earlier versions of C# where the design intent couldn't be determined from the variable declaration. The compiler didn't provide safety against null reference exceptions for reference types:

- **A reference can be null**. No warnings are issued when a reference type is initialized to null, or later assigned to null.
- **A reference is assumed to be not null**. The compiler doesn't issue any warnings when reference types are dereferenced. (With nullable references, the compiler issues warnings whenever you dereference a variable that may be null).

With the addition of nullable reference types, you can declare your intent more clearly. The `null` value is the correct way to represent that a variable doesn't refer to a value. Don't use this feature to remove all `null` values from your code. Rather, you should declare your intent to the compiler and other developers that read your code. By declaring your intent, the compiler informs you when you write code that is inconsistent with that intent.

A **nullable reference type** is noted using the same syntax as nullable value types: a `?` is appended to the type of the variable. For example, the following variable declaration represents a nullable string variable, `name`:

```
string? name;
```

Any variable where the `?` is not appended to the type name is a **non-nullable reference type**. That includes all reference type variables in existing code when you have enabled this feature.

The compiler uses static analysis to determine if a nullable reference is known to be non-null. The compiler warns you if you dereference a nullable reference when it may be null. You can override this behavior by using the **null-forgiving operator** (`!`) following a variable name. For example, if you know the `name` variable isn't null but the compiler issues a warning, you can write the following code to override the compiler's analysis:

```
name!.Length;
```

You can read details about this operator in the draft nullable reference types specification proposal on GitHub.

Nullability of types

Any reference type can have one of four *nullabilities*, which describes when warnings are generated:

- *Nonnullable*: Null can't be assigned to variables of this type. Variables of this type don't need to be null-checked before dereferencing.
- *Nullable*: Null can be assigned to variables of this type. Dereferencing variables of this type without first checking for `null` causes a warning.
- *Oblivious*: This is the pre-C# 8 state. Variables of this type can be dereferenced or assigned without warnings.
- *Unknown*: This is generally for type parameters where constraints don't tell the compiler that the type must be *nullable* or *nonnullable*.

The nullability of a type in a variable declaration is controlled by the *nullable context* in which the variable is declared.

Nullable contexts

Nullable contexts enable fine-grained control for how the compiler interprets reference type variables. The **nullable annotation context** of any given source line is `enabled` or `disabled`. You can think of the pre-C# 8 compiler as compiling all your code in a `disabled` nullable context: Any reference type may be null. The **nullable warnings context** may be set to `enabled`, `disabled`, or `safeonly`. The nullable warnings context specifies the warnings generated by the compiler using its flow analysis.

The nullable annotation context and nullable warning context can be set for a project using the `Nullable` element in your `csproj` file. This element configures how the compiler interprets the nullability of types and what warnings are generated. Valid settings are:

- `enable`: The nullable annotation context is **enabled**. The nullable warning context is **enabled**.
  - Variables of a reference type, `string` for example, are non-nullable. All nullability warnings are enabled.
- `disable`: The nullable annotation context is **disabled**. The nullable warning context is **disabled**.
  - Variables of a reference type are oblivious, just like earlier versions of C#. All nullability warnings are disabled.
- `safeonly`: The nullable annotation context is **enabled**. The nullable warning context is **safeonly**.
  - Variables of a reference type are nonnullable. All safety nullability warnings are enabled.
- `warnings`: The nullable annotation context is **disabled**. The nullable warning context is **enabled**.
  - Variables of a reference type are oblivious. All nullability warnings are enabled.
- `safeonlywarnings`: The nullable annotation context is **disabled**. The nullable warning context is **safeonly**.
  - Variables of a reference type are oblivious. All safety nullability warnings are enabled.

> **IMPORTANT**
> The `Nullable` element was previously named `NullableContextOptions`. The rename ships with Visual Studio 2019, 16.2-p1. The .NET Core SDK 3.0.100-preview5-011568 does not have this change. If you are using the .NET Core CLI, you'll need to use `NullableContextOptions` until the next preview is available.

You can also use directives to set these same contexts anywhere in your project:

- `#nullable enable`: Sets the nullable annotation context and nullable warning context to **enabled**.
- `#nullable disable`: Sets the nullable annotation context and nullable warning context to **disabled**.
- `#nullable safeonly`: Set the nullable annotation context to **enabled**, and the warning context to **safeonly**.
- `#nullable restore`: Restores the nullable annotation context and nullable warning context to the project settings.
- `#pragma warning disable nullable`: Set the nullable warning context to **disabled**.

- `#pragma warning enable nullable`: Set the nullable warning context to **enabled**.
- `#pragma warning restore nullable`: Restores the nullable warning context to the project settings.
- `#pragma warning safeonly nullable`: Sets the nullable warning context to **safeonly**.

The default nullable annotation and warning contexts are `disabled`. That decision means that your existing code compiles without changes and without generating any new warnings.

The differences between the `enabled` and `safeonly` nullable warning contexts are warnings for assigning a nullable reference to a non-nullable reference. The following assignment generates a warning in an `enabled` warning context, but not a `safeonly` warning context. However, the second line, where `s` is dereferenced, generates a warning in a `safeonly` context:

```
string s = null; // warning when nullable warning context is enabled.
var txt = s.ToString(); // warning when nullable warnings context is safeonly, or enabled.
```

Nullable annotation context

The compiler uses the following rules in a disabled nullable annotation context:

- You can't declare nullable references in a disabled context.
- All reference variables may be assigned to null.
- No warnings are generated when a variable of a reference type is dereferenced.
- The null-forgiving operator may not be used in a disabled context.

The behavior is the same as previous versions of C#.

The compiler uses the following rules in an enabled nullable annotation context:

- Any variable of a reference type is a **non-nullable reference**.
- Any non-nullable reference may be dereferenced safely.
- Any nullable reference type (noted by `?` after the type in the variable declaration) may be null. Static analysis determines if the value is known to be non-null when it is dereferenced. If not, the compiler warns you.
- You can use the null-forgiving operator to declare that a nullable reference isn't null.

In an enabled nullable annotation context, the `?` character appended to a reference type declares a **nullable reference type**. The **null forgiveness operator** (`!`) may be appended to an expression to declare that the expression isn't null.

Nullable warning context

The nullable warning context is distinct from the nullable annotation context. Warnings can be enabled even when the new annotations are disabled. The compiler uses static flow analysis to determine the **null state** of any reference. The null state is either **not null** or **maybe null** when the *nullable warning context* isn't **disabled**. If you dereference a reference when the compiler has determined it's **maybe null**, the compiler warns you. The state of a reference is **maybe null** unless the compiler can determine one of two conditions:

1. The variable has been definitely assigned to a non-null value.
2. The variable or expression has been checked against null before de-referencing it.

The compiler generates warnings whenever you dereference a variable or expression in a **maybe null** state when the nullable warning context is `enabled` or `safeonly`. Furthermore, warnings are generated when a **maybe null** variable or expression is assigned to a nonnullable reference type when the nullable annotation context is `enabled`.

Learn more

- Draft Nullable reference specification
- Intro to nullable references tutorial
- Migrate an existing codebase to nullable references

Update libraries to use nullable reference types and communicate nullable rules to callers.

8/7/2019 • 17 minutes to read • Edit Online

The addition of nullable reference types means you can declare whether or not a `null` value is allowed or expected for every variable. That provides a great experience as you write code. You get warnings if a non-nullable variable might be set to `null`. You get warnings if a nullable variable isn't null-checked before you dereference it. Updating your libraries can take time, but the payoffs are worth it. The more information you provide to the compiler about *when* a `null` value is allowed or prohibited, the better warnings users of your API will get. Let's start with a familiar example. Imagine your library has the following API to retrieve a resource string:

```
bool TryGetMessage(string key, out string message)
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference arguments for this API: the `key` and the `message` parameter. This API has the following rules relating to the nullness of these arguments:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.
- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false`, the value of `message` (and its null state) is null.

The rule for `key` can be completely expressed by the variable type: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows `null` as the argument, but guarantees that, on success, that `out` argument isn't null. For these scenarios, you need a richer vocabulary to describe the expectations.

Updating your library for nullable references requires more than sprinkling `?` on some of the variables and type names. The preceding example shows that you need to examine your APIs and consider your expectations for each input argument. Consider the guarantees for the return value, and any `out` or `ref` arguments upon the method's return. Then communicate those rules to the compiler, and the compiler will provide warnings when callers don't abide by those rules.

This work takes time. Let's start with strategies to make your library or application nullable-aware, while balancing other requirements and deliverables. You'll see how to balance ongoing development enabling nullable reference types. You'll learn challenges for generic type definitions. You'll learn to apply attributes to describe pre- and post-conditions on individual APIs.

Choose a nullable strategy

The first choice is whether nullable reference types should be on or off by default. You have two strategies:

- Enable nullable reference types for the entire project, and disable it in code that's not ready.
- Only enable nullable reference types for code that's been annotated for nullable reference types.

The first strategy works best when you're adding other features to the library as you update it for nullable reference types. All new development is nullable aware. As you update existing code, you enable nullable reference types in those classes.

Following this first strategy, you do the following:

1. Enable nullable types for the entire project by adding the `<Nullable>enable</Nullable>` element to your *csproj*

files.
2. Add the `#nullable disable` pragma to every source file in your project.
3. As you work on each file, remove the pragma and address any warnings.

This first strategy has more up-front work to add the pragma to every file. The advantage is that every new code file added to the project will be nullable enabled. Any new work will be nullable aware; only existing code must be updated.

The second strategy works better if the library is generally stable, and the main focus of the development is to adopt nullable reference types. You turn on nullable reference types as you annotate APIs. When you've finished, you enable nullable reference types for the entire project.

Following this second strategy you do the following:

1. Add the `#nullable enable` pragma to the file you want to make nullable aware.
2. Address any warnings.
3. Continue these first two steps until you've made the entire library nullable aware.
4. Enable nullable types for the entire project by adding the `<Nullable>enable</Nullable>` element to your *csproj* files.
5. Remove the `#nullable enable` pragmas, as they're no longer needed.

This second strategy has less work up-front. The tradeoff is that the first task when you create a new file is to add the pragma and make it nullable aware. If any developers on your team forget, that new code is now in the backlog of work to make all code nullable aware.

Which of these strategies you pick depends on how much active development is taking place in your project. The more mature and stable your project, the better the second strategy. The more features being developed, the better the first strategy.

Should nullable warnings introduce breaking changes?

Before you enable nullable reference types, variables are considered *nullable oblivious*. Once you enable nullable reference types, all those variables are *non-nullable*. The compiler will issue warnings if those variables aren't initialized to non-null values.

Another likely source of warnings is return values when the value hasn't been initialized.

The first step in addressing the compiler warnings is to use `?` annotations on parameter and return types to indicate when arguments or return values may be null. When reference variables must not be null, the original declaration is correct. As you do this, your goal isn't just to fix warnings. The more important goal is to make the compiler understand your intent for potential null values. As you examine the warnings, you reach your next major decision for your library. Do you want to consider modifying API signatures to more clearly communicate your design intent? A better API signature for the `TryGetMessage` method examined earlier could be:

```
string? TryGetMessage(string key);
```

The return value indicates success or failure, and carries the value if the value was found. In many cases, changing API signatures can improve how they communicate null values.

However, for public libraries, or libraries with large user bases, you may prefer not introducing any API signature changes. For those cases, and other common patterns, you can apply attributes to more clearly define when an argument or return value may be `null`. Whether or not you consider changing the surface of your API, you'll likely find that type annotations alone aren't sufficient for describing `null` values for arguments or return values. In those instances, you can apply attributes to more clearly describe an API.

Attributes extend type annotations

Several attributes have been added to express additional information about the null state of variables. All code you wrote before C# 8 introduced nullable reference types was *null oblivious*. That means any reference type variable may be null, but null checks aren't required. Once your code is *nullable aware*, those rules change. Reference types should never be the `null` value, and nullable reference types must be checked against `null` before being dereferenced.

The rules for your APIs are likely more complicated, as you saw with the `TryGetValue` API scenario. Many of your APIs have more complex rules for when variables can or can't be `null`. In these cases, you'll use one of the following attributes to express those rules:

- AllowNull: A non-nullable input argument may be null.
- DisallowNull: A nullable input argument should never be null.
- MaybeNull: A non-nullable return value may be null.
- NotNull: A nullable return value will never be null.
- MaybeNullWhen: A non-nullable `out` or `ref` argument may be null when the return value satisfies a condition.
- NotNullWhen: A nullable `out` or `ref` argument may not be null when the return value satisfies a condition.
- NotNullIfNotNull: A return value isn't null if the input argument for the specified parameter isn't null.

The preceding descriptions are a quick reference to what each attribute does. Each following section describes the behavior and meaning more thoroughly.

Adding these attributes gives the compiler more information about the rules for your API. When calling code is compiled in a nullable enabled context, the compiler will warn callers when they violate those rules. These attributes don't enable additional checks on your implementation.

Specify preconditions: `AllowNull` and `DisallowNull`

Consider a read/write property that never returns `null` because it has a reasonable default value. Callers pass `null` to the set accessor when setting it to that default value. For example, consider a messaging system that asks for a screen name in a chat room. If none is provided, the system generates a random name:

```
public string ScreenName
{
   get => screenName;
   set => screenName = value ?? GenerateRandomScreenName();
}
private string screenName;
```

When you compile the preceding code in a nullable oblivious context, everything is fine. Once you enable nullable reference types, the `ScreenName` property becomes a non-nullable reference. That's correct for the `get` accessor: it never returns `null`. Callers don't need to check the returned property for `null`. But now setting the property to `null` generates a warning. In order to continue to support this type of code, you add the System.Diagnostics.CodeAnalysis.AllowNullAttribute attribute to the property, as shown in the following code:

```
[AllowNull]
public string ScreenName
{
   get => screenName;
   set => screenName = value ?? GenerateRandomScreenName();
}
private string screenName = GenerateRandomScreenName();
```

You may need to add a `using` directive for System.Diagnostics.CodeAnalysis to use this and other attributes discussed in this article. The attribute is applied to the property, not the `set` accessor. The `AllowNull` attribute specifies *pre-conditions*, and only applies to inputs. The `get` accessor has a return value, but no input arguments. Therefore, the `AllowNull` attribute only applies to the `set` accessor.

The preceding example demonstrates what to look for when adding the `AllowNull` attribute on an argument:

1. The general contract for that variable is that it shouldn't be `null`, so you want a non-nullable reference type.
2. There are scenarios for the input variable to be `null`, though they aren't the most common usage.

Most often you'll need this attribute for properties, or `in`, `out`, and `ref` arguments. The `AllowNull` attribute is the best choice when a variable is typically non-null, but you need to allow `null` as a precondition.

Contrast that with scenarios for using `DisallowNull`: You use this attribute to specify that an input variable of a nullable type shouldn't be `null`. Consider a property where `null` is the default value, but clients can only set it to a non-null value. Consider the following code:

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

The preceding code is the best way to express your design that the `ReviewComment` could be `null`, but can't be set to `null`. Once this code is nullable aware, you can express this concept more clearly to callers using the System.Diagnostics.CodeAnalysis.DisallowNullAttribute:

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

In a nullable context, the `ReviewComment` `get` accessor could return the default value of `null`. The compiler warns that it must be checked before access. Furthermore, it warns callers that, even though it could be `null`, callers shouldn't explicitly set it to `null`. The `DisallowNull` attribute also specifies a *pre-condition*, it does not affect the `get` accessor. You should choose to use the `DisallowNull` attribute when you observe these characteristics about:

1. The variable could be `null` in core scenarios, often when first instantiated.
2. The variable shouldn't be explicitly set to `null`.

These situations are common in code that was originally *null oblivious*. It may be that object properties are set in two distinct initialization operations. It may be that some properties are set only after some asynchronous work has completed.

The `AllowNull` and `DisallowNull` attributes enable you to specify that preconditions on variables may not match the nullable annotations on those variables. These provide more detail about the characteristics of your API. This additional information helps callers use your API correctly. Remember you specify preconditions using the following attributes:

- AllowNull: A non-nullable input argument may be null.
- DisallowNull: A nullable input argument should never be null.

Specify post-conditions: `MaybeNull` and `NotNull`

Suppose you have a method with the following signature:

```
public Customer FindCustomer(string lastName, string firstName)
```

You've likely written a method like this to return `null` when the name sought wasn't found. The `null` clearly indicates that the record wasn't found. In this example, you'd likely change the return type from `Customer` to `Customer?`. Declaring the return value as a nullable reference type specifies the intent of this API clearly.

For reasons covered under Generic definitions and nullability that technique does not work with generic methods. You may have a generic method that follows a similar pattern:

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> match)
```

You can't specify that the return value is `T?`. The method returns `null` when the sought item isn't found. Since you can't declare a `T?` return type, you add the `MaybeNull` annotation to the method return:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> match)
```

The preceding code informs callers that the contract implies a non-nullable type, but the return value *may* actually be null. Use the `MaybeNull` attribute when your API should be a non-nullable type, typically a generic type parameter, but there may be instances where `null` would be returned.

You can also specify that a return value or an `out` or `ref` argument isn't null even though the type is a nullable type. Consider a method that ensures an array is large enough to hold a number of elements. If the input argument doesn't have capacity, the routine would allocate a new array and copy all the existing elements into it. If the input argument is `null`, the routine would allocate new storage. If there's sufficient capacity, the routine does nothing:

```
public void EnsureCapacity<T>(ref T[] storage, int size)
```

You could call this routine as follows:

```
// messages has the default value (null) when EnsureCapacity is called:
EnsureCapacity<string>(ref messages, 10);
// messages is not null.
EnsureCapacity<string>(messages, 50);
```

After enabling null reference types, you want to ensure that the preceding code compiles without warnings. When the method returns, the `storage` argument is guaranteed to be not null. However, it's acceptable to call `EnsureCapacity` with a null reference. You can make `storage` a nullable reference type, and add the `NotNull` post-condition to the parameter declaration:

```
public void EnsureCapacity<T>([NotNull]ref T[]? storage, int size)
```

The preceding code expresses the existing contract very clearly: Callers can pass a variable with the `null` value, but the return value is guaranteed to never be null. The `NotNull` attribute is most useful for `ref` and `out` arguments where `null` may be passed as an argument, but that argument is guaranteed to be not null when the method returns.

You specify unconditional postconditions using the following attributes:

- MaybeNull: A non-nullable return value may be null.
- NotNull: A nullable return value will never be null.

Specify conditional post-conditions: `NotNullWhen` and `MaybeNullWhen`

You're likely familiar with the `string` method String.IsNullOrEmpty(String). This method returns `true` when the argument isn't null, and not the empty string. It is a form of null-check: Callers don't need to null-check the argument if the method returns `false`. To make a method like this nullable aware, you'd set the argument to a nullable type, and add the `NotNullWhen` attribute:

```
bool IsNullOrEmpty([NotNullWhen(false)]string? value);
```

That informs the compiler that any code where the return value is `false` need not be null-checked. The addition of the attribute informs the compiler's static analysis that `IsNullOrEmpty` performs the necessary null check: when it returns `false`, the input argument is not `null`.

```
string? userInput = GetUserInput();
if (!(string.IsNullOrEmpty(userInput))
{
   int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

The String.IsNullOrEmpty(String) method will be annotated as shown above for .NET Core 3.0. You may have similar methods in your codebase that check the state of objects for null values. The compiler won't recognize custom null check methods, and you'll need to add the annotations yourself. When you add the attribute, the compiler's static analysis knows when the tested variable has been null checked.

Another use for these attributes is the `Try*` pattern. The postconditions for `ref` and `out` variables are communicated through the return value. Consider this method shown earlier:

```
bool TryGetMessage(string key, out string message)
```

The preceding method follows a typical .NET idiom: the return value indicates if `message` was set to the found value or, if no message is found, to the default value. If the method returns `true`, the value of `message` isn't null; otherwise, the method sets `message` to null.

You can communicate that idiom using the `NotNullWhen` attribute. When you update the signature for nullable reference types, make `message` a `string?` and add an attribute:

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
```

In the preceding example, the value of `message` is known to be not null when `TryGetMessage` returns `true`. You should annotate similar methods in your codebase in the same way: the arguments could be `null`, and are known to be not null when the method returns `true`.

There's one final attribute you may also need. Sometimes the null state of a return value depends on the null state of one or more input arguments. These methods will return a non-null value whenever certain input arguments aren't `null`. To correctly annotate these methods, you use the `NotNullIfNotNull` attribute. Consider the following method:

```
string GetTopLevelDomainFromFullUrl(string url);
```

If the `url` argument isn't null, the output isn't `null`. Once nullable references are enabled, that signature works correctly, provided your API never accepts a null input. However, if the input could be null, then return value could also be null. Therefore, you could change the signature to the following code:

```
string? GetTopLevelDomainFromFullUrl(string? url);
```

That also works, but will often force callers to implement extra `null` checks. The contract is that the return value would be `null` only when the input argument `url` is `null`. To express that contract, you would annotate this method as shown in the following code:

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url);
```

The return value and the argument have both been annotated with the `?` indicating that either could be `null`. The attribute further clarifies that the return value won't be null when the `url` argument isn't `null`.

You specify conditional postconditions using these attributes:

- MaybeNullWhen: A non-nullable `out` or `ref` argument may be null when the return value satisfies a condition.
- NotNullWhen: A nullable `out` or `ref` argument may not be null when the return value satisfies a condition.
- NotNullIfNotNull: A return value isn't null if the input argument for the specified parameter isn't null.

Generic definitions and nullability

Correctly communicating the null state of generic types and generic methods requires special care. This stems from the fact that a nullable value type and a nullable reference type are fundamentally different. An `int?` is a synonym for `Nullable<int>`, whereas `string?` is `string` with an attribute added by the compiler. The result is that the compiler can't generate correct code for `T?` without knowing if `T` is a `class` or a `struct`.

This doesn't mean you can't use a nullable type (either value type or reference type) as the type argument for a closed generic type. Both `List<string?>` and `List<int?>` are valid instantiations of `List<T>`.

What it does mean is that you can't use `T?` in a generic class or method declaration without constraints. For example, Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>) won't be changed to return `T?`. You can overcome this limitation by adding either the `struct` or `class` constraint. With either of those constraints, the compiler knows how to generate code for both `T` and `T?`.

You may want to restrict the types used for a generic type argument to be non-nullable types. You can do that by adding the `notnull` constraint on that type argument. When that constraint is applied, the type argument must not be a nullable type.

Conclusions

Adding nullable reference types provides an initial vocabulary to describe your APIs expectations for variables that could be `null`. The additional attributes provide a richer vocabulary to describe the null state of variables as preconditions and postconditions. These attributes more clearly describe your expectations and provide a better experience for the developers using your APIs.

As you update libraries for a nullable context, add these attributes to guide users of your APIs to the correct usage. These attributes help you fully describe the null-state of input arguments and return values:

- AllowNull: A non-nullable input argument may be null.
- DisallowNull: A nullable input argument should never be null.
- MaybeNull: A non-nullable return value may be null.
- NotNull: A nullable return value will never be null.
- MaybeNullWhen: A non-nullable `out` or `ref` argument may be null when the return value satisfies a condition.
- NotNullWhen: A nullable `out` or `ref` argument may not be null when the return value satisfies a condition.
- NotNullIfNotNull: A return value isn't null if the input argument for the specified parameter isn't null.

Namespaces (C# Programming Guide)

8/15/2019 • 2 minutes to read • Edit Online

Namespaces are heavily used in C# programming in two ways. First, the .NET Framework uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

```
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

For more information, see the using Directive.

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# identifier name.

Namespaces Overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET System namespace.

C# language specification

For more information, see the Namespaces section of the C# language specification.

See also

- C# Programming Guide
- Using Namespaces
- How to: Use the My Namespace
- Identifier names
- using Directive
- :: Operator

Types, variables, and values

6/27/2019 • 6 minutes to read • Edit Online

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET Framework class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying types in variable declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and type conversions.

Built-in types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in **string** and **object** types. These are available for you to use in any C# program. For a more information about the built-in types, see Reference table for built-in types.

Custom types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET Framework class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code.

Generic types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET Framework type List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> strings = new List<string>();
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `strings` object in the previous example. For more information, see Generics.

Implicit types, anonymous types, and tuple types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly typed local variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous types.

It's common to want to return more than one value from a method. You can create *tuple types* that return multiple values in a single method call. For more information, see Tuples

The Common type system

It is important to understand two fundamental points about the type system in the .NET Framework:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as Int32 (C# keyword: `int`), derive ultimately from a single base type, which is Object (C# keyword: `object`). This unified type hierarchy is called the Common type system (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET Framework class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are **structs**. For more information about value types, see Structs. Types that you define by using the class keyword are reference types. For more information about reference types, see Classes. Reference types and value types have different compile-time rules, and different run-time behavior.

See also

- Structs
- Classes

Classes (C# Programming Guide)

8/19/2019 • 5 minutes to read • Edit Online

Reference types

A type that is defined as a class is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the new operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue. For more information about garbage collection, see Automatic memory management and garbage collection.

Declaring Classes

Classes are declared by using the class keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

The `class` keyword is preceded by the access level. Because public is used in this case, anyone can create instances of this class. The name of the class follows the `class` keyword. The name of the class must be a valid C# identifier name. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but

does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as this one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it to an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override class virtual methods.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see Inheritance.

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement more than one interface. For more information, see Interfaces.

A class can be declared abstract. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a sealed class does not allow other classes to derive from it. For more information, see Abstract and Sealed Classes and Class Members.

Class definitions can be split between different source files. For more information, see Partial Classes and Methods.

Example

The following example defines a public class that contains an auto-implemented property, a method, and a special method called a constructor. For more information, see Properties, Methods, and Constructors topics. The instances of the class are then instantiated with the `new` keyword.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Object-Oriented Programming
- Polymorphism
- Identifier names
- Members
- Methods

- Constructors
- Finalizers
- Objects

Structs

4/28/2019 • 4 minutes to read • Edit Online

A *struct* is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type Byte.
byte b = Byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to an interface type; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes an Object as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about value types in the .NET Framework, see Common Type System.

Structs share most of the same syntax as classes, although structs are more limited than classes:

- Within a struct declaration, fields cannot be initialized unless they are declared as `const` or `static`.
- A struct cannot declare a parameterless constructor (a constructor without parameters) or a finalizer.
- Structs are copied on assignment. When a struct is assigned to a new variable, all the data is copied, and any

modification to the new copy does not change the data for the original copy. This is important to remember when working with collections of value types such as Dictionary<string, myStruct>.

- Structs are value types and classes are reference types.
- Unlike classes, structs can be instantiated without using a `new` operator.
- Structs can declare constructors that have parameters.
- A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from ValueType, which inherits from Object.
- A struct can implement interfaces.

Literal values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer the `double` type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);

var x = 123_456;
string s2 = "I can use an underscore as a digit separator: " + x;
// Outputs: "I can use an underscore as a digit separator: 123456"
Console.WriteLine(s2);

var b = 0b1010_1011_1100_1110_1111;
string s3 = "I can specify bit patterns: " + b.ToString();
// Outputs: "I can specify bit patterns: 703727"
Console.WriteLine(s3);
```

The last two examples demonstrate language features introduced in C# 7.0. The first allows you to use an underscore character as a *digit separator* inside numeric literals. You can put them wherever you want between digits to improve readability. They have no effect on the value.

The second demonstrates *binary literals*, which allow you to specify bit patterns directly instead of using hexadecimal notation.

Nullable types

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a **?** after the type. For example, **int?** is an **int** type that can also have the value null. In the CTS, nullable types are instances of the generic struct type Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types (C# Programming Guide).

See also

- Classes
- Basic Types

C# tuple types

8/23/2019 • 20 minutes to read • Edit Online

C# tuples are types that you define using a lightweight syntax. The advantages include a simpler syntax, rules for conversions based on number (referred to as cardinality) and types of elements, and consistent rules for copies, equality tests, and assignments. As a tradeoff, tuples do not support some of the object-oriented idioms associated with inheritance. You can get an overview in the section on tuples in the What's new in C# 7.0 article.

In this article, you'll learn the language rules governing tuples in C# 7.0 and later versions, different ways to use them, and initial guidance on working with tuples.

> **NOTE**
>
> The new tuples features require the ValueTuple types. You must add the NuGet package `System.ValueTuple` in order to use it on platforms that do not include the types.
>
> This is similar to other language features that rely on types delivered in the framework. Examples include `async` and `await` relying on the `INotifyCompletion` interface, and LINQ relying on `IEnumerable<T>`. However, the delivery mechanism is changing as .NET is becoming more platform independent. The .NET Framework may not always ship on the same cadence as the language compiler. When new language features rely on new types, those types will be available as NuGet packages when the language features ship. As these new types get added to the .NET Standard API and delivered as part of the framework, the NuGet package requirement will be removed.

Let's start with the reasons for adding new tuple support. Methods return a single object. Tuples enable you to package multiple values in that single object more easily.

The .NET Framework already has generic `Tuple` classes. These classes, however, had two major limitations. For one, the `Tuple` classes named their properties `Item1`, `Item2`, and so on. Those names carry no semantic information. Using these `Tuple` types does not enable communicating the meaning of each of the properties. The new language features enable you to declare and use semantically meaningful names for the elements in a tuple.

The `Tuple` classes cause more performance concerns because they are reference types. Using one of the `Tuple` types means allocating objects. On hot paths, allocating many small objects can have a measurable impact on your application's performance. Therefore, the language support for tuples leverages the new `ValueTuple` structs.

To avoid those deficiencies, you could create a `class` or a `struct` to carry multiple elements. Unfortunately, that's more work for you, and it obscures your design intent. Making a `struct` or `class` implies that you are defining a type with both data and behavior. Many times, you simply want to store multiple values in a single object.

The language features and the `ValueTuple` generic structs enforce the rule that you cannot add any behavior (methods) to these tuple types. All the `ValueTuple` types are *mutable structs*. Each member field is a public field. That makes them very lightweight. However, that means tuples should not be used where immutability is important.

Tuples are both simpler and more flexible data containers than `class` and `struct` types. Let's explore those differences.

Named and unnamed tuples

The `ValueTuple` struct has fields named `Item1`, `Item2`, `Item3`, and so on, similar to the properties defined in the

existing `Tuple` types. These names are the only names you can use for *unnamed tuples*. When you do not provide any alternative field names to a tuple, you've created an unnamed tuple:

```
var unnamed = ("one", "two");
```

The tuple in the previous example was initialized using literal constants and won't have element names created using *tuple field name projections* in C# 7.1.

However, when you initialize a tuple, you can use new language features that give better names to each field. Doing so creates a *named tuple*. Named tuples still have elements named `Item1`, `Item2`, `Item3` and so on. But they also have synonyms for any of those elements that you have named. You create a named tuple by specifying the names for each element. One way is to specify the names as part of the tuple initialization:

```
var named = (first: "one", second: "two");
```

These synonyms are handled by the compiler and the language so that you can use named tuples effectively. IDEs and editors can read these semantic names using the Roslyn APIs. You can reference the elements of a named tuple by those semantic names anywhere in the same assembly. The compiler replaces the names you've defined with `Item*` equivalents when generating the compiled output. The compiled Microsoft Intermediate Language (MSIL) does not include the names you've given these elements.

Beginning with C# 7.1, the field names for a tuple may be provided from the variables used to initialize the tuple. This is referred to as **tuple projection initializers**. The following code creates a tuple named `accumulation` with elements `count` (an integer), and `sum` (a double).

```
var sum = 12.5;
var count = 5;
var accumulation = (count, sum);
```

The compiler must communicate those names you created for tuples that are returned from public methods or properties. In those cases, the compiler adds a TupleElementNamesAttribute attribute on the method. This attribute contains a TransformNames list property that contains the names given to each of the elements in the tuple.

> **NOTE**
> Development Tools, such as Visual Studio, also read that metadata, and provide IntelliSense and other features using the metadata field names.

It is important to understand these underlying fundamentals of the new tuples and the `ValueTuple` type in order to understand the rules for assigning named tuples to each other.

Tuple projection initializers

In general, tuple projection initializers work by using the variable or field names from the right-hand side of a tuple initialization statement. If an explicit name is given, that takes precedence over any projected name. For example, in the following initializer, the elements are `explicitFieldOne` and `explicitFieldTwo`, not `localVariableOne` and `localVariableTwo`:

```
var localVariableOne = 5;
var localVariableTwo = "some text";

var tuple = (explicitFieldOne: localVariableOne, explicitFieldTwo: localVariableTwo);
```

For any field where an explicit name is not provided, an applicable implicit name is projected. There is no requirement to provide semantic names, either explicitly or implicitly. The following initializer has field names `Item1`, whose value is `42` and `stringContent`, whose value is "The answer to everything":

```
var stringContent = "The answer to everything";
var mixedTuple = (42, stringContent);
```

There are two conditions where candidate field names are not projected onto the tuple field:

1. When the candidate name is a reserved tuple name. Examples include `Item3`, `ToString`. or `Rest`.
2. When the candidate name is a duplicate of another tuple field name, either explicit or implicit.

These conditions avoid ambiguity. These names would cause an ambiguity if they were used as the field names for a field in a tuple. Neither of these conditions cause compile-time errors. Instead, the elements without projected names do not have semantic names projected for them. The following examples demonstrate these conditions:

```
var ToString = "This is some text";
var one = 1;
var Item1 = 5;
var projections = (ToString, one, Item1);
// Accessing the first field:
Console.WriteLine(projections.Item1);
// There is no semantic name 'ToString'
// Accessing the second field:
Console.WriteLine(projections.one);
Console.WriteLine(projections.Item2);
// Accessing the third field:
Console.WriteLine(projections.Item3);
// There is no semantic name 'Item1`.

var pt1 = (X: 3, Y: 0);
var pt2 = (X: 3, Y: 4);

var xCoords = (pt1.X, pt2.X);
// There are no semantic names for the fields
// of xCoords.

// Accessing the first field:
Console.WriteLine(xCoords.Item1);
// Accessing the second field:
Console.WriteLine(xCoords.Item2);
```

These situations do not cause compiler errors because that would be a breaking change for code written with C# 7.0, when tuple field name projections were not available.

Equality and tuples

Beginning with C# 7.3, tuple types support the `==` and `!=` operators. These operators work by comparing each member of the left argument to each member of the right argument in order. These comparisons short-circuit. They will stop evaluating members as soon as one pair is not equal. The following code examples use `==`, but the comparison rules all apply to `!=`. The following code example shows an equality comparison for two pairs of integers:

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
Console.WriteLine(left == right); // displays 'true'
```

There are several rules that make tuple equality tests more convenient. Tuple equality performs lifted conversions if one of the tuples is a nullable tuple, as shown in the following code:

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
(int a, int b)? nullableTuple = right;
Console.WriteLine(left == nullableTuple); // Also true
```

Tuple equality also performs implicit conversions on each member of both tuples. These include lifted conversions, widening conversions, or other implicit conversions. The following examples show that an integer 2-tuple can be compared to a long 2-tuple because of the implicit conversion from integer to long:

```
// lifted conversions
var left = (a: 5, b: 10);
(int? a, int? b) nullableMembers = (5, 10);
Console.WriteLine(left == nullableMembers); // Also true

// converted type of left is (long, long)
(long a, long b) longTuple = (5, 10);
Console.WriteLine(left == longTuple); // Also true

// comparisons performed on (long, long) tuples
(long a, int b) longFirst = (5, 10);
(int a, long b) longSecond = (5, 10);
Console.WriteLine(longFirst == longSecond); // Also true
```

The names of the tuple members do not participate in tests for equality. However, if one of the operands is a tuple literal with explicit names, the compiler generates warning CS8383 if those names do not match the names of the other operand. In the case where both operands are tuple literals, the warning is on the right operand as shown in the following example:

```
(int a, string b) pair = (1, "Hello");
(int z, string y) another = (1, "Hello");
Console.WriteLine(pair == another); // true. Member names don't participate.
Console.WriteLine(pair == (z: 1, y: "Hello")); // warning: literal contains different member names
```

Finally, tuples may contain nested tuples. Tuple equality compares the "shape" of each operand through nested tuples as shown in the following example:

```
(int, (int, int)) nestedTuple = (1, (2, 3));
Console.WriteLine(nestedTuple == (1, (2, 3)) );
```

It's a compile time error to compare two tuples for equality (or inequality) when they have different shapes. The compiler won't attempt any deconstruction of nested tuples in order to compare them.

Assignment and tuples

The language supports assignment between tuple types that have the same number of elements, where each right-hand side element can be implicitly converted to its corresponding left-hand side element. Other conversions aren't considered for assignments. It's a compile time error to assign one tuple to another when they have different shapes. The compiler won't attempt any deconstruction of nested tuples in order to assign them.

Let's look at the kinds of assignments that are allowed between tuple types.

Consider these variables used in the following examples:

```
// The 'arity' and 'shape' of all these tuples are compatible.
// The only difference is the field names being used.
var unnamed = (42, "The meaning of life");
var anonymous = (16, "a perfect square");
var named = (Answer: 42, Message: "The meaning of life");
var differentNamed = (SecretConstant: 42, Label: "The meaning of life");
```

The first two variables, `unnamed` and `anonymous` do not have semantic names provided for the elements. The field names are `Item1` and `Item2`. The last two variables, `named` and `differentName` have semantic names given for the elements. These two tuples have different names for the elements.

All four of these tuples have the same number of elements (referred to as 'cardinality') and the types of those elements are identical. Therefore, all of these assignments work:

```
unnamed = named;

named = unnamed;
// 'named' still has fields that can be referred to
// as 'answer', and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");

// unnamed to unnamed:
anonymous = unnamed;

// named tuples.
named = differentNamed;
// The field names are not assigned. 'named' still has
// fields that can be referred to as 'answer' and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");

// With implicit conversions:
// int can be implicitly converted to long
(long, string) conversion = named;
```

Notice that the names of the tuples are not assigned. The values of the elements are assigned following the order of the elements in the tuple.

Tuples of different types or numbers of elements are not assignable:

```
// Does not compile.
// CS0029: Cannot assign Tuple(int,int,int) to Tuple(int, string)
var differentShape = (1, 2, 3);
named = differentShape;
```

Tuples as method return values

One of the most common uses for tuples is as a method return value. Let's walk through one example. Consider this method that computes the standard deviation for a sequence of numbers:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    // Step 1: Compute the Mean:
    var mean = sequence.Average();

    // Step 2: Compute the square of the differences between each number
    // and the mean:
    var squaredMeanDifferences = from n in sequence
                                 select (n - mean) * (n - mean);
    // Step 3: Find the mean of those squared differences:
    var meanOfSquaredDifferences = squaredMeanDifferences.Average();

    // Step 4: Standard Deviation is the square root of that mean:
    var standardDeviation = Math.Sqrt(meanOfSquaredDifferences);
    return standardDeviation;
}
```

NOTE

These examples compute the uncorrected sample standard deviation. The corrected sample standard deviation formula would divide the sum of the squared differences from the mean by (N-1) instead of N, as the `Average` extension method does. Consult a statistics text for more details on the differences between these formulas for standard deviation.

The preceding code follows the textbook formula for the standard deviation. It produces the correct answer, but it's an inefficient implementation. This method enumerates the sequence twice: Once to produce the average, and once to produce the average of the square of the difference of the average. (Remember that LINQ queries are evaluated lazily, so the computation of the differences from the mean and the average of those differences makes only one enumeration.)

There is an alternative formula that computes standard deviation using only one enumeration of the sequence. This computation produces two values as it enumerates the sequence: the sum of all items in the sequence, and the sum of the each value squared:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    double count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

This version enumerates the sequence exactly once. But it's not reusable code. As you keep working, you'll find that many different statistical computations use the number of items in the sequence, the sum of the sequence, and the sum of the squares of the sequence. Let's refactor this method and write a utility method that produces all three of those values. All three values can be returned as a tuple.

Let's update this method so the three values computed during the enumeration are stored in a tuple. That creates this version:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var computation = (Count: 0, Sum: 0.0, SumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.Count++;
        computation.Sum += item;
        computation.SumOfSquares += item * item;
    }

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}
```

Visual Studio's Refactoring support makes it easy to extract the functionality for the core statistics into a private method. That gives you a `private static` method that returns the tuple type with the three values of `Sum`, `SumOfSquares`, and `Count`:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    (int Count, double Sum, double SumOfSquares) computation = ComputeSumAndSumOfSquares(sequence);

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}

private static (int Count, double Sum, double SumOfSquares) ComputeSumAndSumOfSquares(IEnumerable<double>
sequence)
{
    var computation = (count: 0, sum: 0.0, sumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.count++;
        computation.sum += item;
        computation.sumOfSquares += item * item;
    }

    return computation;
}
```

The language enables a couple more options that you can use, if you want to make a few quick edits by hand. First, you can use the `var` declaration to initialize the tuple result from the `ComputeSumAndSumOfSquares` method call. You can also create three discrete variables inside the `ComputeSumAndSumOfSquares` method. The final version is shown in the following code:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var computation = ComputeSumAndSumOfSquares(sequence);

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}

private static (int Count, double Sum, double SumOfSquares) ComputeSumAndSumOfSquares(IEnumerable<double>
sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    int count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    return (count, sum, sumOfSquares);
}
```

This final version can be used for any method that needs those three values, or any subset of them.

The language supports other options in managing the names of the elements in these tuple-returning methods.

You can remove the field names from the return value declaration and return an unnamed tuple:

```
private static (double, double, int) ComputeSumAndSumOfSquares(IEnumerable<double> sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    int count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    return (sum, sumOfSquares, count);
}
```

The fields of this tuple are named `Item1`, `Item2`, and `Item3`. It's recommended that you provide semantic names to the elements of tuples returned from methods.

Another idiom where tuples can be useful is when you are authoring LINQ queries. The final projected result often contains some, but not all, of the properties of the objects being selected.

You would traditionally project the results of the query into a sequence of objects that were an anonymous type. That presented many limitations, primarily because anonymous types could not conveniently be named in the return type for a method. Alternatives using `object` or `dynamic` as the type of the result came with significant performance costs.

Returning a sequence of a tuple type is easy, and the names and types of the elements are available at compile time and through IDE tools. For example, consider a ToDo application. You might define a class similar to the following to represent a single entry in the ToDo list:

```
public class ToDoItem
{
    public int ID { get; set; }
    public bool IsDone { get; set; }
    public DateTime DueDate { get; set; }
    public string Title { get; set; }
    public string Notes { get; set; }
}
```

Your mobile applications may support a compact form of the current ToDo items that only displays the title. That LINQ query would make a projection that includes only the ID and the title. A method that returns a sequence of tuples expresses that design well:

```
internal IEnumerable<(int ID, string Title)> GetCurrentItemsMobileList()
{
    return from item in AllItems
           where !item.IsDone
           orderby item.DueDate
           select (item.ID, item.Title);
}
```

NOTE

In C# 7.1, tuple projections enable you to create named tuples using elements, in a manner similar to the property naming in anonymous types. In the above code, the `select` statement in the query projection creates a tuple that has elements `ID` and `Title`.

The named tuple can be part of the signature. It lets the compiler and IDE tools provide static checking that you are using the result correctly. The named tuple also carries the static type information so there is no need to use expensive run time features like reflection or dynamic binding to work with the results.

Deconstruction

You can unpackage all the items in a tuple by *deconstructing* the tuple returned by a method. There are three different approaches to deconstructing tuples. First, you can explicitly declare the type of each field inside parentheses to create discrete variables for each of the elements in the tuple:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    (int count, double sum, double sumOfSquares) = ComputeSumAndSumOfSquares(sequence);

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

You can also declare implicitly typed variables for each field in a tuple by using the `var` keyword outside the parentheses:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var (sum, sumOfSquares, count) = ComputeSumAndSumOfSquares(sequence);

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

It is also legal to use the `var` keyword with any, or all of the variable declarations inside the parentheses.

```
(double sum, var sumOfSquares, var count) = ComputeSumAndSumOfSquares(sequence);
```

You cannot use a specific type outside the parentheses, even if every field in the tuple has the same type.

You can deconstruct tuples with existing declarations as well:

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

WARNING

You cannot mix existing declarations with declarations inside the parentheses. For instance, the following is not allowed: `(var x, y) = MyMethod();`. This produces error CS8184 because *x* is declared inside the parentheses and *y* is previously declared elsewhere.

Deconstructing user-defined types

Any tuple type can be deconstructed as shown above. It's also easy to enable deconstruction on any user-defined type (classes, structs, or even interfaces).

The type author can define one or more `Deconstruct` methods that assign values to any number of `out` variables representing the data elements that make up the type. For example, the following `Person` type defines a `Deconstruct` method that deconstructs a person object into the elements representing the first name and last name:

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}
```

The deconstruct method enables assignment from a `Person` to two strings, representing the `FirstName` and `LastName` properties:

```
var p = new Person("Althea", "Goodwin");
var (first, last) = p;
```

You can enable deconstruction even for types you did not author. The `Deconstruct` method can be an extension

method that unpackages the accessible data members of an object. The example below shows a `Student` type, derived from the `Person` type, and an extension method that deconstructs a `Student` into three variables, representing the `FirstName`, the `LastName`, and the `GPA`:

```
public class Student : Person
{
    public double GPA { get; }
    public Student(string first, string last, double gpa) :
        base(first, last)
    {
        GPA = gpa;
    }
}

public static class Extensions
{
    public static void Deconstruct(this Student s, out string first, out string last, out double gpa)
    {
        first = s.FirstName;
        last = s.LastName;
        gpa = s.GPA;
    }
}
```

A `Student` object now has two accessible `Deconstruct` methods: the extension method declared for `Student` types, and the member of the `Person` type. Both are in scope, and that enables a `Student` to be deconstructed into either two variables or three. If you assign a student to three variables, the first name, last name, and GPA are all returned. If you assign a student to two variables, only the first name and the last name are returned.

```
var s1 = new Student("Cary", "Totten", 4.5);
var (fName, lName, gpa) = s1;
```

You should be careful defining multiple `Deconstruct` methods in a class or a class hierarchy. Multiple `Deconstruct` methods that have the same number of `out` parameters can quickly cause ambiguities. Callers may not be able to easily call the desired `Deconstruct` method.

In this example, there is minimal chance for an ambiguous call because the `Deconstruct` method for `Person` has two output parameters, and the `Deconstruct` method for `Student` has three.

Deconstruction operators do not participate in testing equality. The following example generates compiler error CS0019:

```
Person p = new Person("Althea", "Goodwin");
if (("Althea", "Goodwin") == p)
    Console.WriteLine(p);
```

The `Deconstruct` method could convert the `Person` object `p` to a tuple containing two strings, but it is not applicable in the context of equality tests.

Conclusion

The new language and library support for named tuples makes it much easier to work with designs that use data structures that store multiple elements but do not define behavior, as classes and structs do. It's easy and concise to use tuples for those types. You get all the benefits of static type checking, without needing to author types using the more verbose `class` or `struct` syntax. Even so, they are most useful for utility methods that are `private`, or `internal`. Create user-defined types, either `class` or `struct` types when your public methods

return a value that has multiple elements.

Deconstructing tuples and other types

7/23/2019 • 10 minutes to read • Edit Online

A tuple provides a lightweight way to retrieve multiple values from a method call. But once you retrieve the tuple, you have to handle its individual elements. Doing this on an element-by-element basis is cumbersome, as the following example shows. The `QueryCityData` method returns a 3-tuple, and each of its elements is assigned to a variable in a separate operation.

```
using System;

public class Example
{
   public static void Main()
   {
       var result = QueryCityData("New York City");

       var city = result.Item1;
       var pop = result.Item2;
       var size = result.Item3;

       // Do something with the data.
   }

   private static (string, int, double) QueryCityData(string name)
   {
      if (name == "New York City")
         return (name, 8175133, 468.48);

      return ("", 0, 0);
   }
}
```

Retrieving multiple field and property values from an object can be equally cumbersome: you have to assign a field or property value to a variable on a member-by-member basis.

Starting with C# 7.0, you can retrieve multiple elements from a tuple or retrieve multiple field, property, and computed values from an object in a single *deconstruct* operation. When you deconstruct a tuple, you assign its elements to individual variables. When you deconstruct an object, you assign selected values to individual variables.

Deconstructing a tuple

C# features built-in support for deconstructing tuples, which lets you unpackage all the items in a tuple in a single operation. The general syntax for deconstructing a tuple is similar to the syntax for defining one: you enclose the variables to which each element is to be assigned in parentheses in the left side of an assignment statement. For example, the following statement assigns the elements of a 4-tuple to four separate variables:

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

There are three ways to deconstruct a tuple:

- You can explicitly declare the type of each field inside parentheses. The following example uses this approach to deconstruct the 3-tuple returned by the `QueryCityData` method.

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- You can use the `var` keyword so that C# infers the type of each variable. You place the `var` keyword outside of the parentheses. The following example uses type inference when deconstructing the 3-tuple returned by the `QueryCityData` method.

  ```
  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  You can also use the `var` keyword individually with any or all of the variable declarations inside the parentheses.

  ```
  public static void Main()
  {
      (string city, var population, var area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  This is cumbersome and is not recommended.

- Lastly, you may deconstruct the tuple into variables that have already been declared.

  ```
  public static void Main()
  {
      string city = "Raleigh";
      int population = 458880;
      double area = 144.8;

      (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

Note that you cannot specify a specific type outside the parentheses even if every field in the tuple has the same type. This generates compiler error CS8136, "Deconstruction 'var (...)' form disallows a specific type for 'var'.".

Note that you must also assign each element of the tuple to a variable. If you omit any elements, the compiler generates error CS8132, "Cannot deconstruct a tuple of 'x' elements into 'y' variables."

Note that you cannot mix declarations and assignments to existing variables on the left-hand side of a deconstruction. The compiler generates error CS8184, "a deconstruction cannot mix declarations and expressions on the left-hand-side." when the members include newly declared and existing variables.

Deconstructing tuple elements with discards

Often when deconstructing a tuple, you're interested in the values of only some elements. Starting with C# 7.0, you can take advantage of C#'s support for *discards*, which are write-only variables whose values you've chosen to

ignore. A discard is designated by an underscore character ("_") in an assignment. You can discard as many values as you like; all are represented by the single discard, `_`.

The following example illustrates the use of tuples with discards. The `QueryCityDataForYears` method returns a 6-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
       var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

       Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
   }

   private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
   {
      int population1 = 0, population2 = 0;
      double area = 0;

      if (name == "New York City") {
         area = 468.48;
         if (year1 == 1960) {
            population1 = 7781984;
         }
         if (year2 == 2010) {
            population2 = 8175133;
         }
      return (name, area, year1, population1, year2, population2);
      }

      return ("", 0, 0, 0, 0, 0);
   }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Deconstructing user-defined types

C# does not offer built-in support for deconstructing non-tuple types. However, as the author of a class, a struct, or an interface, you can allow instances of the type to be deconstructed by implementing one or more `Deconstruct` methods. The method returns void, and each value to be deconstructed is indicated by an out parameter in the method signature. For example, the following `Deconstruct` method of a `Person` class returns the first, middle, and last name:

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

You can then deconstruct an instance of the `Person` class named `p` with an assignment like the following:

```
var (fName, mName, lName) = p;
```

The following example overloads the `Deconstruct` method to return various combinations of properties of a `Person` object. Individual overloads return:

- A first and last name.
- A first, last, and middle name.
- A first name, a last name, a city name, and a state name.

```
using System;

public class Person
{
   public string FirstName { get; set; }
   public string MiddleName { get; set; }
   public string LastName { get; set; }
   public string City { get; set; }
   public string State { get; set; }

   public Person(string fname, string mname, string lname,
                 string cityName, string stateName)
   {
      FirstName = fname;
      MiddleName = mname;
      LastName = lname;
      City = cityName;
      State = stateName;
   }

   // Return the first and last name.
   public void Deconstruct(out string fname, out string lname)
   {
      fname = FirstName;
      lname = LastName;
   }

   public void Deconstruct(out string fname, out string mname, out string lname)
   {
      fname = FirstName;
      mname = MiddleName;
      lname = LastName;
   }

   public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
   {
      fname = FirstName;
      lname = LastName;
      city = City;
      state = State;
   }
}

public class Example
{
   public static void Main()
   {
       Person p = new Person("John", "Quincy", "Adams", "Boston", "MA");

       // Deconstruct the person object.
       var (fName, lName, city, state) = p;
       Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
   }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Because you can overload the `Deconstruct` method to reflect groups of data that are commonly extracted from an object, you should be careful to define `Deconstruct` methods with signatures that are distinctive and unambiguous. Multiple `Deconstruct` methods that have the same number of `out` parameters or the same number and type of `out` parameters in a different order can cause confusion.

The overloaded `Deconstruct` method in the following example illustrates one possible source of confusion. The first overload returns the first name, middle name, last name, and age of a `Person` object, in that order. The second overload returns name information only along with annual income, but the first, middle, and last name are in a different order. This makes it easy to confuse the order of arguments when deconstructing a `Person` instance.

```
using System;

public class Person
{
   public string FirstName { get; set; }
   public  string MiddleName { get; set; }
   public  string LastName { get; set; }
   public  string City { get; set; }
   public  string State { get; set; }
   public DateTime DateOfBirth { get; set; }
   public Decimal AnnualIncome { get; set; }

   public void Deconstruct(out string fname, out string mname, out string lname, out int age)
   {
      fname = FirstName;
      mname = MiddleName;
      lname = LastName;
      age = DateTime.Now.Year - DateOfBirth.Year;

      if (DateTime.Now.DayOfYear - (new DateTime(DateTime.Now.Year, DateOfBirth.Month,
DateOfBirth.Day)).DayOfYear < 0)
         age--;
   }

   public void Deconstruct(out string lname, out string fname, out string mname, out decimal income)
   {
      fname = FirstName;
      mname = MiddleName;
      lname = LastName;
      income = AnnualIncome;
   }
}
```

Deconstructing a user-defined type with discards

Just as you do with tuples, you can use discards to ignore selected items returned by a `Deconstruct` method. Each discard is defined by a variable named "_", and a single deconstruction operation can include multiple discards.

The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state) but discards the last name and the state.

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Deconstructing a user-defined type with an extension method

If you didn't author a class, struct, or interface, you can still deconstruct objects of that type by implementing one or more `Deconstruct` extension methods to return the values in which you're interested.

The following example defines two `Deconstruct` extension methods for the System.Reflection.PropertyInfo class. The first returns a set of values that indicate the characteristics of the property, including its type, whether it's static or instance, whether it's read-only, and whether it's indexed. The second indicates the property's accessibility. Because the accessibility of get and set accessors can differ, Boolean values indicate whether the property has separate get and set accessors and, if it does, whether they have the same accessibility. If there is only one accessor or both the get and the set accessor have the same accessibility, the `access` variable indicates the accessibility of the property as a whole. Otherwise, the accessibility of the get and set accessors are indicated by the `getAccess` and `setAccess` variables.

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
   public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                  out bool isReadOnly, out bool isIndexed,
                                  out Type propertyType)
   {
      var getter = p.GetMethod;

      // Is the property read-only?
      isReadOnly = ! p.CanWrite;

      // Is the property instance or static?
      isStatic = getter.IsStatic;

      // Is the property indexed?
      isIndexed = p.GetIndexParameters().Length > 0;

      // Get the property type.
      propertyType = p.PropertyType;
   }

   public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,
                                  out bool sameAccess, out string access,
                                  out string getAccess, out string setAccess)
   {
      hasGetAndSet = sameAccess = false;
      string getAccessTemp = null;
      string setAccessTemp = null;

      MethodInfo getter = null;
      if (p.CanRead)
         getter = p.GetMethod;

      MethodInfo setter = null;
      if (p.CanWrite)
          setter = p.SetMethod;

      if (setter != null && getter != null)
         hasGetAndSet = true;

      if (getter != null) {
         if (getter.IsPublic)
            getAccessTemp = "public";
         else if (getter.IsPrivate)
            getAccessTemp = "private";
         else if (getter.IsAssembly)
            getAccessTemp = "internal";
         else if (getter.IsFamily)
            getAccessTemp = "protected";
```

```
         else if (getter.IsFamilyOrAssembly)
            getAccessTemp = "protected internal";
      }

      if (setter != null) {
         if (setter.IsPublic)
            setAccessTemp = "public";
         else if (setter.IsPrivate)
            setAccessTemp = "private";
         else if (setter.IsAssembly)
            setAccessTemp = "internal";
         else if (setter.IsFamily)
            setAccessTemp = "protected";
         else if (setter.IsFamilyOrAssembly)
            setAccessTemp = "protected internal";
      }

      // Are the accessibility of the getter and setter the same?
      if (setAccessTemp == getAccessTemp) {
         sameAccess = true;
         access = getAccessTemp;
         getAccess = setAccess = String.Empty;
      }
      else {
         access = null;
         getAccess = getAccessTemp;
         setAccess = setAccessTemp;
      }
   }
}

public class Example
{
   public static void Main()
   {
      Type dateType = typeof(DateTime);
      PropertyInfo prop = dateType.GetProperty("Now");
      var (isStatic, isRO, isIndexed, propType) = prop;
      Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name} property:");
      Console.WriteLine($"   PropertyType: {propType.Name}");
      Console.WriteLine($"   Static:       {isStatic}");
      Console.WriteLine($"   Read-only:    {isRO}");
      Console.WriteLine($"   Indexed:      {isIndexed}");

      Type listType = typeof(List<>);
      prop = listType.GetProperty("Item",
                                  BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.Static);
      var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAccessibility) = prop;
      Console.Write($"\nAccessibility of the {listType.FullName}.{prop.Name} property: ");

      if (!hasGetAndSet | sameAccess) {
          Console.WriteLine(accessibility);
      }
      else {
         Console.WriteLine($"\n   The get accessor: {getAccessibility}");
         Console.WriteLine($"   The set accessor: {setAccessibility}");
      }
   }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item property: public
```

See also

- Discards
- Tuples

Interfaces (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the interface keyword. as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of the struct must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see Related Sections. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be static.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, `Car`, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a get accessor. The class that implements the interface can declare the same property with both a `get` and set accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see Explicit Interface Implementation and Interface Properties.

Interfaces can inherit from other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see Polymorphism.

Interfaces summary

An interface has the following properties:

- An interface is like an abstract base class with only abstract members. Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

In this section

Explicit Interface Implementation
Explains how to create a class member that's specific to an interface.

How to: Explicitly Implement Interface Members
Provides an example of how to explicitly implement members of interfaces.

How to: Explicitly Implement Members of Two Interfaces
Provides an example of how to explicitly implement members of interfaces with inheritance.

Related Sections

- Interface Properties
- Indexers in Interfaces
- How to: Implement Interface Events
- Classes and Structs
- Inheritance
- Methods
- Polymorphism
- Abstract and Sealed Classes and Class Members
- Properties
- Events
- Indexers

featured book chapter

Interfaces in Learning C# 3.0: Master the Fundamentals of C# 3.0

See also

- C# Programming Guide
- Inheritance
- Identifier names

Methods

6/12/2019 • 21 minutes to read • Edit Online

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The `Main` method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

> **NOTE**
> This topic discusses named methods. For information about anonymous functions, see Anonymous Functions.

This topic contains the following sections:

- Method signatures
- Method invocation
- Inherited and overridden methods
- Passing parameters
  - Passing parameters by value
  - Passing parameters by reference
  - Parameter arrays
- Optional parameters and arguments
- Return values
- Extension methods
- Async Methods
- Expression-bodied members
- Iterators

Method signatures

Methods are declared in a `class` or `struct` by specifying:

- An optional access level, such as `public` or `private`. The default is `private`.
- Optional modifiers such as `abstract` or `sealed`.
- The return value, or `void` if the method has none.
- The method name.
- Any method parameters. Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters.

These parts together form the method signature.

> **NOTE**
> A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

The following example defines a class named `Motorcycle` that contains five methods:

```
using System;

abstract class Motorcycle
{
   // Anyone can call this.
   public void StartEngine() {/* Method statements here */ }

   // Only derived classes can call this.
   protected void AddGas(int gallons) { /* Method statements here */ }

   // Derived classes can override the base class implementation.
   public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

   // Derived classes can override the base class implementation.
   public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

   // Derived classes must implement this.
   public abstract double GetTopSpeed();
}
```

Note that the `Motorcycle` class includes an overloaded method, `Drive`. Two methods have the same name, but must be differentiated by their parameter types.

Method invocation

Methods can be either *instance* or *static*. Invoking an instance method requires that you instantiate an object and call the method on that object; an instance method operates on that instance and its data. You invoke a static method by referencing the name of the type to which the method belongs; static methods do not operate on instance data. Attempting to call a static method through an object instance generates a compiler error.

Calling a method is like accessing a field. After the object name (if you are calling an instance method) or the type name (if you are calling a `static` method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, does not have to be the same as the parameter named defined in the method. In the following example, the `Square` method includes a single parameter of type `int` named *i*. The first method call passes the `Square` method a variable of type `int` named *num*; the second, a numeric constant; and the third, an expression.

```
public class Example
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

The most common form of method invocation used positional arguments; it supplies arguments in the same order as method parameters. The methods of the `Motorcycle` class can therefore be called as in the following example. The call to the `Drive` method, for example, includes two arguments that correspond to the two parameters in the method's syntax. The first becomes the value of the `miles` parameter, the second the value of the `speed` parameter.

```
class TestMotorcycle : Motorcycle
{
   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();

      moto.StartEngine();
      moto.AddGas(15);
      moto.Drive(5, 20);
      double speed = moto.GetTopSpeed();
      Console.WriteLine("My top speed is {0}", speed);
   }
}
```

You can also used *named arguments* instead of positional arguments when invoking a method. When using named arguments, you specify the parameter name followed by a colon (":") and the argument. Arguments to the method can appear in any order, as long as all required arguments are present. The following example uses named arguments to invoke the `TestMotorcycle.Drive` method. In this example, the named arguments are passed in the opposite order from the method's parameter list.

```
using System;

class TestMotorcycle : Motorcycle
{
   public override int Drive(int miles, int speed)
   {
      return (int) Math.Round( ((double)miles) / speed, 0);
   }

   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();
      moto.StartEngine();
      moto.AddGas(15);
      var travelTime = moto.Drive(speed: 60, miles: 170);
      Console.WriteLine("Travel time: approx. {0} hours", travelTime);
   }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

You can invoke a method using both positional arguments and named arguments. However, a positional argument cannot follow a named argument. The following example invokes the `TestMotorcycle.Drive` method from the previous example using one positional argument and one named argument.

```
var travelTime = moto.Drive(170, speed: 55);
```

Inherited and overridden methods

In addition to the members that are explicitly defined in a type, a type inherits members defined in its base classes. Since all types in the managed type system inherit directly or indirectly from the Object class, all types inherit its members, such as Equals(Object), GetType(), and ToString(). The following example defines a `Person` class, instantiates two `Person` objects, and calls the `Person.Equals` method to determine whether the two objects are equal. The `Equals` method, however, is not defined in the `Person` class; it is inherited from Object.

```
using System;

public class Person
{
   public String FirstName;
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: False
```

Types can override inherited members by using the `override` keyword and providing an implementation for the overridden method. The method signature must be the same as that of the overridden method. The following example is like the previous one, except that it overrides the Equals(Object) method. (It also overrides the GetHashCode() method, since the two methods are intended to provide consistent results.)

```
using System;

public class Person
{
   public String FirstName;

   public override bool Equals(object obj)
   {
      var p2 = obj as Person;
      if (p2 == null)
         return false;
      else
         return FirstName.Equals(p2.FirstName);
   }

   public override int GetHashCode()
   {
      return FirstName.GetHashCode();
   }
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: True
```

Passing parameters

Types in C# are either *value types* or *reference types*. For a list of built-in value types, see Types and variables. By default, both value types and reference types are passed to a method by value.

Passing parameters by value

When a value type is passed to a method by value, a copy of the object instead of the object itself is passed to the method. Therefore, changes to the object in the called method have no effect on the original object when control returns to the caller.

The following example passes a value type to a method by value, and the called method attempts to change the value type's value. It defines a variable of type `int`, which is a value type, initializes its value to 20, and passes it to a method named `ModifyValue` that changes the variable's value to 30. When the method returns, however, the variable's value remains unchanged.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

When an object of a reference type is passed to a method by value, a reference to the object is passed by value. That is, the method receives not the object itself, but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the object when control returns to the calling method. However, replacing the object passed to the method has no effect on the original object when control returns to the caller.

The following example defines a class (which is a reference type) named `SampleRefType`. It instantiates a `SampleRefType` object, assigns 44 to its `value` field, and passes the object to the `ModifyObject` method. This example does essentially the same thing as the previous example -- it passes an argument by value to a method. But because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `obj.value` field also changes the `value` field of the argument, `rt`, in the `Main` method to 33, as the output from the example shows.

```
using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Passing parameters by reference

You pass a parameter by reference when you want to change the value of an argument in a method and want to reflect that change when control returns to the calling method. To pass a parameter by reference, you use the `ref` or `out` keyword. You can also pass a value by reference to avoid copying but still prevent modifications using the `in` keyword.

The following example is identical to the previous one, except the value is passed by reference to the `ModifyValue` method. When the value of the parameter is modified in the `ModifyValue` method, the change in value is reflected when control returns to the caller.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

A common pattern that uses by ref parameters involves swapping the values of variables. You pass two variables to a method by reference, and the method swaps their contents. The following example swaps integer values.

```
using System;

public class Example
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passing a reference-type parameter allows you to change the value of the reference itself, rather than the value of its individual elements or fields.

Parameter arrays

Sometimes, the requirement that you specify the exact number of arguments to your method is restrictive. By using the `params` keyword to indicate that a parameter is a parameter array, you allow your method to be called with a variable number of arguments. The parameter tagged with the `params` keyword must be an array type, and it must be the last parameter in the method's parameter list.

A caller can then invoke the method in either of three ways:

- By passing an array of the appropriate type that contains the desired number of elements.
- By passing a comma-separated list of individual arguments of the appropriate type to the method.
- By not providing an argument to the parameter array.

The following example defines a method named `GetVowels` that returns all the vowels from a parameter array. The `Main` method illustrates all three ways of invoking the method. Callers are not required to supply any arguments for parameters that include the `params` modifier. In that case, the parameter is `null`.

```
using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from no value: ''
```

Optional parameters and arguments

A method definition can specify that its parameters are required or that they are optional. By default, parameters are required. Optional parameters are specified by including the parameter's default value in the method definition. When the method is called, if no argument is supplied for an optional parameter, the default value is used instead.

The parameter's default value must be assigned by one of the following kinds of expressions:

- A constant, such as a literal string or number.
- An expression of the form `new ValType`, where `ValType` is a value type. Note that this invokes the value type's implicit parameterless constructor, which is not an actual member of the type.
- An expression of the form `default(ValType)`, where `ValType` is a value type.

If a method includes both required and optional parameters, optional parameters are defined at the end of the parameter list, after all required parameters.

The following example defines a method, `ExampleMethod`, that has one required and two optional parameters.

```
using System;

public class Options
{
   public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
   {
      Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                        optionalInt, required + optionalInt);
   }
}
```

If a method with multiple optional arguments is invoked using positional arguments, the caller must supply an argument for all optional parameters from the first one to the last one for which an argument is supplied. In the case of the `ExampleMethod` method, for example, if the caller supplies an argument for the `description` parameter, it must also supply one for the `optionalInt` parameter. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` is a valid method call; `opt.ExampleMethod(2, , "Addition of 2 and 0");` generates an "Argument missing" compiler error.

If a method is called using named arguments or a combination of positional and named arguments, the caller can omit any arguments that follow the last positional argument in the method call.

The following example calls the `ExampleMethod` method three times. The first two method calls use positional arguments. The first omits both optional arguments, while the second omits the last argument. The third method call supplies a positional argument for the required parameter, but uses a named argument to supply a value to the `description` parameter while omitting the `optionalInt` argument.

```
public class Example
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

The use of optional parameters affects *overload resolution*, or the way in which the C# compiler determines which particular overload should be invoked by a method call, as follows:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by

a variable, constant, or expression that matches the return type will return that value to the method caller. Methods with a non-void return type are required to use the `return` keyword to return a value. The `return` keyword also stops the execution of the method.

If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block.

For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

Sometimes, you want your method to return more than a single value. Starting with C# 7.0, you can do this easily by using *tuple types* and *tuple literals*. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)` defines the tuple type that is returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` is the tuple literal; the method returns the first, middle, and last name, along with the age, of a `PersonInfo` object.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The caller can then consume the returned tuple with code like the following:

```
var person = GetPersonalInfo("111111111")
Console.WriteLine("{person.Item1} {person.Item3}: age = {person.Item4}");
```

Names can also be assigned to the tuple elements in the tuple type definition. The following example shows an alternate version of the `GetPersonalInfo` method that uses named elements:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The previous call to the `GetPersonInfo` method can then be modified as follows:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine("{person.FName} {person.LName}: age = {person.Age}");
```

If a method is passed an array as an argument and modifies the value of individual elements, it is not necessary for the method to return the array, although you may choose to do so for good style or functional flow of values. This is because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the following example, changes to the contents of the `values` array that are made in the `DoubleValues` method are observable by any code that has a reference to the array.

```
using System;

public class Example
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Extension methods

Ordinarily, there are two ways to add a method to an existing type:

- Modify the source code for that type. You cannot do this, of course, if you do not own the type's source code. And this becomes a breaking change if you also add any private data fields to support the method.
- Define the new method in a derived class. A method cannot be added in this way using inheritance for other types, such as structures and enumerations. Nor can it be used to "add" a method to a sealed class.

Extension methods let you "add" a method to an existing type without modifying the type itself or implementing the new method in an inherited type. The extension method also does not have to reside in the same assembly as the type it extends. You call an extension method as if it were a defined member of a type.

For more information, see Extension Methods.

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller if the awaited task is not completed, and progress in the method with the `await` keyword is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **NOTE**
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method can have a return type of Task<TResult>, Task, or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. Starting with C# 7.0, an async method can have any task-like return type.

In the following example, `DelayAsync` is an async method that has a return statement that returns an integer. Because it is an async method, its method declaration must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer, as the following `int result = await delayTask` statement demonstrates.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

public class Example
{
    // This Click event is marked with the async modifier.
    public static void Main()
    {
       DoSomethingAsync().Wait();
    }

    private static async Task DoSomethingAsync()
    {
        int result = await DelayAsync();
        Console.WriteLine("Result: " + result);
    }

    private static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }

    // Output:
    //  Result: 5
}
// The example displays the following output:
//        Result: 5
```

An async method can't declare any in, ref, or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous Programming with Async and Await, Control Flow in Async Programs, and Async Return Types.

Expression-bodied members

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read-only, and you do not use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location is remembered so that the caller can request the next element in the sequence.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

See also

- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- out
- ref
- in
- Passing Parameters

Lambda expressions (C# Programming Guide)

7/31/2019 • 11 minutes to read • Edit Online

A *lambda expression* is an expression of any of the following two forms:

- Expression lambda that has an expression as its body:

```
(input-parameters) => expression
```

- Statement lambda that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

Use the lambda declaration operator `=>` to separate the lambda's parameter list from its body. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Any lambda expression can be converted to a delegate type. The delegate type to which a lambda expression can be converted is defined by the types of its parameters and return value. If a lambda expression doesn't return a value, it can be converted to one of the `Action` delegate types; otherwise, it can be converted to one of the `Func` delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an Action<T1,T2> delegate. A lambda expression that has one parameter and returns a value can be converted to a Func<T,TResult> delegate. In the following example, the lambda expression `x => x * x`, which specifies a parameter that's named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Expression lambdas also can be converted to the expression tree types, as the following example shows:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

You can use lambda expressions in any code that requires instances of delegate types or expression trees, for example as an argument to the Task.Run(Action) method to pass the code that should be executed in the background. You also can use lambda expressions when you write LINQ query expressions, as the following example shows:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

When you use method-based syntax to call the Enumerable.Select method in the System.Linq.Enumerable class,

for example in LINQ to Objects and LINQ to XML, the parameter is a delegate type System.Func<T,TResult>. When you call the Queryable.Select method in the System.Linq.Queryable class, for example in LINQ to SQL, the parameter type is an expression tree type `Expression<Func<TSource,TResult>>`. In both cases you can use the same lambda expression to specify the parameter value. That makes the two `Select` calls to look similar although in fact the type of objects created from the lambdas is different.

Expression lambdas

A lambda expression with an expression on the right side of the `=>` operator is called an *expression lambda*. Expression lambdas are used extensively in the construction of expression trees. An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

The parentheses are optional only if the lambda has one input parameter; otherwise they are required.

Specify zero input parameters with empty parentheses:

```
Action line = () => Console.WriteLine();
```

Two or more input parameters are separated by commas enclosed in parentheses:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Sometimes it's impossible for the compiler to infer the input types. You can specify the types explicitly as shown in the following example:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Input parameter types must be all explicit or all implicit; otherwise, a CS0748 compiler error occurs.

The body of an expression lambda can consist of a method call. However, if you are creating expression trees that are evaluated outside the context of the .NET common language runtime, such as in SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

Statement lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

```
(input-parameters) => { <sequence-of-statements> }
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Statement lambdas cannot be used to create expression trees.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the async and await keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

For more information about how to create and use async methods, see Asynchronous Programming with async

and await.

Lambda expressions and tuples

Starting with C# 7.0, the C# language provides built-in support for tuples. You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple components.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, etc. You can, however, define a tuple with named components, as the following example does.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

For more information about C# tuples, see C# tuple types.

Lambdas with the standard query operators

LINQ to Objects, among other implementations, have an input parameter whose type is one of the Func<TResult> family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the Func<T,TResult> delegate type:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a `Func<int, bool>` instance where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. For example, `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it's invoked, returns Boolean value that indicates whether the input parameter is equal to five:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

You can also supply a lambda expression when the argument type is an Expression<TDelegate>, for example in the standard query operators that are defined in the Queryable type. When you specify an Expression<TDelegate> argument, the lambda is compiled to an expression tree.

The following example uses the Count standard query operator:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following example produces a sequence that contains all elements in the `numbers` array that precede the 9, because that's the first number in the sequence that doesn't meet the condition:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

The following example specifies multiple input parameters by enclosing them in parentheses. The method returns all the elements in the `numbers` array until it encounters a number whose value is less than its ordinal position in the array:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you are querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Note that lambda expressions in themselves don't have a type because the common type system has no intrinsic concept of "lambda expression." However, it's sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or Expression type to which the lambda expression is converted.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be

garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the enclosing method.
- A lambda expression cannot directly capture an in, ref, or out parameter from the enclosing method.
- A return statement in a lambda expression doesn't cause the enclosing method to return.

- A lambda expression cannot contain a goto, break, or continue statement if the target of that jump statement is outside the lambda expression block. It's also an error to have a jump statement outside the lambda expression block if the target is inside the block.

C# language specification

For more information, see the Anonymous function expressions section of the C# language specification.

Featured book chapter

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

See also

- C# Programming Guide
- LINQ (Language-Integrated Query)
- Expression Trees
- Local functions compared to lambda expressions
- Implicitly typed lambda expressions
- Visual Studio 2008 C# Samples (see LINQ Sample Queries files and XQuery program)
- Recursive lambda expressions

Properties

8/19/2019 • 9 minutes to read • Edit Online

Properties are first class citizens in C#. The language defines syntax that enables developers to write code that accurately expresses their design intent.

Properties behave like fields when they are accessed. However, unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned.

Property syntax

The syntax for properties is a natural extension to fields. A field defines a storage location:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

A property definition contains declarations for a `get` and `set` accessor that retrieves and assigns the value of that property:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

The syntax shown above is the *auto property* syntax. The compiler generates the storage location for the field that backs up the property. The compiler also implements the body of the `get` and `set` accessors.

Sometimes, you need to initialize a property to a value other than the default for its type. C# enables that by setting a value after the closing brace for the property. You may prefer the initial value for the `FirstName` property to be the empty string rather than `null`. You would specify that as shown below:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

Specific initialization is most useful for read-only properties, as you'll see later in this article.

You can also define the storage yourself, as shown below:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

When a property implementation is a single expression, you can use *expression-bodied members* for the getter or setter:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

This simplified syntax will be used where applicable throughout this article.

The property definition shown above is a read-write property. Notice the keyword `value` in the set accessor. The `set` accessor always has a single parameter named `value`. The `get` accessor must return a value that is convertible to the type of the property (`string` in this example).

That's the basics of the syntax. There are many different variations that support a variety of different design idioms. Let's explore, and learn the syntax options for each.

Scenarios

The examples above showed one of the simplest cases of property definition: a read-write property with no validation. By writing the code you want in the `get` and `set` accessors, you can create many different scenarios.

Validation

You can write code in the `set` accessor to ensure that the values represented by a property are always valid. For example, suppose one rule for the `Person` class is that the name cannot be blank or white space. You would write that as follows:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

The preceding example can be simplified by using a `throw` expression as part of the property setter validation:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

The example above enforces the rule that the first name must not be blank or white space. If a developer writes

```
hero.FirstName = "";
```

That assignment throws an `ArgumentException`. Because a property set accessor must have a void return type, you report errors in the set accessor by throwing an exception.

You can extend this same syntax to anything needed in your scenario. You can check the relationships between different properties, or validate against any external conditions. Any valid C# statements are valid in a property accessor.

Read-only

Up to this point, all the property definitions you have seen are read/write properties with public accessors. That's not the only valid accessibility for properties. You can create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in that class. You could give the set accessor `private` accessibility instead of `public`:

```
public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}
```

Now, the `FirstName` property can be accessed from any code, but it can only be assigned from other code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. Any access modifier you place on the individual accessor must be more limited than the access modifier on the property definition. The above is legal because the `FirstName` property is `public`, but the set accessor is `private`. You could not declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or, even `private`.

It is also legal to place the more restrictive modifier on the `get` accessor. For example, you could have a `public` property, but restrict the `get` accessor to `private`. That scenario is rarely done in practice.

You can also restrict modifications to a property so that it can only be set in a constructor or a property initializer. You can modify the `Person` class so as follows:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

This feature is most commonly used for initializing collections that are exposed as read-only properties:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

Computed properties

A property does not need to simply return the value of a member field. You can create properties that return a computed value. Let's expand the `Person` object to return the full name, computed by concatenating the first and last names:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

The example above uses the string interpolation feature to create the formatted string for the full name.

You can also use an *expression-bodied member*, which provides a more succinct way to create the computed `FullName` property:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Expression-bodied members use the *lambda expression* syntax to define methods that contain a single expression.

Here, that expression returns the full name for the person object.

Cached evaluated properties

You can mix the concept of a computed property with storage and create a *cached evaluated property*. For example, you could update the `FullName` property so that the string formatting only happened the first time it was accessed:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

The above code contains a bug though. If code updates the value of either the `FirstName` or `LastName` property, the previously evaluated `fullName` field is invalid. You modify the `set` accessors of the `FirstName` and `LastName` property so that the `fullName` field is calculated again:

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

This final version evaluates the `FullName` property only when needed. If the previously calculated version is valid, it's used. If another state change invalidates the previously calculated version, it will be recalculated. Developers that use this class do not need to know the details of the implementation. None of these internal changes affect the use of the Person object. That's the key reason for using Properties to expose data members of an object.

Attaching attributes to auto-implemented properties

Beginning with C# 7.3, field attributes can be attached to the compiler generated backing field in auto-implemented properties. For example, consider a revision to the `Person` class that adds a unique integer `Id` property. You write the `Id` property using an auto-implemented property, but your design does not call for persisting the `Id` property. The NonSerializedAttribute can only be attached to fields, not properties. You can attach the NonSerializedAttribute to the backing field for the `Id` property by using the `field:` specifier on the attribute, as shown in the following example:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

This technique works for any attribute you attach to the backing field on the auto-implemented property.

Implementing INotifyPropertyChanged

A final scenario where you need to write code in a property accessor is to support the INotifyPropertyChanged interface used to notify data binding clients that a value has changed. When the value of a property changes, the object raises the INotifyPropertyChanged.PropertyChanged event to indicate the change. The data binding libraries, in turn, update display elements based on that change. The code below shows how you would implement `INotifyPropertyChanged` for the `FirstName` property of this person class.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

The `?.` operator is called the *null conditional operator*. It checks for a null reference before evaluating the right side of the operator. The end result is that if there are no subscribers to the `PropertyChanged` event, the code to raise the event doesn't execute. It would throw a `NullReferenceException` without this check in that case. For more information, see `events`. This example also uses the new `nameof` operator to convert from the property name symbol to its text representation. Using `nameof` can reduce errors where you have mistyped the name of the property.

Again, implementing INotifyPropertyChanged is an example of a case where you can write code in your accessors to support the scenarios you need.

Summing up

Properties are a form of smart fields in a class or object. From outside the object, they appear like fields in the object. However, properties can be implemented using the full palette of C# functionality. You can provide validation, different accessibility, lazy evaluation, or any requirements your scenarios need.

Indexers

10/16/2018 • 9 minutes to read • Edit Online

Indexers are similar to properties. In many ways indexers build on the same language features as properties. Indexers enable *indexed* properties: properties referenced using one or more arguments. Those arguments provide an index into some collection of values.

Indexer Syntax

You access an indexer through a variable name and square brackets. You place the indexer arguments inside the brackets:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

You declare indexers using the `this` keyword as the property name, and declaring the arguments within square brackets. This declaration would match the usage shown in the previous paragraph:

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

From this initial example, you can see the relationship between the syntax for properties and for indexers. This analogy carries through most of the syntax rules for indexers. Indexers can have any valid access modifiers (public, protected internal, protected, internal, private or private protected). They may be sealed, virtual, or abstract. As with properties, you can specify different access modifiers for the get and set accessors in an indexer. You may also specify read-only indexers (by omitting the set accessor), or write-only indexers (by omitting the get accessor).

You can apply almost everything you learn from working with properties to indexers. The only exception to that rule is *auto implemented properties*. The compiler cannot always generate the correct storage for an indexer.

The presence of arguments to reference an item in a set of items distinguishes indexers from properties. You may define multiple indexers on a type, as long as the argument lists for each indexer is unique. Let's explore different scenarios where you might use one or more indexers in a class definition.

Scenarios

You would define *indexers* in your type when its API models some collection where you define the arguments to that collection. Your indexers may or may not map directly to the collection types that are part of the .NET core framework. Your type may have other responsibilities in addition to modeling a collection. Indexers enable you to provide the API that matches your type's abstraction without exposing the inner details of how the values for that abstraction are stored or computed.

Let's walk through some of the common scenarios for using *indexers*. You can access the sample folder for indexers. For download instructions, see Samples and Tutorials.

Arrays and Vectors

One of the most common scenarios for creating indexers is when your type models an array, or a vector. You can create an indexer to model an ordered list of data.

The advantage of creating your own indexer is that you can define the storage for that collection to suit your needs. Imagine a scenario where your type models historical data that is too large to load into memory at once. You need to load and unload sections of the collection based on usage. The example following models this behavior. It reports on how many data points exist. It creates pages to hold sections of the data on demand. It removes pages from memory to make room for pages needed by more recent requests.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }

    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();

    public DataSamples(int totalSize)
    {
        this.totalSize = totalSize;
    }

    public Measurements this[int index]
```

```
    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}
```

You can follow this design idiom to model any sort of collection where there are good reasons not to load the entire set of data into an in- memory collection. Notice that the `Page` class is a private nested class that is not part of the public interface. Those details are hidden from any users of this class.

Dictionaries

Another common scenario is when you need to model a dictionary or a map. This scenario is when your type stores values based on key, typically text keys. This example creates a dictionary that maps command line arguments to lambda expressions that manage those options. The following example shows two classes: an `ArgsActions` class that maps a command line option to an `Action` delegate, and an `ArgsProcessor` that uses the

`ArgsActions` to execute each `Action` when it encounters that option.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

In this example, the `ArgsAction` collection maps closely to the underlying collection. The `get` determines if a given option has been configured. If so, it returns the `Action` associated with that option. If not, it returns an `Action` that does nothing. The public accessor does not include a `set` accessor. Rather, the design using a public method for setting options.

Multi-Dimensional Maps

You can create indexers that use multiple arguments. In addition, those arguments are not constrained to be the same type. Let's look at two examples.

The first example shows a class that generates values for a Mandelbrot set. For more information on the mathematics behind the set, read this article. The indexer uses two doubles to define a point in the X, Y plane. The get accessor computes the number of iterations until a point is determined to be not in the set. If the maximum iterations is reached, the point is in the set, and the class's maxIterations value is returned. (The computer generated images popularized for the Mandelbrot set define colors for the number of iterations necessary to determine that a point is outside the set.

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

The Mandelbrot Set defines values at every (x,y) coordinate for real number values. That defines a dictionary that could contain an infinite number of values. Therefore, there is no storage behind the set. Instead, this class computes the value for each point when code calls the `get` accessor. There's no underlying storage used.

Let's examine one last use of indexers, where the indexer takes multiple arguments of different types. Consider a program that manages historical temperature data. This indexer uses a city and a date to set or get the high and low temperatures for that location:

```
using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}
```

This example creates an indexer that maps weather data on two different arguments: a city (represented by a `string`) and a date (represented by a `DateTime`). The internal storage uses two `Dictionary` classes to represent the two-dimensional dictionary. The public API no longer represents the underlying storage. Rather, the language features of indexers enables you to create a public interface that represents your abstraction, even though the underlying storage must use different core collection types.

There are two parts of this code that may be unfamiliar to some developers. These two `using` statements:

```
using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;
```

create an *alias* for a constructed generic type. Those statements enable the code later to use the more descriptive `DateMeasurements` and `CityDateMeasurements` names instead of the generic construction of `Dictionary<DateTime, Measurements>` and `Dictionary<string, Dictionary<DateTime, Measurements> >`. This construct does require using the fully qualified type names on the right side of the `=` sign.

The second technique is to strip off the time portions of any `DateTime` object used to index into the collections. The

.NET framework does not include a Date only type. Developers use the `DateTime` type, but use the `Date` property to ensure that any `DateTime` object from that day are equal.

Summing Up

You should create indexers anytime you have a property-like element in your class where that property represents not a single value, but rather a collection of values where each individual item is identified by a set of arguments. Those arguments can uniquely identify which item in the collection should be referenced. Indexers extend the concept of properties, where a member is treated like a data item from outside the class, but like a method on the side. Indexers allow arguments to find a single item in a property that represents a set of items.

Discards - C# Guide

10/16/2018 • 7 minutes to read • Edit Online

Starting with C# 7.0, C# supports discards, which are temporary, dummy variables that are intentionally unused in application code. Discards are equivalent to unassigned variables; they do not have a value. Because there is only a single discard variable, and that variable may not even be allocated storage, discards can reduce memory allocations. Because they make the intent of your code clear, they enhance its readability and maintainability.

You indicate that a variable is a discard by assigning it the underscore (`_`) as its name. For example, the following method call returns a 3-tuple in which the first and second values are discards and *area* is a previously declared variable to be set to the corresponding third component returned by *GetCityInformation*:

```
(_, _, area) = city.GetCityInformation(cityName);
```

In C# 7.0, discards are supported in assignments in the following contexts:

- Tuple and object deconstruction.
- Pattern matching with `is` and `switch`.
- Calls to methods with `out` parameters.
- A standalone `_` when no `_` is in scope.

When `_` is a valid discard, attempting to retrieve its value or use it in an assignment operation generates compiler error CS0301, "The name '_' does not exist in the current context". This is because `_` is not assigned a value, and may not even be assigned a storage location. If it were an actual variable, you could not discard more than one value, as the previous example did.

Tuple and object deconstruction

Discards are particularly useful in working with tuples when your application code uses some tuple elements but ignores others. For example, the following `QueryCityDataForYears` method returns a 6-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
       var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

       Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
   }

   private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
   {
      int population1 = 0, population2 = 0;
      double area = 0;

      if (name == "New York City") {
         area = 468.48;
         if (year1 == 1960) {
            population1 = 7781984;
         }
         if (year2 == 2010) {
            population2 = 8175133;
         }
      return (name, area, year1, population1, year2, population2);
      }

      return ("", 0, 0, 0, 0, 0);
   }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

For more information on deconstructing tuples with discards, see Deconstructing tuples and other types.

The `Deconstruct` method of a class, structure, or interface also allows you to retrieve and deconstruct a specific set of data from an object. You can use discards when you are interested in working with only a subset of the deconstructed values. The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state), but discards the last name and the state.

```
using System;

public class Person
{
   public string FirstName { get; set; }
   public string MiddleName { get; set; }
   public string LastName { get; set; }
   public string City { get; set; }
   public string State { get; set; }

   public Person(string fname, string mname, string lname,
                 string cityName, string stateName)
   {
      FirstName = fname;
      MiddleName = mname;
      LastName = lname;
      City = cityName;
      State = stateName;
   }

   // Return the first and last name.
   public void Deconstruct(out string fname, out string lname)
   {
      fname = FirstName;
      lname = LastName;
   }

   public void Deconstruct(out string fname, out string mname, out string lname)
   {
      fname = FirstName;
      mname = MiddleName;
      lname = LastName;
   }

   public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
   {
      fname = FirstName;
      lname = LastName;
      city = City;
      state = State;
   }
}

public class Example
{
   public static void Main()
   {
       Person p = new Person("John", "Quincy", "Adams", "Boston", "MA");

       // <Snippet1>
       // Deconstruct the person object.
       var (fName, _, city, _) = p;
       Console.WriteLine($"Hello {fName} of {city}!");
       // The example displays the following output:
       //      Hello John of Boston!
       // </Snippet1>
   }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

For more information on deconstructing user-defined types with discards, see Deconstructing tuples and other types.

Pattern matching with `switch` and `is`

The *discard pattern* can be used in pattern matching with the is and switch keywords. Every expression always matches the discard pattern.

The following example defines a `ProvidesFormatInfo` method that uses is statements to determine whether an object provides an IFormatProvider implementation and tests whether the object is `null`. It also uses the discard pattern to handle non-null objects of any other type.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      object[] objects = { CultureInfo.CurrentCulture,
                           CultureInfo.CurrentCulture.DateTimeFormat,
                           CultureInfo.CurrentCulture.NumberFormat,
                           new ArgumentException(), null };
      foreach (var obj in objects)
         ProvidesFormatInfo(obj);
   }

   private static void ProvidesFormatInfo(object obj)
   {
      switch (obj)
      {
         case IFormatProvider fmt:
            Console.WriteLine($"{fmt} object");
            break;
         case null:
            Console.Write("A null object reference: ");
            Console.WriteLine("Its use could result in a NullReferenceException");
            break;
         case object _:
            Console.WriteLine("Some object type without format information");
            break;
      }
   }
}
// The example displays the following output:
//    en-US object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
//    Some object type without format information
//    A null object reference: Its use could result in a NullReferenceException
```

Calls to methods with out parameters

When calling the `Deconstruct` method to deconstruct a user-defined type (an instance of a class, structure, or interface), you can discard the values of individual `out` arguments. But you can also discard the value of `out` arguments when calling any method with an out parameter.

The following example calls the DateTime.TryParse(String, out DateTime) method to determine whether the string representation of a date is valid in the current culture. Because the example is concerned only with validating the date string and not with parsing it to extract the date, the `out` argument to the method is a discard.

```
using System;

public class Example
{
   public static void Main()
   {
      string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                              "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                              "5/01/2018 14:57:32.80 -07:00",
                              "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                              "Fri, 15 May 2018 20:10:57 GMT" };
      foreach (string dateString in dateStrings)
      {
         if (DateTime.TryParse(dateString, out _))
            Console.WriteLine($"'{dateString}': valid");
         else
            Console.WriteLine($"'{dateString}': invalid");
      }
   }
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

A standalone discard

You can use a standalone discard to indicate any variable that you choose to ignore. The following example uses a standalone discard to ignore the Task object returned by an asynchronous operation. This has the effect of suppressing the exception that the operation throws as it is about to complete.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      ExecuteAsyncMethods().Wait();
   }

   private static async Task ExecuteAsyncMethods()
   {
      Console.WriteLine("About to launch a task...");
      _ = Task.Run(() => { var iterations = 0;
                           for (int ctr = 0; ctr < int.MaxValue; ctr++)
                              iterations++;
                           Console.WriteLine("Completed looping operation...");
                           throw new InvalidOperationException();
                         });
      await Task.Delay(5000);
      Console.WriteLine("Exiting after 5 second delay");
   }
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Note that `_` is also a valid identifier. When used outside of a supported context, `_` is treated not as a discard but as a valid variable. If an identifier named `_` is already in scope, the use of `_` as a standalone discard can result in:

- Accidental modification of the value of the in-scope `_` variable by assigning it the value of the intended discard. For example:

```
private static void ShowValue(int _)
{
   byte[] arr = { 0, 0, 1, 2 };
   _ = BitConverter.ToInt32(arr, 0);
   Console.WriteLine(_);
}
// The example displays the following output:
//       33619968
```

- A compiler error for violating type safety. For example:

```
private static bool RoundTrips(int _)
{
   string value = _.ToString();
   int newValue = 0;
   _ = Int32.TryParse(value, out newValue);
   return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Compiler error CS0136, "A local or parameter named '_' cannot be declared in this scope because that name is used in an enclosing local scope to define a local or parameter." For example:

```
public void DoSomething(int _)
{
 var _ = GetValue(); // Error: cannot declare local _ when one is already in scope
}
// The example displays the following compiler error:
// error CS0136:
//       A local or parameter named '_' cannot be declared in this scope
//       because that name is used in an enclosing local scope
//       to define a local or parameter
```

See also

- Deconstructing tuples and other types
- `is` keyword
- `switch` keyword

Generics (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. Version 2.0 of the .NET Framework class library provides a new namespace, System.Collections.Generic, which contains several new generic-based collection classes. It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as ArrayList. For more information, see Generics in .NET.

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the List<T> class provided by the .NET Framework class library instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

- As the type of a method parameter in the `AddHead` method.
- As the return type of the `Data` property in the nested `Node` class.
- As the type of the private member `data` in the nested class.

Note that T is available to the nested `Node` class. When `GenericList<T>` is instantiated with a concrete type, for example as a `GenericList<int>`, each occurrence of `T` will be replaced with `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

The following code example shows how client code uses the generic `GenericList<T>` class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Generics Overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related Sections

For more information:

- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics and Reflection
- Generics in the Run Time

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- C# Programming Guide
- Types
- <typeparam>
- <typeparamref>
- Generics in .NET

Iterators

4/16/2019 • 5 minutes to read • Edit Online

Almost every program you write will have some need to iterate over a collection. You'll write code that examines every item in a collection.

You'll also create iterator methods which are methods that produces an iterator for the elements of that class. These can be used for:

- Performing an action on each item in a collection.
- Enumerating a custom collection.
- Extending LINQ or other libraries.
- Creating a data pipeline where data flows efficiently through iterator methods.

The C# language provides features for both these scenarios. This article provides an overview of those features.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also view or download the completed sample for this topic. For download instructions, see Samples and Tutorials.

Iterating with foreach

Enumerating a collection is simple: The `foreach` keyword enumerates a collection, executing the embedded statement once for each element in the collection:

```
foreach (var item in collection)
{
   Console.WriteLine(item.ToString());
}
```

That's all there is to it. To iterate over all the contents of a collection, the `foreach` statement is all you need. The `foreach` statement isn't magic, though. It relies on two generic interfaces defined in the .NET core library in order to generate the code necessary to iterate a collection: `IEnumerable<T>` and `IEnumerator<T>`. This mechanism is explained in more detail below.

Both of these interfaces also have non-generic counterparts: `IEnumerable` and `IEnumerator`. The generic versions are preferred for modern code.

Enumeration sources with iterator methods

Another great feature of the C# language enables you to build methods that create a source for an enumeration. These are referred to as *iterator methods*. An iterator method defines how to generate the objects in a sequence when requested. You use the `yield return` contextual keywords to define an iterator method.

You could write this method to produce the sequence of integers from 0 through 9:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

The code above shows distinct `yield return` statements to highlight the fact that you can use multiple discrete `yield return` statements in an iterator method. You can (and often do) use other language constructs to simplify the code of an iterator method. The method definition below produces the exact same sequence of numbers:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index++ < 10)
        yield return index;
}
```

You don't have to decide one or the other. You can have as many `yield return` statements as necessary to meet the needs of your method:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index++ < 10)
        yield return index;

    yield return 50;

    index = 100;
    while (index++ < 110)
        yield return index;
}
```

That's the basic syntax. Let's consider a real world example where you would write an iterator method. Imagine you're on an IoT project and the device sensors generate a very large stream of data. To get a feel for the data, you might write a method that samples every Nth data element. This small iterator method does the trick:

```
public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

There is one important restriction on iterator methods: you can't have both a `return` statement and a `yield return` statement in the same method. The following will not compile:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index++ < 10)
        yield return index;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

This restriction normally isn't a problem. You have a choice of either using `yield return` throughout the method, or separating the original method into multiple methods, some using `return`, and some using `yield return`.

You can modify the last method slightly to use `yield return` everywhere:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index++ < 10)
        yield return index;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

Sometimes, the right answer is to split an iterator method into two different methods. One that uses `return`, and a second that uses `yield return`. Consider a situation where you might want to return an empty collection, or the first 5 odd numbers, based on a boolean argument. You could write that as these two methods:

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index++ < 10)
        if (index % 2 == 1)
            yield return index;
}
```

Look at the methods above. The first uses the standard `return` statement to return either an empty collection, or the iterator created by the second method. The second method uses the `yield return` statement to create the requested sequence.

Deeper Dive into `foreach`

The `foreach` statement expands into a standard idiom that uses the `IEnumerable<T>` and `IEnumerator<T>` interfaces to iterate across all elements of a collection. It also minimizes errors developers make by not properly

managing resources.

The compiler translates the `foreach` loop shown in the first example into something similar to this construct:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

The construct above represents the code generated by the C# compiler as of version 5 and above. Prior to version 5, the `item` variable had a different scope:

```
// C# versions 1 through 4:
IEnumerator<int> enumerator = collection.GetEnumerator();
int item = default(int);
while (enumerator.MoveNext())
{
    item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

This was changed because the earlier behavior could lead to subtle and hard to diagnose bugs involving lambda expressions. For more information about lambda expressions, see Lambda expressions.

The exact code generated by the compiler is somewhat more complicated, and handles situations where the object returned by `GetEnumerator()` implements the `IDisposable` interface. The full expansion generates code more like this:

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    } finally
    {
        // dispose of enumerator.
    }
}
```

The manner in which the enumerator is disposed of depends on the characteristics of the type of `enumerator`. In the general case, the `finally` clause expands to:

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

However, if the type of `enumerator` is a sealed type and there is no implicit conversion from the type of `enumerator` to `IDisposable`, the `finally` clause expands to an empty block:

```
finally
{
}
```

If there is an implicit conversion from the type of `enumerator` to `IDisposable`, and `enumerator` is a non-nullable value type, the `finally` clause expands to:

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Thankfully, you don't need to remember all these details. The `foreach` statement handles all those nuances for you. The compiler will generate the correct code for any of these constructs.

Introduction to Delegates

5/18/2019 • 2 minutes to read • Edit Online

Delegates provide a *late binding* mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm.

For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.

These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

As you'll see later in this series, the C# code you write for algorithms like this is type safe, and leverages the language and the compiler to ensure that the types match for arguments and return types.

Language Design Goals for Delegates

The language designers enumerated several goals for the feature that eventually became delegates.

The team wanted a common language construct that could be used for any late binding algorithms. That enables developers to learn one concept, and use that same concept across many different software problems.

Second, the team wanted to support both single and multicast method calls. (Multicast delegates are delegates that chain together multiple method calls. You'll see examples later in this series.)

The team wanted delegates to support the same type safety that developers expect from all C# constructs.

Finally, the team recognized that an event pattern is one specific pattern where delegates, or any late binding algorithm, is very useful. The team wanted to ensure that the code for delegates could provide the basis for the .NET event pattern.

The result of all that work was the delegate and event support in C# and .NET. The remaining articles in this section will cover the language features, the library support, and the common idioms that are used when you work with delegates.

You'll learn about the `delegate` keyword and what code it generates. You'll learn about the features in the `System.Delegate` class, and how those features are used. You'll learn how to create type safe delegates, and how to create methods that can be invoked through delegates. You'll also learn how to work with delegates and events by using Lambda expressions. You'll see where delegates become one of the building blocks for LINQ. You'll learn how delegates are the basis for the .NET event pattern, and how they are different.

Overall, you'll see how delegates are an integral part of programming in .NET and working with the framework APIs.

Let's get started.

Next

System.Delegate and the `delegate` keyword

3/26/2019 • 6 minutes to read • Edit Online

Previous

This article will cover the classes in the .NET framework that support delegates, and how those map to the `delegate` keyword.

Defining Delegate Types

Let's start with the 'delegate' keyword, because that's primarily what you will use as you work with delegates. The code that the compiler generates when you use the `delegate` keyword will map to method calls that invoke members of the Delegate and MulticastDelegate classes.

You define a delegate type using syntax that is similar to defining a method signature. You just add the `delegate` keyword to the definition.

Let's continue to use the List.Sort() method as our example. The first step is to create a type for the comparison delegate:

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

The compiler generates a class, derived from `System.Delegate` that matches the signature used (in this case, a method that returns an integer, and has two arguments). The type of that delegate is `Comparison`. The `Comparison` delegate type is a generic type. For details on generics see here.

Notice that the syntax may appear as though it is declaring a variable, but it is actually declaring a *type*. You can define delegate types inside classes, directly inside namespaces, or even in the global namespace.

> **NOTE**
> Declaring delegate types (or other types) directly in the global namespace is not recommended.

The compiler also generates add and remove handlers for this new type so that clients of this class can add and remove methods from an instance's invocation list. The compiler will enforce that the signature of the method being added or removed matches the signature used when declaring the method.

Declaring instances of delegates

After defining the delegate, you can create an instance of that type. Like all variables in C#, you cannot declare delegate instances directly in a namespace, or in the global namespace.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

The type of the variable is `Comparison<T>`, the delegate type defined earlier. The name of the variable is

`comparator`.

That code snippet above declared a member variable inside a class. You can also declare delegate variables that are local variables, or arguments to methods.

Invoking Delegates

You invoke the methods that are in the invocation list of a delegate by calling that delegate. Inside the `Sort()` method, the code will call the comparison method to determine which order to place objects:

```
int result = comparator(left, right);
```

In the line above, the code *invokes* the method attached to the delegate. You treat the variable as a method name, and invoke it using normal method call syntax.

That line of code makes an unsafe assumption: There's no guarantee that a target has been added to the delegate. If no targets have been attached, the line above would cause a `NullReferenceException` to be thrown. The idioms used to address this problem are more complicated than a simple null-check, and are covered later in this series.

Assigning, Adding and removing Invocation Targets

That's how a delegate type is defined, and how delegate instances are declared and invoked.

Developers that want to use the `List.Sort()` method need to define a method whose signature matches the delegate type definition, and assign it to the delegate used by the sort method. This assignment adds the method to the invocation list of that delegate object.

Suppose you wanted to sort a list of strings by their length. Your comparison function might be the following:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

The method is declared as a private method. That's fine. You may not want this method to be part of your public interface. It can still be used as the comparison method when attached to a delegate. The calling code will have this method attached to the target list of the delegate object, and can access it through that delegate.

You create that relationship by passing that method to the `List.Sort()` method:

```
phrases.Sort(CompareLength);
```

Notice that the method name is used, without parentheses. Using the method as an argument tells the compiler to convert the method reference into a reference that can be used as a delegate invocation target, and attach that method as an invocation target.

You could also have been explicit by declaring a variable of type `Comparison<string>` and doing an assignment:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

In uses where the method being used as a delegate target is a small method, it's common to use lambda expression syntax to perform the assignment:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Using lambda expressions for delegate targets is covered more in a later section.

The Sort() example typically attaches a single target method to the delegate. However, delegate objects do support invocation lists that have multiple target methods attached to a delegate object.

Delegate and MulticastDelegate classes

The language support described above provides the features and support you'll typically need to work with delegates. These features are built on two classes in the .NET Core framework: Delegate and MulticastDelegate.

The `System.Delegate` class, and its single direct sub-class, `System.MulticastDelegate`, provide the framework support for creating delegates, registering methods as delegate targets, and invoking all methods that are registered as a delegate target.

Interestingly, the `System.Delegate` and `System.MulticastDelegate` classes are not themselves delegate types. They do provide the basis for all specific delegate types. That same language design process mandated that you cannot declare a class that derives from `Delegate` or `MulticastDelegate`. The C# language rules prohibit it.

Instead, the C# compiler creates instances of a class derived from `MulticastDelegate` when you use the C# language keyword to declare delegate types.

This design has its roots in the first release of C# and .NET. One goal for the design team was to ensure that the language enforced type safety when using delegates. That meant ensuring that delegates were invoked with the right type and number of arguments. And, that any return type was correctly indicated at compile time. Delegates were part of the 1.0 .NET release, which was before generics.

The best way to enforce this type safety was for the compiler to create the concrete delegate classes that represented the method signature being used.

Even though you cannot create derived classes directly, you will use the methods defined on these classes. Let's go through the most common methods that you will use when you work with delegates.

The first, most important fact to remember is that every delegate you work with is derived from `MulticastDelegate`. A multicast delegate means that more than one method target can be invoked when invoking through a delegate. The original design considered making a distinction between delegates where only one target method could be attached and invoked, and delegates where multiple target methods could be attached and invoked. That distinction proved to be less useful in practice than originally thought. The two different classes were already created, and have been in the framework since its initial public release.

The methods that you will use the most with delegates are `Invoke()` and `BeginInvoke()` / `EndInvoke()`. `Invoke()` will invoke all the methods that have been attached to a particular delegate instance. As you saw above, you typically invoke delegates using the method call syntax on the delegate variable. As you'll see later in this series, there are patterns that work directly with these methods.

Now that you've seen the language syntax and the classes that support delegates, let's examine how strongly typed delegates are used, created and invoked.

Next

Strongly Typed Delegates

5/4/2018 • 2 minutes to read • Edit Online

Previous

In the previous article, you saw that you create specific delegate types using the `delegate` keyword.

The abstract Delegate class provide the infrastructure for loose coupling and invocation. Concrete Delegate types become much more useful by embracing and enforcing type safety for the methods that are added to the invocation list for a delegate object. When you use the `delegate` keyword and define a concrete delegate type, the compiler generates those methods.

In practice, this would lead to creating new delegate types whenever you need a different method signature. This work could get tedious after a time. Every new feature requires new delegate types.

Thankfully, this isn't necessary. The .NET Core framework contains several types that you can reuse whenever you need delegate types. These are generic definitions so you can declare customizations when you need new method declarations.

The first of these types is the Action type, and several variations:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

The `in` modifier on the generic type argument is covered in the article on covariance.

There are variations of the `Action` delegate that contain up to 16 arguments such as Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. It's important that these definitions use different generic arguments for each of the delegate arguments: That gives you maximum flexibility. The method arguments need not be, but may be, the same type.

Use one of the `Action` types for any delegate type that has a void return type.

The framework also includes several generic delegate types that you can use for delegate types that return values:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

The `out` modifier on the result generic type argument is covered in the article on covariance.

There are variations of the `Func` delegate with up to 16 input arguments such as Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. The type of the result is always the last type parameter in all the `Func` declarations, by convention.

Use one of the `Func` types for any delegate type that returns a value.

There's also a specialized Predicate<T> type for a delegate that returns a test on a single value:

```
public delegate bool Predicate<in T>(T obj);
```

You may notice that for any `Predicate` type, a structurally equivalent `Func` type exists For example:

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

You might think these two types are equivalent. They are not. These two variables cannot be used interchangeably. A variable of one type cannot be assigned the other type. The C# type system uses the names of the defined types, not the structure.

All these delegate type definitions in the .NET Core Library should mean that you do not need to define a new delegate type for any new feature you create that requires delegates. These generic definitions should provide all the delegate types you need under most situations. You can simply instantiate one of these types with the required type parameters. In the case of algorithms that can be made generic, these delegates can be used as generic types.

This should save time, and minimize the number of new types that you need to create in order to work with delegates.

In the next article, you'll see several common patterns for working with delegates in practice.

Next

Common Patterns for Delegates

4/8/2019 • 8 minutes to read • Edit Online

Previous

Delegates provide a mechanism that enables software designs involving minimal coupling between components.

One excellent example for this kind of design is LINQ. The LINQ Query Expression Pattern relies on delegates for all of its features. Consider this simple example:

```
var smallNumbers = numbers.Where(n => n < 10);
```

This filters the sequence of numbers to only those less than the value 10. The `Where` method uses a delegate that determines which elements of a sequence pass the filter. When you create a LINQ query, you supply the implementation of the delegate for this specific purpose.

The prototype for the Where method is:

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool>
predicate);
```

This example is repeated with all the methods that are part of LINQ. They all rely on delegates for the code that manages the specific query. This API design pattern is a very powerful one to learn and understand.

This simple example illustrates how delegates require very little coupling between components. You don't need to create a class that derives from a particular base class. You don't need to implement a specific interface. The only requirement is to provide the implementation of one method that is fundamental to the task at hand.

Building Your Own Components with Delegates

Let's build on that example by creating a component using a design that relies on delegates.

Let's define a component that could be used for log messages in a large system. The library components could be used in many different environments, on multiple different platforms. There are a lot of common features in the component that manages the logs. It will need to accept messages from any component in the system. Those messages will have different priorities, which the core component can manage. The messages should have timestamps in their final archived form. For more advanced scenarios, you could filter messages by the source component.

There is one aspect of the feature that will change often: where messages are written. In some environments, they may be written to the error console. In others, a file. Other possibilities include database storage, OS event logs, or other document storage.

There are also combinations of output that might be used in different scenarios. You may want to write messages to the console and to a file.

A design based on delegates will provide a great deal of flexibility, and make it easy to support storage mechanisms that may be added in the future.

Under this design, the primary log component can be a non-virtual, even sealed class. You can plug in any set of delegates to write the messages to different storage media. The built in support for multicast delegates makes it easy to support scenarios where messages must be written to multiple locations (a file, and a console).

A First Implementation

Let's start small: the initial implementation will accept new messages, and write them using any attached delegate. You can start with one delegate that writes messages to the console.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

The static class above is the simplest thing that can work. We need to write the single implementation for the method that writes messages to the console:

```
public static void LogToConsole(string message)
{
    Console.Error.WriteLine(message);
}
```

Finally, you need to hook up the delegate by attaching it to the WriteMessage delegate declared in the logger:

```
Logger.WriteMessage += LogToConsole;
```

Practices

Our sample so far is fairly simple, but it still demonstrates some of the important guidelines for designs involving delegates.

Using the delegate types defined in the Core Framework makes it easier for users to work with the delegates. You don't need to define new types, and developers using your library do not need to learn new, specialized delegate types.

The interfaces used are as minimal and as flexible as possible: To create a new output logger, you must create one method. That method may be a static method, or an instance method. It may have any access.

Formatting Output

Let's make this first version a bit more robust, and then start creating other logging mechanisms.

Next, let's add a few arguments to the `LogMessage()` method so that your log class creates more structured messages:

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Next, let's make use of that `Severity` argument to filter the messages that are sent to the log's output.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Practices

You've added new features to the logging infrastructure. Because the logger component is very loosely coupled to any output mechanism, these new features can be added with no impact on any of the code implementing the logger delegate.

As you keep building this, you'll see more examples of how this loose coupling enables greater flexibility in updating parts of the site without any changes to other locations. In fact, in a larger application, the logger output classes might be in a different assembly, and not even need to be rebuilt.

Building a Second Output Engine

The Log component is coming along well. Let's add one more output engine that logs messages to a file. This will be a slightly more involved output engine. It will be a class that encapsulates the file operations, and ensures that the file is always closed after each write. That ensures that all the data is flushed to disk after each message is generated.

Here is that file based logger:

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Once you've created this class, you can instantiate it and it attaches its LogMessage method to the Logger component:

```
var file = new FileLogger("log.txt");
```

These two are not mutually exclusive. You could attach both log methods and generate messages to the console and a file:

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LogToConsole;
```

Later, even in the same application, you can remove one of the delegates without any other issues to the system:

```
Logger.WriteMessage -= LogToConsole;
```

Practices

Now, you've added a second output handler for the logging sub-system. This one needs a bit more infrastructure to correctly support the file system. The delegate is an instance method. It's also a private method. There's no need for greater accessibility because the delegate infrastructure can connect the delegates.

Second, the delegate-based design enables multiple output methods without any extra code. You don't need to build any additional infrastructure to support multiple output methods. They simply become another method on the invocation list.

Pay special attention to the code in the file logging output method. It is coded to ensure that it does not throw any exceptions. While this isn't always strictly necessary, it's often a good practice. If either of the delegate methods throws an exception, the remaining delegates that are on the invocation won't be invoked.

As a last note, the file logger must manage its resources by opening and closing the file on each log message. You could choose to keep the file open and implement IDisposable to close the file when you are completed. Either method has its advantages and disadvantages. Both do create a bit more coupling between the classes.

None of the code in the Logger class would need to be updated in order to support either scenario.

Handling Null Delegates

Finally, let's update the LogMessage method so that it is robust for those cases when no output mechanism is selected. The current implementation will throw a `NullReferenceException` when the `WriteMessage` delegate does not have an invocation list attached. You may prefer a design that silently continues when no methods have been attached. This is easy using the null conditional operator, combined with the `Delegate.Invoke()` method:

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

The null conditional operator (`?.`) short-circuits when the left operand (`WriteMessage` in this case) is null, which means no attempt is made to log a message.

You won't find the `Invoke()` method listed in the documentation for `System.Delegate` or `System.MulticastDelegate`. The compiler generates a type safe `Invoke` method for any delegate type declared. In this example, that means `Invoke` takes a single `string` argument, and has a void return type.

Summary of Practices

You've seen the beginnings of a log component that could be expanded with other writers, and other features. By using delegates in the design these different components are very loosely coupled. This provides several advantages. It's very easy to create new output mechanisms and attach them to the system. These other mechanisms only need one method: the method that writes the log message. It's a design that is very resilient when new features are added. The contract required for any writer is to implement one method. That method could be a static or instance method. It could be public, private, or any other legal access.

The Logger class can make any number of enhancements or changes without introducing breaking changes. Like any class, you cannot modify the public API without the risk of breaking changes. But, because the coupling between the logger and any output engines is only through the delegate, no other types (like interfaces or base classes) are involved. The coupling is as small as possible.

Next

Introduction to Events

5/23/2019 • 3 minutes to read • Edit Online

Previous

Events are, like delegates, a *late binding* mechanism. In fact, events are built on the language support for delegates.

Events are a way for an object to broadcast (to all interested components in the system) that something has happened. Any other component can subscribe to the event, and be notified when an event is raised.

You've probably used events in some of your programming. Many graphical systems have an event model to report user interaction. These events would report mouse movement, button presses and similar interactions. That's one of the most common, but certainly not the only scenario where events are used.

You can define events that should be raised for your classes. One important consideration when working with events is that there may not be any object registered for a particular event. You must write your code so that it does not raise events when no listeners are configured.

Subscribing to an event also creates a coupling between two objects (the event source, and the event sink). You need to ensure that the event sink unsubscribes from the event source when no longer interested in events.

Design Goals for Event Support

The language design for events targets these goals.

First, enable very minimal coupling between an event source and an event sink. These two components may not be written by the same organization, and may even be updated on totally different schedules.

Secondly, it should be very simple to subscribe to an event, and to unsubscribe from that same event.

And finally, event sources should support multiple event subscribers. It should also support having no event subscribers attached.

You can see that the goals for events are very similar to the goals for delegates. That's why the event language support is built on the delegate language support.

Language Support for Events

The syntax for defining events, and subscribing or unsubscribing from events is an extension of the syntax for delegates.

To define an event you use the `event` keyword:

```
public event EventHandler<FileListArgs> Progress;
```

The type of the event (`EventHandler<FileListArgs>` in this example) must be a delegate type. There are a number of conventions that you should follow when declaring an event. Typically, the event delegate type has a void return. Event declarations should be a verb, or a verb phrase. Use past tense (as in this example) when the event reports something that has happened. Use a present tense verb (for example, `Closing`) to report something that is about to happen. Often, using present tense indicates that your class supports some kind of customization behavior. One of the most common scenarios is to support cancellation. For example, a `Closing` event may include an argument that would indicate if the close operation should continue, or not. Other scenarios may

enable callers to modify behavior by updating properties of the event arguments. You may raise an event to indicate a proposed next action an algorithm will take. The event handler may mandate a different action by modifying properties of the event argument.

When you want to raise the event, you call the event handlers using the delegate invocation syntax:

```
Progress?.Invoke(this, new FileListArgs(file));
```

As discussed in the section on delegates, the ?. operator makes it easy to ensure that you do not attempt to raise the event when there are no subscribers to that event.

You subscribe to an event by using the `+=` operator:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

The handler method typically is the prefix 'On' followed by the event name, as shown above.

You unsubscribe using the `-=` operator:

```
fileLister.Progress -= onProgress;
```

It's important to note that I declared a local variable for the expression that represents the event handler. That ensures the unsubscribe removes the handler. If, instead, you used the body of the lambda expression, you are attempting to remove a handler that has never been attached, which does nothing.

In the next article, you'll learn more about typical event patterns, and different variations on this example.

Next

Standard .NET event patterns

5/6/2019 • 8 minutes to read • Edit Online

Previous

.NET events generally follow a few known patterns. Standardizing on these patterns means that developers can leverage knowledge of those standard patterns, which can be applied to any .NET event program.

Let's go through these standard patterns so you will have all the knowledge you need to create standard event sources, and subscribe and process standard events in your code.

Event Delegate Signatures

The standard signature for a .NET event delegate is:

```
void OnEventRaised(object sender, EventArgs args);
```

The return type is void. Events are based on delegates and are multicast delegates. That supports multiple subscribers for any event source. The single return value from a method doesn't scale to multiple event subscribers. Which return value does the event source see after raising an event? Later in this article you'll see how to create event protocols that support event subscribers that report information to the event source.

The argument list contains two arguments: the sender, and the event arguments. The compile time type of `sender` is `System.Object`, even though you likely know a more derived type that would always be correct. By convention, use `object`.

The second argument has typically been a type that is derived from `System.EventArgs`. (You'll see in the next section that this convention is no longer enforced.) If your event type does not need any additional arguments, you will still provide both arguments. There is a special value, `EventArgs.Empty` that you should use to denote that your event does not contain any additional information.

Let's build a class that lists files in a directory, or any of its subdirectories that follow a pattern. This component raises an event for each file found that matches the pattern.

Using an event model provides some design advantages. You can create multiple event listeners that perform different actions when a sought file is found. Combining the different listeners can create more robust algorithms.

Here is the initial event argument declaration for finding a sought file:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Even though this type looks like a small, data-only type, you should follow the convention and make it a reference (`class`) type. That means the argument object will be passed by reference, and any updates to the data will be viewed by all subscribers. The first version is an immutable object. You should prefer to make the properties in your event argument type immutable. That way, one subscriber cannot change the values before another

subscriber sees them. (There are exceptions to this, as you'll see below.)

Next, we need to create the event declaration in the FileSearcher class. Leveraging the `EventHandler<T>` type means that you don't need to create yet another type definition. You simply use a generic specialization.

Let's fill out the FileSearcher class to search for files that match a pattern, and raise the correct event when a match is discovered.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Defining and Raising Field-Like Events

The simplest way to add an event to your class is to declare that event as a public field, as in the preceding example:

```
public event EventHandler<FileFoundArgs> FileFound;
```

This looks like it's declaring a public field, which would appear to be bad object-oriented practice. You want to protect data access through properties, or methods. While this may look like a bad practice, the code generated by the compiler does create wrappers so that the event objects can only be accessed in safe ways. The only operations available on a field-like event are add handler:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

and remove handler:

```
fileLister.FileFound -= onFileFound;
```

Note that there's a local variable for the handler. If you used the body of the lambda, the remove would not work correctly. It would be a different instance of the delegate, and silently do nothing.

Code outside the class cannot raise the event, nor can it perform any other operations.

Returning Values from Event Subscribers

Your simple version is working fine. Let's add another feature: Cancellation.

When you raise the found event, listeners should be able to stop further processing, if this file is that last one sought.

The event handlers do not return a value, so you need to communicate that in another way. The standard event pattern uses the EventArgs object to include fields that event subscribers can use to communicate cancel.

There are two different patterns that could be used, based on the semantics of the Cancel contract. In both cases, you'll add a boolean field to the EventArguments for the found file event.

One pattern would allow any one subscriber to cancel the operation. For this pattern, the new field is initialized to `false`. Any subscriber can change it to `true`. After all subscribers have seen the event raised, the FileSearcher component examines the boolean value and takes action.

The second pattern would only cancel the operation if all subscribers wanted the operation cancelled. In this pattern, the new field is initialized to indicate the operation should cancel, and any subscriber could change it to indicate the operation should continue. After all subscribers have seen the event raised, the FileSearcher component examines the boolean and takes action. There is one extra step in this pattern: the component needs to know if any subscribers have seen the event. If there are no subscribers, the field would indicate a cancel incorrectly.

Let's implement the first version for this sample. You need to add a boolean field named `CancelRequested` to the `FileFoundArgs` type:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set;}

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

This new Field is automatically initialized to `false`, the default value for a Boolean field, so you don't cancel accidentally. The only other change to the component is to check the flag after raising the event to see if any of the subscribers have requested a cancellation:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

One advantage of this pattern is that it isn't a breaking change. None of the subscribers requested a cancellation before, and they still are not. None of the subscriber code needs updating unless they want to support the new cancel protocol. It's very loosely coupled.

Let's update the subscriber so that it requests a cancellation once it finds the first executable:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Adding Another Event Declaration

Let's add one more feature, and demonstrate other language idioms for events. Let's add an overload of the `Search` method that traverses all subdirectories in search of files.

This could get to be a lengthy operation in a directory with many sub-directories. Let's add an event that gets raised when each new directory search begins. This enables subscribers to track progress, and update the user as to progress. All the samples you've created so far are public. Let's make this one an internal event. That means you can also make the types used for the arguments internal as well.

You'll start by creating the new EventArgs derived class for reporting the new directory and progress.

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Again, you can follow the recommendations to make an immutable reference type for the event arguments.

Next, define the event. This time, you'll use a different syntax. In addition to using the field syntax, you can explicitly create the property, with add and remove handlers. In this sample, you won't need extra code in those handlers, but this shows how you would create them.

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;
```

In many ways, the code you write here mirrors the code the compiler generates for the field event definitions you've seen earlier. You create the event using syntax very similar to that used for properties. Notice that the handlers have different names: `add` and `remove`. These are called to subscribe to the event, or unsubscribe from the event. Notice that you also must declare a private backing field to store the event variable. It is initialized to null.

Next, let's add the overload of the `Search` method that traverses subdirectories and raises both events. The easiest way to accomplish this is to use a default argument to specify that you want to search all directories:

```
public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

At this point, you can run the application calling the overload for searching all sub-directories. There are no subscribers on the new `ChangeDirectory` event, but using the `?.Invoke()` idiom ensures that this works correctly.

Let's add a handler to write a line that shows the progress in the console window.

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};
```

You've seen patterns that are followed throughout the .NET ecosystem. By learning these patterns and conventions, you'll be writing idiomatic C# and .NET quickly.

Next, you'll see some changes in these patterns in the most recent release of .NET.

Next

The Updated .NET Core Event Pattern

4/25/2019 • 3 minutes to read • Edit Online

Previous

The previous article discussed the most common event patterns. .NET Core has a more relaxed pattern. In this version, the `EventHandler<TEventArgs>` definition no longer has the constraint that `TEventArgs` must be a class derived from `System.EventArgs`.

This increases flexibility for you, and is backwards compatible. Let's start with the flexibility. The class System.EventArgs introduces one method: `MemberwiseClone()`, which creates a shallow copy of the object. That method must use reflection in order to implement its functionality for any class derived from `EventArgs`. That functionality is easier to create in a specific derived class. That effectively means that deriving from System.EventArgs is a constraint that limits your designs, but does not provide any additional benefit. In fact, you can change the definitions of `FileFoundArgs` and `SearchDirectoryArgs` so that they do not derive from `EventArgs`. The program will work exactly the same.

You could also change the `SearchDirectoryArgs` to a struct, if you make one more change:

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

The additional change is to call the parameterless constructor before entering the constructor that initializes all the fields. Without that addition, the rules of C# would report that the properties are being accessed before they have been assigned.

You should not change the `FileFoundArgs` from a class (reference type) to a struct (value type). That's because the protocol for handling cancel requires that the event arguments are passed by reference. If you made the same change, the file search class could never observe any changes made by any of the event subscribers. A new copy of the structure would be used for each subscriber, and that copy would be a different copy than the one seen by the file search object.

Next, let's consider how this change can be backwards compatible. The removal of the constraint does not affect any existing code. Any existing event argument types do still derive from `System.EventArgs`. Backwards compatibility is one major reason why they will continue to derive from `System.EventArgs`. Any existing event subscribers will be subscribers to an event that followed the classic pattern.

Following similar logic, any event argument type created now would not have any subscribers in any existing codebases. New event types that do not derive from `System.EventArgs` will not break those codebases.

Events with Async subscribers

You have one final pattern to learn: How to correctly write event subscribers that call async code. The challenge is described in the article on async and await. Async methods can have a void return type, but that is strongly discouraged. When your event subscriber code calls an async method, you have no choice but to create an `async void` method. The event handler signature requires it.

You need to reconcile this opposing guidance. Somehow, you must create a safe `async void` method. The basics of the pattern you need to implement are below:

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

First, notice that the handler is marked as an async handler. Because it is being assigned to an event handler delegate type, it will have a void return type. That means you must follow the pattern shown in the handler, and not allow any exceptions to be thrown out of the context of the async handler. Because it does not return a task, there is no task that can report the error by entering the faulted state. Because the method is async, the method can't simply throw the exception. (The calling method has continued execution because it is `async`.) The actual runtime behavior will be defined differently for different environments. It may terminate the thread, it may terminate the program, or it may leave the program in an undetermined state. None of those are good outcomes.

That's why you should wrap the await statement for the async Task in your own try block. If it does cause a faulted task, you can log the error. If it is an error from which your application cannot recover, you can exit the program quickly and gracefully

Those are the major updates to the .NET event pattern. You will see many examples of the earlier versions in the libraries you work with. However, you should understand what the latest patterns are as well.

The next article in this series helps you distinguish between using `delegates` and `events` in your designs. They are similar concepts, and that article will help you make the best decision for your programs.

Next

Distinguishing Delegates and Events

5/4/2018 • 2 minutes to read • Edit Online

Previous

Developers that are new to the .NET Core platform often struggle when deciding between a design based on `delegates` and a design based on `events`. This is a difficult concept, because the two language features are very similar. Events are even built using the language support for delegates.

They both offer a late binding scenario: they enable scenarios where a component communicates by calling a method that is only known at runtime. They both support single and multiple subscriber methods. You may find this referred to as singlecast and multicast support. They both support similar syntax for adding and removing handlers. Finally, raising an event and calling a delegate use exactly the same method call syntax. They even both support the same `Invoke()` method syntax for use with the `?.` operator.

With all those similarities, it is easy to have trouble determining when to use which.

Listening to Events is Optional

The most important consideration in determining which language feature to use is whether or not there must be an attached subscriber. If your code must call the code supplied by the subscriber, you should use a design based on delegates. If your code can complete all its work without calling any subscribers, you should use a design based on events.

Consider the examples built during this section. The code you built using `List.Sort()` must be given a comparer function in order to properly sort the elements. LINQ queries must be supplied with delegates in order to determine what elements to return. Both used a design built with delegates.

Consider the `Progress` event. It reports progress on a task. The task continues to proceed whether or not there are any listeners. The `FileSearcher` is another example. It would still search and find all the files that were sought, even with no event subscribers attached. UX controls still work correctly, even when there are no subscribers listening to the events. They both use designs based on events.

Return Values Require Delegates

Another consideration is the method prototype you would want for your delegate method. As you've seen, the delegates used for events all have a void return type. You've also seen that there are idioms to create event handlers that do pass information back to event sources through modifying properties of the event argument object. While these idioms do work, they are not as natural as returning a value from a method.

Notice that these two heuristics may often both be present: If your delegate method returns a value, it will likely impact the algorithm in some way.

Event Listeners Often Have Longer Lifetimes

This is a slightly weaker justification. However, you may find that event-based designs are more natural when the event source will be raising events over a long period of time. You can see examples of this for UX controls on many systems. Once you subscribe to an event, the event source may raise events throughout the lifetime of the program. (You can unsubscribe from events when you no longer need them.)

Contrast that with many delegate-based designs, where a delegate is used as an argument to a method, and the delegate is not used after that method returns.

Evaluate Carefully

The above considerations are not hard and fast rules. Instead, they represent guidance that can help you decide which choice is best for your particular usage. Because they are similar, you can even prototype both, and consider which would be more natural to work with. They both handle late binding scenarios well. Use the one that communicates your design the best.

Language Integrated Query (LINQ)

6/4/2019 • 3 minutes to read • Edit Online

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see Type relationships in LINQ query operations.
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see Introduction to LINQ queries.

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see C# language specification and Standard query operators overview .
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see Query syntax and method syntax in LINQ.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in Query expression basics, and then read the documentation for the LINQ technology in which you are interested:

- XML documents: LINQ to XML
- ADO.NET Entity Framework: LINQ to entities
- .NET collections, files, strings and so on: LINQ to objects

To gain a deeper understanding of LINQ in general, see LINQ in C#.

To start working with LINQ in C#, see the tutorial Working with LINQ.

Asynchronous programming

5/30/2019 • 10 minutes to read • Edit Online

If you have any I/O-bound needs (such as requesting data from a network or accessing a database), you'll want to utilize asynchronous programming. You could also have CPU-bound code, such as performing an expensive calculation, which is also a good scenario for writing async code.

C# has a language-level asynchronous programming model which allows for easily writing asynchronous code without having to juggle callbacks or conform to a library which supports asynchrony. It follows what is known as the Task-based Asynchronous Pattern (TAP).

Basic Overview of the Asynchronous Model

The core of async programming is the `Task` and `Task<T>` objects, which model asynchronous operations. They are supported by the `async` and `await` keywords. The model is fairly simple in most cases:

For I/O-bound code, you `await` an operation which returns a `Task` or `Task<T>` inside of an `async` method.

For CPU-bound code, you `await` an operation which is started on a background thread with the `Task.Run` method.

The `await` keyword is where the magic happens. It yields control to the caller of the method that performed `await`, and it ultimately allows a UI to be responsive or a service to be elastic.

There are other ways to approach async code than `async` and `await` outlined in the TAP article linked above, but this document will focus on the language-level constructs from this point forward.

I/O-Bound Example: Downloading data from a web service

You may need to download some data from a web service when a button is pressed, but don't want to block the UI thread. It can be accomplished simply like this:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

And that's it! The code expresses the intent (downloading some data asynchronously) without getting bogged down in interacting with Task objects.

CPU-bound Example: Performing a Calculation for a Game

Say you're writing a mobile game where pressing a button can inflict damage on many enemies on the screen. Performing the damage calculation can be expensive, and doing it on the UI thread would make the game appear to pause as the calculation is performed!

The best way to handle this is to start a background thread which does the work using `Task.Run`, and `await` its result. This will allow the UI to feel smooth as the work is being done.

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work.  The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

And that's it! This code cleanly expresses the intent of the button's click event, it doesn't require managing a background thread manually, and it does so in a non-blocking way.

What happens under the covers

There's a lot of moving pieces where asynchronous operations are concerned. If you're curious about what's happening underneath the covers of `Task` and `Task<T>`, checkout the Async in-depth article for more information.

On the C# side of things, the compiler transforms your code into a state machine which keeps track of things like yielding execution when an `await` is reached and resuming execution when a background job has finished.

For the theoretically-inclined, this is an implementation of the Promise Model of asynchrony.

Key Pieces to Understand

- Async code can be used for both I/O-bound and CPU-bound code, but differently for each scenario.
- Async code uses `Task<T>` and `Task`, which are constructs used to model work being done in the background.
- The `async` keyword turns a method into an async method, which allows you to use the `await` keyword in its body.
- When the `await` keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
- `await` can only be used inside an async method.

Recognize CPU-Bound and I/O-Bound Work

The first two examples of this guide showed how you can use `async` and `await` for I/O-bound and CPU-bound work. It's key that you can identify when a job you need to do is I/O-bound or CPU-bound, because it can greatly affect the performance of your code and could potentially lead to misusing certain constructs.

Here are two questions you should ask before you write any code:

1. Will your code be "waiting" for something, such as data from a database?

   If your answer is "yes", then your work is **I/O-bound**.

2. Will your code be performing a very expensive computation?

   If you answered "yes", then your work is **CPU-bound**.

If the work you have is **I/O-bound**, use `async` and `await` *without* `Task.Run`. You *should not* use the Task Parallel Library. The reason for this is outlined in the Async in Depth article.

If the work you have is **CPU-bound** and you care about responsiveness, use `async` and `await` but spawn the work off on another thread *with* `Task.Run`. If the work is appropriate for concurrency and parallelism, you should also consider using the Task Parallel Library.

Additionally, you should always measure the execution of your code. For example, you may find yourself in a situation where your CPU-bound work is not costly enough compared with the overhead of context switches when multithreading. Every choice has its tradeoff, and you should pick the correct tradeoff for your situation.

More Examples

The following examples demonstrate various ways you can write async code in C#. They cover a few different scenarios you may come across.

Extracting Data from a Network

This snippet downloads the HTML from the homepage at www.dotnetfoundation.org and counts the number of times the string ".NET" occurs in the HTML. It uses ASP.NET MVC to define a web controller method which performs this task, returning the number.

> **NOTE**
> If you plan on doing HTML parsing in production code, don't use regular expressions. Use a parsing library instead.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet]
[Route("DotNetCount")]
public async Task<int> GetDotNetCountAsync()
{
    // Suspends GetDotNetCountAsync() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Here's the same scenario written for a Universal Windows App, which performs the same task when a Button is pressed:

```
private readonly HttpClient _httpClient = new HttpClient();

private async void SeeTheDotNets_Click(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://www.dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends SeeTheDotNets_Click, returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Waiting for Multiple Tasks to Complete

You may find yourself in a situation where you need to retrieve multiple pieces of data concurrently. The `Task` API contains two methods, `Task.WhenAll` and `Task.WhenAny` which allow you to write asynchronous code which performs a non-blocking wait on multiple background jobs.

This example shows how you might grab `User` data for a set of `userId`s.

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();

    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Here's another way to write this a bit more succinctly, using LINQ:

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

Although it's less code, take care when mixing LINQ with asynchronous code. Because LINQ uses deferred (lazy) execution, async calls won't happen immediately as they do in a `foreach()` loop unless you force the generated sequence to iterate with a call to `.ToList()` or `.ToArray()`.

Important Info and Advice

Although async programming is relatively straightforward, there are some details to keep in mind which can prevent unexpected behavior.

- `async` **methods need to have an** `await` **keyword in their body or they will never yield!**

This is important to keep in mind. If `await` is not used in the body of an `async` method, the C# compiler will generate a warning, but the code will compile and run as if it were a normal method. Note that this would also be incredibly inefficient, as the state machine generated by the C# compiler for the async method would not be accomplishing anything.

- **You should add "Async" as the suffix of every async method name you write.**

This is the convention used in .NET to more-easily differentiate synchronous and asynchronous methods. Note that certain methods which aren't explicitly called by your code (such as event handlers or web controller methods) don't necessarily apply. Because these are not explicitly called by your code, being explicit about their naming isn't as important.

- `async void` **should only be used for event handlers.**

`async void` is the only way to allow asynchronous event handlers to work because events do not have return types (thus cannot make use of `Task` and `Task<T>`). Any other use of `async void` does not follow the TAP model and can be challenging to use, such as:

- Exceptions thrown in an `async void` method can't be caught outside of that method.
- `async void` methods are very difficult to test.
- `async void` methods can cause bad side effects if the caller isn't expecting them to be async.
- **Tread carefully when using async lambdas in LINQ expressions**

Lambda expressions in LINQ use deferred execution, meaning code could end up executing at a time when you're not expecting it to. The introduction of blocking tasks into this can easily result in a deadlock if not written correctly. Additionally, the nesting of asynchronous code like this can also make it more difficult to reason about the execution of the code. Async and LINQ are powerful, but should be used together as carefully and clearly as possible.

- **Write code that awaits Tasks in a non-blocking manner**

Blocking the current thread as a means to wait for a Task to complete can result in deadlocks and blocked context threads, and can require significantly more complex error-handling. The following table provides guidance on how to deal with waiting for Tasks in a non-blocking way:

USE THIS...	INSTEAD OF THIS...	WHEN WISHING TO DO THIS
`await`	`Task.Wait` or `Task.Result`	Retrieving the result of a background task
`await Task.WhenAny`	`Task.WaitAny`	Waiting for any task to complete
`await Task.WhenAll`	`Task.WaitAll`	Waiting for all tasks to complete
`await Task.Delay`	`Thread.Sleep`	Waiting for a period of time

- **Write less stateful code**

Don't depend on the state of global objects or the execution of certain methods. Instead, depend only on the return values of methods. Why?

- Code will be easier to reason about.
- Code will be easier to test.
- Mixing async and synchronous code is far simpler.
- Race conditions can typically be avoided altogether.
- Depending on return values makes coordinating async code simple.
- (Bonus) it works really well with dependency injection.

A recommended goal is to achieve complete or near-complete Referential Transparency in your code. Doing so will result in an extremely predictable, testable, and maintainable codebase.

Other Resources

- Async in-depth provides more information about how Tasks work.
- Asynchronous programming with async and await (C#)
- Lucian Wischik's Six Essential Tips for Async are a wonderful resource for async programming

Pattern Matching

4/11/2019 • 13 minutes to read • Edit Online

Patterns test that a value has a certain *shape*, and can *extract* information from the value when it has the matching shape. Pattern matching provides more concise syntax for algorithms you already use today. You already create pattern matching algorithms using existing syntax. You write `if` or `switch` statements that test values. Then, when those statements match, you extract and use information from that value. The new syntax elements are extensions to statements you're already familiar with: `is` and `switch`. These new extensions combine testing a value and extracting that information.

In this article, we'll look at the new syntax to show you how it enables readable, concise code. Pattern matching enables idioms where data and the code are separated, unlike object-oriented designs where data and the methods that manipulate them are tightly coupled.

To illustrate these new idioms, let's work with structures that represent geometric shapes using pattern matching statements. You're probably familiar with building class hierarchies and creating virtual methods and overridden methods to customize object behavior based on the runtime type of the object.

Those techniques aren't possible for data that isn't structured in a class hierarchy. When data and methods are separate, you need other tools. The new *pattern matching* constructs enable cleaner syntax to examine data and manipulate control flow based on any condition of that data. You already write `if` statements and `switch` that test a variable's value. You write `is` statements that test a variable's type. *Pattern matching* adds new capabilities to those statements.

In this article, you'll build a method that computes the area of different geometric shapes. But, you'll do it without resorting to object-oriented techniques and building a class hierarchy for the different shapes. You'll use *pattern matching* instead. As you go through this sample, contrast this code with how it would be structured as an object hierarchy. When the data you must query and manipulate isn't a class hierarchy, pattern matching enables elegant designs.

Rather than starting with an abstract shape definition and adding different specific shape classes, let's start instead with simple data only definitions for each of the geometric shapes:

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}
public struct Rectangle
{
    public double Length { get; }
    public double Height { get; }

    public Rectangle(double length, double height)
    {
        Length = length;
        Height = height;
    }
}
public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}
```

From these structures, let's write a method that computes the area of some shape.

The `is` type pattern expression

Before C# 7.0, you'd need to test each type in a series of `if` and `is` statements:

```
public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

That code above is a classic expression of the *type pattern*: You're testing a variable to determine its type and taking a different action based on that type.

This code becomes simpler using extensions to the `is` expression to assign a variable if the test succeeds:

```
public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

In this updated version, the `is` expression both tests the variable and assigns it to a new variable of the proper type. Also, notice that this version includes the `Rectangle` type, which is a `struct`. The new `is` expression works with value types as well as reference types.

Language rules for pattern matching expressions help you avoid misusing the results of a match expression. In the example above, the variables `s`, `c`, and `r` are only in scope and definitely assigned when the respective pattern match expressions have `true` results. If you try to use either variable in another location, your code generates compiler errors.

Let's examine both of those rules in detail, beginning with scope. The variable `c` is in scope only in the `else` branch of the first `if` statement. The variable `s` is in scope in the method `ComputeAreaModernIs`. That's because each branch of an `if` statement establishes a separate scope for variables. However, the `if` statement itself doesn't. That means variables declared in the `if` statement are in the same scope as the `if` statement (the method in this case.) This behavior isn't specific to pattern matching, but is the defined behavior for variable scopes and `if` and `else` statements.

The variables `c` and `s` are assigned when the respective `if` statements are true because of the definitely assigned when true mechanism.

TIP

The samples in this topic use the recommended construct where a pattern match `is` expression definitely assigns the match variable in the `true` branch of the `if` statement. You could reverse the logic by saying `if (!(shape is Square s))` and the variable `s` would be definitely assigned only in the `false` branch. While this is valid C#, it is not recommended because it is more confusing to follow the logic.

These rules mean that you're unlikely to accidentally access the result of a pattern match expression when that pattern wasn't met.

Using pattern matching `switch` statements

As time goes on, you may need to support other shape types. As the number of conditions you're testing grows, you'll find that using the `is` pattern matching expressions can become cumbersome. In addition to requiring `if` statements on each type you want to check, the `is` expressions are limited to testing if the input matches a single type. In this case, you'll find that the `switch` pattern matching expressions becomes a better choice.

The traditional `switch` statement was a pattern expression: it supported the constant pattern. You could compare a variable to any constant used in a `case` statement:

```
public static string GenerateMessage(params string[] parts)
{
    switch (parts.Length)
    {
        case 0:
            return "No elements to the input";
        case 1:
            return $"One element: {parts[0]}";
        case 2:
            return $"Two elements: {parts[0]}, {parts[1]}";
        default:
            return $"Many elements. Too many to write";
    }
}
```

The only pattern supported by the `switch` statement was the constant pattern. It was further limited to numeric types and the `string` type. Those restrictions have been removed, and you can now write a `switch` statement using the type pattern:

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

The pattern matching `switch` statement uses familiar syntax to developers who have used the traditional C-style

`switch` statement. Each `case` is evaluated and the code beneath the condition that matches the input variable is executed. Code execution can't "fall through" from one case expression to the next; the syntax of the `case` statement requires that each `case` end with a `break`, `return`, or `goto`.

> **NOTE**
> The `goto` statements to jump to another label are valid only for the constant pattern (the classic switch statement).

There are important new rules governing the `switch` statement. The restrictions on the type of the variable in the `switch` expression have been removed. Any type, such as `object` in this example, may be used. The case expressions are no longer limited to constant values. Removing that limitation means that reordering `switch` sections may change a program's behavior.

When limited to constant values, no more than one `case` label could match the value of the `switch` expression. Combine that with the rule that every `switch` section must not fall through to the next section, and it followed that the `switch` sections could be rearranged in any order without affecting behavior. Now, with more generalized `switch` expressions, the order of each section matters. The `switch` expressions are evaluated in textual order. Execution transfers to the first `switch` label that matches the `switch` expression.
The `default` case will only be executed if no other case labels match. The `default` case is evaluated last, regardless of its textual order. If there's no `default` case, and none of the other `case` statements match, execution continues at the statement following the `switch` statement. None of the `case` labels code is executed.

`when` clauses in `case` expressions

You can make special cases for those shapes that have 0 area by using a `when` clause on the `case` label. A square with a side length of 0, or a circle with a radius of 0 has a 0 area. You specify that condition using a `when` clause on the `case` label:

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

This change demonstrates a few important points about the new syntax. First, multiple `case` labels can be applied to one `switch` section. The statement block is executed when any of those labels is `true`. In this instance, if the `switch` expression is either a circle or a square with 0 area, the method returns the constant 0.

This example introduces two different variables in the two `case` labels for the first `switch` block. Notice that the statements in this `switch` block don't use either the variables `c` (for the circle) or `s` (for the square). Neither of those variables is definitely assigned in this `switch` block. If either of these cases match, clearly one of the variables has been assigned. However, it's impossible to tell *which* has been assigned at compile time, because

either case could match at runtime. For that reason, most times when you use multiple `case` labels for the same block, you won't introduce a new variable in the `case` statement, or you'll only use the variable in the `when` clause.

Having added those shapes with 0 area, let's add a couple more shape types: a rectangle and a triangle:

```
public static double ComputeArea_Version4(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

This set of changes adds `case` labels for the degenerate case, and labels and blocks for each of the new shapes.

Finally, you can add a `null` case to ensure the argument isn't `null`:

```
public static double ComputeArea_Version5(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        case null:
            throw new ArgumentNullException(paramName: nameof(shape), message: "Shape must not be null");
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

The special behavior for the `null` pattern is interesting because the constant `null` in the pattern doesn't have a type but can be converted to any reference type or nullable type. Rather than convert a `null` to any type, the

language defines that a `null` value won't match any type pattern, regardless of the compile-time type of the variable. This behavior makes the new `switch` based type pattern consistent with the `is` statement: `is` statements always return `false` when the value being checked is `null`. It's also simpler: once you've checked the type, you don't need an additional null check. You can see that from the fact that there are no null checks in any of the case blocks of the samples above: they aren't necessary, since matching the type pattern guarantees a non-null value.

`var` declarations in `case` expressions

The introduction of `var` as one of the match expressions introduces new rules to the pattern match.

The first rule is that the `var` declaration follows the normal type inference rules: The type is inferred to be the static type of the switch expression. From that rule, the type always matches.

The second rule is that a `var` declaration doesn't have the null check that other type pattern expressions include. That means the variable may be null, and a null check is necessary in that case.

Those two rules mean that in many instances, a `var` declaration in a `case` expression matches the same conditions as a `default` expression. Because any non-default case is preferred to the `default` case, the `default` case will never execute.

> **NOTE**
> The compiler does not emit a warning in those cases where a `default` case has been written but will never execute. This is consistent with current `switch` statement behavior where all possible cases have been listed.

The third rule introduces uses where a `var` case may be useful. Imagine that you're doing a pattern match where the input is a string and you're searching for known command values. You might write something like:

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

The `var` case matches `null`, the empty string, or any string that contains only white space. Notice that the preceding code uses the `?.` operator to ensure that it doesn't accidentally throw a NullReferenceException. The `default` case handles any other string values that aren't understood by this command parser.

This is one example where you may want to consider a `var` case expression that is distinct from a `default` expression.

Conclusions

Pattern Matching constructs enable you to easily manage control flow among different variables and types that aren't related by an inheritance hierarchy. You can also control logic to use any condition you test on the variable. It enables patterns and idioms that you'll need more often as you build more distributed applications, where data and the methods that manipulate that data are separate. You'll notice that the shape structs used in this sample don't contain any methods, just read-only properties. Pattern Matching works with any data type. You write expressions that examine the object, and make control flow decisions based on those conditions.

Compare the code from this sample with the design that would follow from creating a class hierarchy for an abstract `Shape` and specific derived shapes each with their own implementation of a virtual method to calculate the area. You'll often find that pattern matching expressions can be a very useful tool when you're working with data and want to separate the data storage concerns from the behavior concerns.

Write safe and efficient C# code

6/11/2019 • 14 minutes to read • Edit Online

New features in C# enable you to write verifiable safe code with better performance. If you carefully apply these techniques, fewer scenarios require unsafe code. These features make it easier to use references to value types as method arguments and method returns. When done safely, these techniques minimize copying value types. By using value types, you can minimize the number of allocations and garbage collection passes.

Much of the sample code in this article uses features added in C# 7.2. To use those features, you must configure your project to use C# 7.2 or later. For more information on setting the language version, see configure the language version.

This article focuses on techniques for efficient resource management. One advantage to using value types is that they often avoid heap allocations. The disadvantage is that they're copied by value. This tradeoff makes it harder to optimize algorithms that operate on large amounts of data. New language features in C# 7.2 provide mechanisms that enable safe efficient code using references to value types. Use these features wisely to minimize both allocations and copy operations. This article explores those new features.

This article focuses on the following resource management techniques:

- Declare a `readonly struct` to express that a type is **immutable** and enables the compiler to save copies when using `in` parameters.
- Use a `ref readonly` return when the return value is a `struct` larger than IntPtr.Size and the storage lifetime is greater than the method returning the value.
- When the size of a `readonly struct` is bigger than IntPtr.Size, you should pass it as an `in` parameter for performance reasons.
- Never pass a `struct` as an `in` parameter unless it's declared with the `readonly` modifier because it may negatively affect performance and could lead to an obscure behavior.
- Use a `ref struct`, or a `readonly ref struct` such as Span<T> or ReadOnlySpan<T> to work with memory as a sequence of bytes.

These techniques force you to balance two competing goals with regard to **references** and **values**. Variables that are reference types hold a reference to the location in memory. Variables that are value types directly contain their value. These differences highlight the key differences that are important for managing memory resources. **Value types** are typically copied when passed to a method or returned from a method. This behavior includes copying the value of `this` when calling members of a value type. The cost of the copy is related to the size of the type. **Reference types** are allocated on the managed heap. Each new object requires a new allocation, and subsequently must be reclaimed. Both these operations take time. The reference is copied when a reference type is passed as an argument to a method or returned from a method.

This article uses the following example concept of the 3D-point structure to explain these recommendations:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Different examples use different implementations of this concept.

Declare readonly structs for immutable value types

Declaring a `struct` using the `readonly` modifier informs the compiler that your intent is to create an immutable type. The compiler enforces that design decision with the following rules:

- All field members must be `readonly`
- All properties must be read-only, including auto-implemented properties.

These two rules are sufficient to ensure that no member of a `readonly struct` modifies the state of that struct. The `struct` is immutable. The `Point3D` structure could be defined as an immutable struct as shown in the following example:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Follow this recommendation whenever your design intent is to create an immutable value type. Any performance improvements are an added benefit. The `readonly struct` clearly expresses your design intent.

Use `ref readonly return` statements for large structures when possible

You can return values by reference when the value being returned isn't local to the returning method. Returning by reference means that only the reference is copied, not the structure. In the following example, the `Origin` property can't use a `ref` return because the value being returned is a local variable:

```
public Point3D Origin => new Point3D(0,0,0);
```

However, the following property definition can be returned by reference because the returned value is a static member:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

You don't want callers modifying the origin, so you should return the value by `readonly ref`:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Returning `ref readonly` enables you to save copying larger structures and preserve the immutability of your internal data members.

At the call site, callers make the choice to use the `Origin` property as a `readonly ref` or as a value:

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

The first assignment in the preceding code makes a copy of the `Origin` constant and assigns that copy. The second assigns a reference. Notice that the `readonly` modifier must be part of the declaration of the variable. The reference to which it refers can't be modified. Attempts to do so result in a compile-time error.

The `readonly` modifier is required on the declaration of `originReference`.

The compiler enforces that the caller can't modify the reference. Attempts to assign the value directly generate a compile-time error. However, the compiler can't know if any member method modifies the state of the struct. To ensure that the object isn't modified, the compiler creates a copy and calls member references using that copy. Any modifications are to that defensive copy.

Apply the `in` modifier to `readonly struct` parameters larger than `System.IntPtr.Size`

The `in` keyword complements the existing `ref` and `out` keywords to pass arguments by reference. The `in` keyword specifies passing the argument by reference, but the called method doesn't modify the value.

This addition provides a full vocabulary to express your design intent. Value types are copied when passed to a called method when you don't specify any of the following modifiers in the method signature. Each of these modifiers specifies that a variable is passed by reference, avoiding the copy. Each modifier expresses a different intent:

- `out`: This method sets the value of the argument used as this parameter.
- `ref`: This method may set the value of the argument used as this parameter.
- `in`: This method doesn't modify the value of the argument used as this parameter.

Add the `in` modifier to pass an argument by reference and declare your design intent to pass arguments by reference to avoid unnecessary copying. You don't intend to modify the object used as that argument.

This practice often improves performance for readonly value types that are larger than IntPtr.Size. For simple types (`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` and `bool`, and `enum` types), any potential performance gains are minimal. In fact, performance may degrade by using pass-by-reference for types smaller than IntPtr.Size.

The following code shows an example of a method that calculates the distance between two points in 3D space.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The arguments are two structures that each contain three doubles. A double is 8 bytes, so each argument is 24 bytes. By specifying the `in` modifier, you pass a 4 byte or 8-byte reference to those arguments, depending on the architecture of the machine. The difference in size is small, but it adds up when your application calls this method in a tight loop using many different values.

The `in` modifier complements `out` and `ref` in other ways as well. You can't create overloads of a method that differ only in the presence of `in`, `out`, or `ref`. These new rules extend the same behavior that had always been defined for `out` and `ref` parameters. Like the `out` and `ref` modifiers, value types aren't boxed because the `in` modifier is applied.

The `in` modifier may be applied to any member that takes parameters: methods, delegates, lambdas, local functions, indexers, operators.

Another feature of `in` parameters is that you may use literal values or constants for the argument to an `in` parameter. Also, unlike a `ref` or `out` parameter, you don't need to apply the `in` modifier at the call site. The following code shows you two examples of calling the `CalculateDistance` method. The first uses two local variables passed by reference. The second includes a temporary variable created as part of the method call.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

There are several ways in which the compiler enforces the read-only nature of an `in` argument. First of all, the called method can't directly assign to an `in` parameter. It can't directly assign to any field of an `in` parameter when that value is a `struct` type. In addition, you can't pass an `in` parameter to any method using the `ref` or `out` modifier. These rules apply to any field of an `in` parameter, provided the field is a `struct` type and the parameter is also a `struct` type. In fact, these rules apply for multiple layers of member access provided the types at all levels of member access are `structs`. The compiler enforces that `struct` types passed as `in` arguments and their `struct` members are read-only variables when used as arguments to other methods.

The use of `in` parameters can avoid the potential performance costs of making copies. It doesn't change the semantics of any method call. Therefore, you don't need to specify the `in` modifier at the call site. Omitting the `in` modifier at the call site informs the compiler that it's allowed to make a copy of the argument for the following reasons:

- There exists an implicit conversion but not an identity conversion from the argument type to the parameter type.
- The argument is an expression but doesn't have a known storage variable.
- An overload exists that differs by the presence or absence of `in`. In that case, the by value overload is a better match.

These rules are useful as you update existing code to use read-only reference arguments. Inside the called method, you can call any instance method that uses by value parameters. In those instances, a copy of the `in` parameter is created. Because the compiler may create a temporary variable for any `in` parameter, you can also specify default values for any `in` parameter. The following code specifies the origin (point 0,0) as the default value for the second point:

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

To force the compiler to pass read-only arguments by reference, specify the `in` modifier on the arguments at the call site, as shown in the following code:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

This behavior makes it easier to adopt `in` parameters over time in large codebases where performance gains are possible. You add the `in` modifier to method signatures first. Then, you can add the `in` modifier at call sites and create `readonly struct` types to enable the compiler to avoid creating defensive copies of `in` parameters in more locations.

The `in` parameter designation can also be used with reference types or numeric values. However, the benefits in both cases are minimal, if any.

Never use mutable structs as in `in` argument

The techniques described above explain how to avoid copies by returning references and passing values by reference. These techniques work best when the argument types are declared as `readonly struct` types. Otherwise, the compiler must create **defensive copies** in many situations to enforce the readonly-ness of any arguments. Consider the following example that calculates the distance of a 3D point from the origin:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The `Point3D` structure is *not* a readonly struct. There are six different property access calls in the body of this method. On first examination, you may have thought these accesses were safe. After all, a `get` accessor shouldn't modify the state of the object. But there's no language rule that enforces that. It's only a common convention. Any type could implement a `get` accessor that modified the internal state. Without some language guarantee, the compiler must create a temporary copy of the argument before calling any member. The temporary storage is created on the stack, the values of the argument are copied to the temporary storage, and the value is copied to the stack for each member access as the `this` argument. In many situations, these copies harm performance enough that pass-by-value is faster than pass-by-readonly-reference when the argument type isn't a `readonly struct`.

Instead, if the distance calculation uses the immutable struct, `ReadonlyPoint3D`, temporary objects are not needed:

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

The compiler generates more efficient code when you call members of a `readonly struct`: The `this` reference, instead of a copy of the receiver, is always an `in` parameter passed by reference to the member method. This optimization saves copying when you use a `readonly struct` as an `in` argument.

You should not pass a nullable value type as an `in` argument. The Nullable<T> type is not declared as a read-only struct. That means the compiler must generate defensive copies for any nullable value type argument passed to a method using the `in` modifier on the parameter declaration.

You can see an example program that demonstrates the performance differences using Benchmark.net in our samples repository on GitHub. It compares passing a mutable struct by value and by reference with passing an immutable struct by value and by reference. The use of the immutable struct and pass by reference is fastest.

Use `ref struct` types to work with blocks or memory on a single stack frame

A related language feature is the ability to declare a value type that must be constrained to a single stack frame. This restriction enables the compiler to make several optimizations. The primary motivation for this feature was Span<T> and related structures. You'll achieve performance improvements from these enhancements by using new and updated .NET APIs that make use of the Span<T> type.

You may have similar requirements working with memory created using `stackalloc` or when using memory from interop APIs. You can define your own `ref struct` types for those needs.

`readonly ref struct` type

Declaring a struct as `readonly ref` combines the benefits and restrictions of `ref struct` and `readonly struct` declarations. The memory used by the readonly span is restricted to a single stack frame, and the memory used by the readonly span can't be modified.

Conclusions

Using value types minimizes the number of allocation operations:

- Storage for value types is stack allocated for local variables and method arguments.
- Storage for value types that are members of other objects is allocated as part of that object, not as a separate allocation.
- Storage for value type return values is stack allocated.

Contrast that with reference types in those same situations:

- Storage for reference types are heap allocated for local variables and method arguments. The reference is stored on the stack.
- Storage for reference types that are members of other objects are separately allocated on the heap. The containing object stores the reference.
- Storage for reference type return values is heap allocated. The reference to that storage is stored on the stack.

Minimizing allocations comes with tradeoffs. You copy more memory when the size of the `struct` is larger than the size of a reference. A reference is typically 64 bits or 32 bits, and depends on the target machine CPU.

These tradeoffs generally have minimal performance impact. However, for large structs or larger collections, the performance impact increases. The impact can be large in tight loops and hot paths for programs.

These enhancements to the C# language are designed for performance critical algorithms where minimizing memory allocations is a major factor in achieving the necessary performance. You may find that you don't often use these features in the code you write. However, these enhancements have been adopted throughout .NET. As more and more APIs make use of these features, you'll see the performance of your applications improve.

See also

- ref keyword
- Ref returns and ref locals

Expression Trees

4/8/2019 • 2 minutes to read • Edit Online

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you are not familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) *Expression Trees* provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you can examine, modify, or execute. These tools give you the power to manipulate code during run time. You can write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you can modify running algorithms, and even translate C# expressions into another form for execution in another environment.

You've likely already written code that uses Expression Trees. Entity Framework's LINQ APIs accept Expression Trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

The remaining sections of this tutorial will explore what Expression Trees are, examine the framework classes that support expression trees, and show you how to work with expression trees. You'll learn how to read expression trees, how to create expression trees, how to create modified expression trees, and how to execute the code represented by expression trees. After reading, you will be ready to use these structures to create rich adaptive algorithms.

1. Expression Trees Explained

   Understand the structure and concepts behind *Expression Trees*.

2. Framework Types Supporting Expression Trees

   Learn about the structures and classes that define and manipulate expression trees.

3. Executing Expressions

   Learn how to convert an expression tree represented as a Lambda Expression into a delegate and execute the resulting delegate.

4. Interpreting Expressions

   Learn how to traverse and examine *expression trees* to understand what code the expression tree represents.

5. Building Expressions

   Learn how to construct the nodes for an expression tree and build expression trees.

6. Translating Expressions

   Learn how to build a modified copy of an expression tree, or translate an expression tree into a different format.

7. Summing up

Review the information on expression trees.

Expression Trees Explained

4/16/2019 • 4 minutes to read • Edit Online

Previous -- Overview

An Expression Tree is a data structure that defines code. They are based on the same structures that a compiler uses to analyze code and generate the compiled output. As you read through this tutorial, you will notice quite a bit of similarity between Expression Trees and the types used in the Roslyn APIs to build Analyzers and CodeFixes. (Analyzers and CodeFixes are NuGet packages that perform static analysis on code and can suggest potential fixes for a developer.) The concepts are similar, and the end result is a data structure that allows examination of the source code in a meaningful way. However, Expression Trees are based on a totally different set of classes and APIs than the Roslyn APIs.

Let's look at a simple example. Here's a line of code:

```
var sum = 1 + 2;
```

If you were to analyze this as an expression tree, the tree contains several nodes. The outermost node is a variable declaration statement with assignment (`var sum = 1 + 2;`) That outermost node contains several child nodes: a variable declaration, an assignment operator, and an expression representing the right hand side of the equals sign. That expression is further subdivided into expressions that represent the addition operation, and left and right operands of the addition.

Let's drill down a bit more into the expressions that make up the right side of the equals sign. The expression is `1 + 2`. That's a binary expression. More specifically, it's a binary addition expression. A binary addition expression has two children, representing the left and right nodes of the addition expression. Here, both nodes are constant expressions: The left operand is the value `1`, and the right operand is the value `2`.

Visually, the entire statement is a tree: You could start at the root node, and travel to each node in the tree to see the code that makes up the statement:

- Variable declaration statement with assignment (`var sum = 1 + 2;`)
  - Implicit variable type declaration (`var sum`)
    - Implicit var keyword (`var`)
    - Variable name declaration (`sum`)
  - Assignment operator (`=`)
  - Binary addition expression (`1 + 2`)
    - Left operand (`1`)
    - Addition operator (`+`)
    - Right operand (`2`)

This may look complicated, but it is very powerful. Following the same process, you can decompose much more complicated expressions. Consider this expression:

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

The expression above is also a variable declaration with an assignment. In this instance, the right hand side of the assignment is a much more complicated tree. I'm not going to decompose this expression, but consider what the different nodes might be. There are method calls using the current object as a receiver, one that has an explicit `this` receiver, one that does not. There are method calls using other receiver objects, there are constant arguments of different types. And finally, there is a binary addition operator. Depending on the return type of `SecretSauceFunction()` or `MoreSecretSauce()`, that binary addition operator may be a method call to an overridden addition operator, resolving to a static method call to the binary addition operator defined for a class.

Despite this perceived complexity, the expression above creates a tree structure that can be navigated as easily as the first sample. You can keep traversing child nodes to find leaf nodes in the expression. Parent nodes will have references to their children, and each node has a property that describes what kind of node it is.

The structure of an expression tree is very consistent. Once you've learned the basics, you can understand even the most complex code when it is represented as an expression tree. The elegance in the data structure explains how the C# compiler can analyze the most complex C# programs and create proper output from that complicated source code.

Once you become familiar with the structure of expression trees, you will find that knowledge you've gained quickly enables you to work with many more and more advanced scenarios. There is incredible power to expression trees.

In addition to translating algorithms to execute in other environments, expression trees can be used to make it easier to write algorithms that inspect code before executing it. You can write a method whose arguments are expressions and then examine those expressions before executing the code. The Expression Tree is a full representation of the code: you can see values of any sub-expression. You can see method and property names. You can see the value of any constant expressions. You can also convert an expression tree into an executable delegate, and execute the code.

The APIs for Expression Trees enable you to create trees that represent almost any valid code construct. However, to keep things as simple as possible, some C# idioms cannot be created in an expression tree. One example is asynchronous expressions (using the `async` and `await` keywords). If your needs require asynchronous algorithms, you would need to manipulate the `Task` objects directly, rather than rely on the compiler support. Another is in creating loops. Typically, you create these by using `for`, `foreach`, `while` or `do` loops. As you'll see later in this series, the APIs for expression trees support a single loop expression, with `break` and `continue` expressions that control repeating the loop.

The one thing you can't do is modify an expression tree. Expression Trees are immutable data structures. If you want to mutate (change) an expression tree, you must create a new tree that is a copy of the original, but with your desired changes.

Next -- Framework Types Supporting Expression Trees

Framework Types Supporting Expression Trees

4/8/2019 • 3 minutes to read • Edit Online

Previous -- Expression Trees Explained

There is a large list of classes in the .NET Core framework that work with Expression Trees. You can see the full list at System.Linq.Expressions. Rather than run through the full list, let's understand how the framework classes have been designed.

In language design, an expression is a body of code that evaluates and returns a value. Expressions may be very simple: the constant expression `1` returns the constant value of 1. They may be more complicated: The expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` returns one root for a quadratic equation (in the case where the equation has a solution).

It all starts with System.Linq.Expression

One of the complexities of working with expression trees is that many different kinds of expressions are valid in many places in programs. Consider an assignment expression. The right hand side of an assignment could be a constant value, a variable, a method call expression, or others. That language flexibility means that you may encounter many different expression types anywhere in the nodes of a tree when you traverse an expression tree. Therefore, when you can work with the base expression type, that's the simplest way to work. However, sometimes you need to know more. The base Expression class contains a `NodeType` property for this purpose. It returns an `ExpressionType` which is an enumeration of possible expression types. Once you know the type of the node, you can cast it to that type, and perform specific actions knowing the type of the expression node. You can search for certain node types, and then work with the specific properties of that kind of expression.

For example, this code will print the name of a variable for a variable access expression. I've followed the practice of checking the node type, then casting to a variable access expression and then checking the properties of the specific expression type:

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Creating Expression Trees

The `System.Linq.Expression` class also contains many static methods to create expressions. These methods create an expression node using the arguments supplied for its children. In this way, you build an expression up from its leaf nodes. For example, this code builds an Add expression:

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

You can see from this simple example that many types are involved in creating and working with expression trees. That complexity is necessary to provide the capabilities of the rich vocabulary provided by the C# language.

Navigating the APIs

There are Expression node types that map to almost all of the syntax elements of the C# language. Each type has specific methods for that type of language element. It's a lot to keep in your head at one time. Rather than try to memorize everything, here are the techniques I use to work with Expression trees:

1. Look at the members of the `ExpressionType` enum to determine possible nodes you should be examining. This really helps when you want to traverse and understand an expression tree.
2. Look at the static members of the `Expression` class to build an expression. Those methods can build any expression type from a set of its child nodes.
3. Look at the `ExpressionVisitor` class to build a modified expression tree.

You'll find more as you look at each of those three areas. Invariably, you will find what you need when you start with one of those three steps.

Next -- Executing Expression Trees

Executing Expression Trees

3/1/2019 • 6 minutes to read • Edit Online

Previous -- Framework Types Supporting Expression Trees

An *expression tree* is a data structure that represents some code. It is not compiled and executable code. If you want to execute the .NET code that is represented by an expression tree, you must convert it into executable IL instructions.

Lambda Expressions to Functions

You can convert any LambdaExpression, or any type derived from LambdaExpression into executable IL. Other expression types cannot be directly converted into code. This restriction has little effect in practice. Lambda expressions are the only types of expressions that you would want to execute by converting to executable intermediate language (IL). (Think about what it would mean to directly execute a `ConstantExpression`. Would it mean anything useful?) Any expression tree that is a `LambdaExpression`, or a type derived from `LambdaExpression` can be converted to IL. The expression type `Expression<TDelegate>` is the only concrete example in the .NET Core libraries. It's used to represent an expression that maps to any delegate type. Because this type maps to a delegate type, .NET can examine the expression, and generate IL for an appropriate delegate that matches the signature of the lambda expression.

In most cases, this creates a simple mapping between an expression, and its corresponding delegate. For example, an expression tree that is represented by `Expression<Func<int>>` would be converted to a delegate of the type `Func<int>`. For a lambda expression with any return type and argument list, there exists a delegate type that is the target type for the executable code represented by that lambda expression.

The `LambdaExpression` type contains `Compile` and `CompileToMethod` members that you would use to convert an expression tree to executable code. The `Compile` method creates a delegate. The `CompileToMethod` method updates a `MethodBuilder` object with the IL that represents the compiled output of the expression tree. Note that `CompileToMethod` is only available in the full desktop framework, not in the .NET Core.

Optionally, you can also provide a `DebugInfoGenerator` that will receive the symbol debugging information for the generated delegate object. This enables you to convert the expression tree into a delegate object, and have full debugging information about the generated delegate.

You would convert an expression into a delegate using the following code:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Notice that the delegate type is based on the expression type. You must know the return type and the argument list if you want to use the delegate object in a strongly typed manner. The `LambdaExpression.Compile()` method returns the `Delegate` type. You will have to cast it to the correct delegate type to have any compile-time tools check the argument list or return type.

Execution and Lifetimes

You execute the code by invoking the delegate created when you called `LambdaExpression.Compile()`. You can see this above where `add.Compile()` returns a delegate. Invoking that delegate, by calling `func()` executes the code.

That delegate represents the code in the expression tree. You can retain the handle to that delegate and invoke it later. You don't need to compile the expression tree each time you want to execute the code it represents. (Remember that expression trees are immutable, and compiling the same expression tree later will create a delegate that executes the same code.)

I will caution you against trying to create any more sophisticated caching mechanisms to increase performance by avoiding unnecessary compile calls. Comparing two arbitrary expression trees to determine if they represent the same algorithm will also be time consuming to execute. You'll likely find that the compute time you save avoiding any extra calls to `LambdaExpression.Compile()` will be more than consumed by the time executing code that determines of two different expression trees result in the same executable code.

Caveats

Compiling a lambda expression to a delegate and invoking that delegate is one of the simplest operations you can perform with an expression tree. However, even with this simple operation, there are caveats you must be aware of.

Lambda Expressions create closures over any local variables that are referenced in the expression. You must guarantee that any variables that would be part of the delegate are usable at the location where you call `Compile`, and when you execute the resulting delegate.

In general, the compiler will ensure that this is true. However, if your expression accesses a variable that implements `IDisposable`, it's possible that your code might dispose of the object while it is still held by the expression tree.

For example, this code works fine, because `int` does not implement `IDisposable`:

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

The delegate has captured a reference to the local variable `constant`. That variable is accessed at any time later, when the function returned by `CreateBoundFunc` executes.

However, consider this (rather contrived) class that implements `IDisposable`:

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

If you use it in an expression as shown below, you'll get an `ObjectDisposedException` when you execute the code referenced by the `Resource.Argument` property:

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

The delegate returned from this method has closed over the `constant` object, which has been disposed of. (It's been disposed, because it was declared in a `using` statement.)

Now, when you execute the delegate returned from this method, you'll have a `ObjectDisposedException` thrown at the point of execution.

It does seem strange to have a runtime error representing a compile-time construct, but that's the world we enter when we work with expression trees.

There are a lot of permutations of this problem, so it's hard to offer general guidance to avoid it. Be careful about accessing local variables when defining expressions, and be careful about accessing state in the current object (represented by `this`) when creating an expression tree that can be returned by a public API.

The code in your expression may reference methods or properties in other assemblies. That assembly must be accessible when the expression is defined, and when it is compiled, and when the resulting delegate is invoked. You'll be met with a `ReferencedAssemblyNotFoundException` in cases where it is not present.

Summary

Expression Trees that represent lambda expressions can be compiled to create a delegate that you can execute. This provides one mechanism to execute the code represented by an expression tree.

The Expression Tree does represent the code that would execute for any given construct you create. As long as the environment where you compile and execute the code matches the environment where you create the expression, everything works as expected. When that doesn't happen, the errors are very predictable, and they will be caught in your first tests of any code using the expression trees.

Next -- Interpreting Expressions

Interpreting Expressions

4/8/2019 • 14 minutes to read • Edit Online

Previous -- Executing Expressions

Now, let's write some code to examine the structure of an *expression tree*. Every node in an expression tree will be an object of a class that is derived from `Expression`.

That design makes visiting all the nodes in an expression tree a relatively straight forward recursive operation. The general strategy is to start at the root node and determine what kind of node it is.

If the node type has children, recursively visit the children. At each child node, repeat the process used at the root node: determine the type, and if the type has children, visit each of the children.

Examining an Expression with No Children

Let's start by visiting each node in a very simple expression tree. Here's the code that creates a constant expression and then examines its properties:

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

This will print the following:

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Now, let's write the code that would examine this expression and write out some important properties about it. Here's that code:

Examining a simple Addition Expression

Let's start with the addition sample from the introduction to this section.

```
Expression<Func<int>> sum = () => 1 + 2;
```

> I'm not using `var` to declare this expression tree, as it is not possible because the right-hand side of the assignment is implicitly typed. To understand this more deeply, read here.

The root node is a `LambdaExpression`. In order to get the interesting code on the right hand side of the `=>` operator, you need to find one of the children of the `LambdaExpression`. We'll do that with all the expressions in this section. The parent node does help us find the return type of the `LambdaExpression`.

To examine each node in this expression, we'll need to recursively visit a number of nodes. Here's a simple first implementation:

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type: {argumentExpression.Type.ToString()}, Name:
{argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");
```

This sample prints the following output:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

You'll notice a lot of repetition in the code sample above. Let's clean that up and build a more general purpose expression node visitor. That's going to require us to write a recursive algorithm. Any node could be of a type that might have children. Any node that has children requires us to visit those children and determine what that node is. Here's the cleaned up version that utilizes recursion to visit the addition operations:

```
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
```

```
            case ExpressionType.Add:
                return new BinaryVisitor((BinaryExpression)node);
            default:
                Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
                return default(Visitor);
        }
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name == null) ? "<null>" :
node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
```

```
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}
```

This algorithm is the basis of an algorithm that can visit any arbitrary `LambdaExpression`. There are a lot of holes, namely that the code I created only looks for a very small sample of the possible sets of expression tree nodes that it may encounter. However, you can still learn quite a bit from what it produces. (The default case in the `Visitor.CreateFromExpression` method prints a message to the error console when a new node type is encountered. That way, you know to add a new expression type.)

When you run this visitor on the addition expression shown above, you get the following output:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Now that you've built a more general visitor implementation, you can visit and process many more different types of expressions.

Examining an Addition Expression with Many Levels

Let's try a more complicated example, yet still limit the node types to addition only:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Before you run this on the visitor algorithm, try a thought exercise to work out what the output might be. Remember that the `+` operator is a *binary operator*: it must have two children, representing the left and right operands. There are several possible ways to construct a tree that could be correct:

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

You can see the separation into two possible answers to highlight the most promising. The first represents *right associative* expressions. The second represent *left associative* expressions. The advantage of both of those two formats is that the format scales to any arbitrary number of addition expressions.

If you do run this expression through the visitor, you will see this this output, verifying that the simple addition expression is *left associative*.

In order to run this sample, and see the full expression tree, I had to make one change to the source expression tree. When the expression tree contains all constants, the resulting tree simply contains the constant value of `10`. The compiler performs all the addition and reduces the expression to its simplest form. Simply adding one variable in the expression is sufficient to see the original tree:

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Create a visitor for this sum and run the visitor you'll see this output:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

You can also run any of the other samples through the visitor code and see what tree it represents. Here's an example of the `sum3` expression above (with an additional parameter to prevent the compiler from computing the constant):

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Here's the output from the visitor:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Notice that the parentheses are not part of the output. There are no nodes in the expression tree that represent the parentheses in the input expression. The structure of the expression tree contains all the information necessary to communicate the precedence.

Extending from this sample

The sample deals with only the most rudimentary expression trees. The code you've seen in this section only handles constant integers and the binary `+` operator. As a final sample, let's update the visitor to handle a more complicated expression. Let's make it work for this:

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

This code represents one possible implementation for the mathematical *factorial* function. The way I've written this code highlights two limitations of building expression trees by assigning lambda expressions to Expressions. First, statement lambdas are not allowed. That means I can't use loops, blocks, if / else statements, and other control structures common in C#. I'm limited to using expressions. Second, I can't recursively call the same expression. I could if it were already a delegate, but I can't call it in its expression tree form. In the section on building expression trees you'll learn techniques to overcome these limitations.

In this expression, you'll encounter nodes of all these types:

1. Equal (binary expression)
2. Multiply (binary expression)
3. Conditional (the ? : expression)
4. Method Call Expression (calling `Range()` and `Aggregate()`)

One way to modify the visitor algorithm is to keep executing it, and write the node type every time you reach your

`default` clause. After a few iterations, you'll have seen each of the potential nodes. Then, you have all you need. The result would be something like this:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

The ConditionalVisitor and MethodCallVisitor process those two nodes:

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach(var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

And the output for the expression tree would be:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef: False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef: False
                        The expression body is:
                                This binary expression is a Multiply expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product, ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor, ByRef: False
```

Extending the Sample Library

The samples in this section show the core techniques to visit and examine nodes in an expression tree. I glossed over many actions you might need in order to concentrate on the core tasks of visiting and accessing nodes in an expression tree.

First, the visitors only handle constants that are integers. Constant values could be any other numeric type, and the C# language supports conversions and promotions between those types. A more robust version of this code would mirror all those capabilities.

Even the last example recognizes a subset of the possible node types. You can still feed it many expressions that will cause it to fail. A full implementation is included in the .NET Standard under the name ExpressionVisitor and

can handle all the possible node types.

Finally, the library I used in this article was built for demonstration and learning. It's not optimized. I wrote it to make the structures used very clear, and to highlight the techniques used to visit the nodes and analyze what's there. A production implementation would pay more attention to performance than I have.

Even with those limitations, you should be well on your way to writing algorithms that read and understand expression trees.

Next -- Building Expressions

Building Expression Trees

6/19/2018 • 5 minutes to read • Edit Online

Previous -- Interpreting Expressions

All the expression trees you've seen so far have been created by the C# compiler. All you had to do was create a lambda expression that was assigned to a variable typed as an `Expression<Func<T>>` or some similar type. That's not the only way to create an expression tree. For many scenarios you may find that you need to build an expression in memory at runtime.

Building Expression Trees is complicated by the fact that those expression trees are immutable. Being immutable means that you must build the tree from the leaves up to the root. The APIs you'll use to build expression trees reflect this fact: The methods you'll use to build a node take all its children as arguments. Let's walk through a few examples to show you the techniques.

Creating Nodes

Let's start relatively simply again. We'll use the addition expression I've been working with throughout these sections:

```
Expression<Func<int>> sum = () => 1 + 2;
```

To construct that expression tree, you must construct the leaf nodes. The leaf nodes are constants, so you can use the `Expression.Constant` method to create the nodes:

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Next, you'll build the addition expression:

```
var addition = Expression.Add(one, two);
```

Once you've got the addition expression, you can create the lambda expression:

```
var lambda = Expression.Lambda(addition);
```

This is a very simple lambda expression, because it contains no arguments. Later in this section, you'll see how to map arguments to parameters and build more complicated expressions.

For expressions that are as simple as this one, you may combine all the calls into a single statement:

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Building a Tree

That's the basics of building an expression tree in memory. More complex trees generally mean more node types, and more nodes in the tree. Let's run through one more example and show two more node types that you will typically build when you create expression trees: the argument nodes, and method call nodes.

Let's build an expression tree to create this expression:

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

You'll start by creating parameter expressions for `x` and `y`:

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

Creating the multiplication and addition expressions follows the pattern you've already seen:

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Next, you need to create a method call expression for the call to `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });
var distance = Expression.Call(sqrtMethod, sum);
```

And then finally, you put the method call into a lambda expression, and make sure to define the arguments to the lambda expression:

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

In this more complicated example, you see a couple more techniques that you will often need to create expression trees.

First, you need to create the objects that represent parameters or local variables before you use them. Once you've created those objects, you can use them in your expression tree wherever you need.

Second, you need to use a subset of the Reflection APIs to create a `MethodInfo` object so that you can create an expression tree to access that method. You must limit yourself to the subset of the Reflection APIs that are available on the .NET Core platform. Again, these techniques will extend to other expression trees.

Building Code In Depth

You aren't limited in what you can build using these APIs. However, the more complicated expression tree that you want to build, the more difficult the code is to manage and to read.

Let's build an expression tree that is the equivalent of this code:

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Notice above that I did not build the expression tree, but simply the delegate. Using the `Expression` class, you can't build statement lambdas. Here's the code that is required to build the same functionality. It's complicated by the fact that there isn't an API to build a `while` loop, instead you need to build a loop that contains a conditional test, and a label target to break out of the loop.

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

The code to build the expression tree for the factorial function is quite a bit longer, more complicated, and it's riddled with labels and break statements and other elements we'd like to avoid in our everyday coding tasks.

For this section, I've also updated the visitor code to visit every node in this expression tree and write out information about the nodes that are created in this sample. You can view or download the sample code at the dotnet/docs GitHub repository. Experiment for yourself by building and running the samples. For download instructions, see Samples and Tutorials.

Examining the APIs

The expression tree APIs are some of the more difficult to navigate in .NET Core, but that's fine. Their purpose is a rather complex undertaking: writing code that generates code at runtime. They are necessarily complicated to provide a balance between supporting all the control structures available in the C# language and keeping the surface area of the APIs as small as reasonable. This balance means that many control structures are represented

not by their C# constructs, but by constructs that represent the underlying logic that the compiler generates from these higher level constructs.

Also, at this time, there are C# expressions that cannot be built directly using `Expression` class methods. In general, these will be the newest operators and expressions added in C# 5 and C# 6. (For example, `async` expressions cannot be built, and the new `?.` operator cannot be directly created.)

Next -- Translating Expressions

Translating Expression Trees

3/6/2019 • 6 minutes to read • Edit Online

Previous -- Building Expressions

In this final section, you'll learn how to visit each node in an expression tree while building a modified copy of that expression tree. These are the techniques that you will use in two important scenarios. The first is to understand the algorithms expressed by an expression tree so that it can be translated into another environment. The second is when you want to change the algorithm that has been created. This might be to add logging, intercept method calls and track them, or other purposes.

Translating is Visiting

The code you build to translate an expression tree is an extension of what you've already seen to visit all the nodes in a tree. When you translate an expression tree, you visit all the nodes, and while visiting them, build the new tree. The new tree may contain references to the original nodes, or new nodes that you have placed in the tree.

Let's see this in action by visiting an expression tree, and creating a new tree with some replacement nodes. In this example, let's replace any constant with a constant that is ten times larger. Otherwise, we'll leave the expression tree intact. Rather than reading the value of the constant, and replacing it with a new constant, we'll make this replacement by replacing the constant node with a new node that performs the multiplication.

Here, once you find a constant node, you create a new multiplication node whose children are the original constant, and the constant `10`:

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

By replacing the original node with the substitute, a new tree is formed that contains our modifications. We can verify that by compiling and executing the replaced tree.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Building a new tree is a combination of visiting the nodes in the existing tree, and creating new nodes and inserting them into the tree.

This example shows the importance of expression trees being immutable. Notice that the new tree created above contains a mixture of newly created nodes, and nodes from the existing tree. That's safe, because the nodes in the existing tree cannot be modified. This can result in significant memory efficiencies. The same nodes can be used throughout a tree, or in multiple expression trees. Since nodes can't be modified, the same node can be reused whenever its needed.

Traversing and Executing an Addition

Let's verify this by building a second visitor that walks the tree of addition nodes and computes the result. You can do this by making a couple modifications to the visitor that you've seen so far. In this new version, the visitor will return the partial sum of the addition operation up to this point. For a constant expression, that is simply the value of the constant expression. For an addition expression, the result is the sum of the left and right operands, once those trees have been traversed.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three= Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

There's quite a bit of code here, but the concepts are very approachable. This code visits children in a depth first search. When it encounters a constant node, the visitor returns the value of the constant. After the visitor has visited both children, those children will have computed the sum computed for that sub-tree. The addition node can now compute its sum. Once all the nodes in the expression tree have been visited, the sum will have been computed. You can trace the execution by running the sample in the debugger and tracing the execution.

Let's make it easier to trace how the nodes are analyzed and how the sum is computed by traversing the tree. Here's an updated version of the Aggregate method that includes quite a bit of tracing information:

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Running it on the same expression yields the following output:

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Trace the output and follow along in the code above. You should be able to work out how the code visits each node and computes the sum as it goes through the tree and finds the sum.

Now, let's look at a different run, with the expression given by `sum1`:

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

Here's the output from examining this expression:

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

While the final answer is the same, the tree traversal is completely different. The nodes are traveled in a different order, because the tree was constructed with different operations occurring first.

Learning More

This sample shows a small subset of the code you would build to traverse and interpret the algorithms represented by an expression tree. For a complete discussion of all the work necessary to build a general purpose library that translates expression trees into another language, please read this series by Matt Warren. It goes into great detail on how to translate any of the code you might find in an expression tree.

I hope you've now seen the true power of expression trees. You can examine a set of code, make any changes you'd like to that code, and execute the changed version. Because the expression trees are immutable, you can create new trees by using the components of existing trees. This minimizes the amount of memory needed to create modified expression trees.

Next -- Summing up

Expression Trees Summary

4/8/2019 • 2 minutes to read • Edit Online

Previous -- Translating Expressions

In this series, you've seen how you can use *expression trees* to create dynamic programs that interpret code as data and build new functionality based on that code.

You can examine expression trees to understand the intent of an algorithm. You can not only examine that code. You can build new expression trees that represent modified versions of the original code.

You can also use expression trees to look at an algorithm, and translate that algorithm into another language or environment.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees cannot contain `await` expressions, or `async` lambda expressions. Many of the features added in the C# 6 release don't appear exactly as written in expression trees. Instead, newer features will be exposed in expressions trees in the equivalent, earlier syntax. This may not be as much of a limitation as you might think. In fact, it means that your code that interprets expression trees will likely still work the same when new language features are introduced.

Even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It's a powerful tool, and it's one of the features of the .NET ecosystem that enables rich libraries such as Entity Framework to accomplish what they do.

Interoperability (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

The .NET Framework enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

Interoperability Overview
Describes methods to interoperate between C# managed code and unmanaged code.

How to: Access Office Interop Objects by Using Visual C# Features
Describes features that are introduced in Visual C# to facilitate Office programming.

How to: Use Indexed Properties in COM Interop Programming
Describes how to use indexed properties to access COM properties that have parameters.

How to: Use Platform Invoke to Play a Wave File
Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

Walkthrough: Office Programming
Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

Example COM Class
Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see Basic concepts in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Marshal.ReleaseComObject
- C# Programming Guide
- Interoperating with Unmanaged Code
- Walkthrough: Office Programming

Documenting your code with XML comments

5/15/2019 • 28 minutes to read • Edit Online

XML documentation comments are a special kind of comment, added above the definition of any user-defined type or member. They are special because they can be processed by the compiler to generate an XML documentation file at compile time. The compiler generated XML file can be distributed alongside your .NET assembly so that Visual Studio and other IDEs can use IntelliSense to show quick information about types or members. Additionally, the XML file can be run through tools like DocFX and Sandcastle to generate API reference websites.

XML documentation comments, like all other comments, are ignored by the compiler.

You can generate the XML file at compile time by doing one of the following:

- If you are developing an application with .NET Core from the command line, you can add a DocumentationFile element to the `<PropertyGroup>` section of your .csproj project file. The following example generates an XML file in the project directory with the same root filename as the assembly:

  ```
  <DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
  ```

  You can also specify the exact absolute or relative path and name of the XML file. The following example generates the XML file in the same directory as the debug version of an application:

  ```
  <DocumentationFile>bin\Debug\netcoreapp2.1\App.xml</DocumentationFile>
  ```

- If you are developing an application using Visual Studio, right-click on the project and select **Properties**. In the properties dialog, select the **Build** tab, and check **XML documentation file**. You can also change the location to which the compiler writes the file.

- If you are compiling a .NET Framework application from the command line, add the /doc compiler option when compiling.

XML documentation comments use triple forward slashes (`///`) and an XML formatted comment body. For example:

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{

}
```

Walkthrough

Let's walk through documenting a very basic math library to make it easy for new developers to understand/contribute and for third party developers to use.

Here's code for the simple math library:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

The sample library supports four major arithmetic operations `add`, `subtract`, `multiply` and `divide` on `int` and `double` data types.

Now you want to be able to create an API reference document from your code for third party developers who use your library but don't have access to the source code. As mentioned earlier XML documentation tags can be used to achieve this. You will now be introduced to the standard XML tags the C# compiler supports.

<summary>

The `<summary>` tag adds brief information about a type or member. I'll demonstrate its use by adding it to the `Math` class definition and the first `Add` method. Feel free to apply it to the rest of your code.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

The `<summary>` tag is very important, and we recommend that you include it because its content is the primary source of type or member information in IntelliSense or an API reference document.

<remarks>

The `<remarks>` tag supplements the information about types or members that the `<summary>` tag provides. In this example, you'll just add it to the class.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{

}
```

<returns>

The `<returns>` tag describes the return value of a method declaration. As before, the following example illustrates the `<returns>` tag on the first `Add` method. You can do the same on other methods.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

<value>

The `<value>` tag is similar to the `<returns>` tag, except that you use it for properties. Assuming your `Math` library had a static property called `PI`, here's how you'd use this tag:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

<example>

You use the `<example>` tag to include an example in your XML documentation. This involves using the child `<code>` tag.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

The `code` tag preserves line breaks and indentation for longer examples.

<para>

You use the `<para>` tag to format the content within its parent tag. `<para>` is usually used inside a tag, such as `<remarks>` or `<returns>`, to divide text into paragraphs. You can format the contents of the `<remarks>` tag for your class definition.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{

}
```

<c>

Still on the topic of formatting, you use the `<c>` tag for marking part of text as code. It's like the `<code>` tag but inline. It's useful when you want to show a quick code example as part of a tag's content. Let's update the documentation for the `Math` class.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{

}
```

<exception>

By using the `<exception>` tag, you let your developers know that a method can throw specific exceptions. Looking at your `Math` library, you can see that both `Add` methods throw an exception if a certain condition is met. Not so obvious, though, is that integer `Divide` method throws as well if the `b` parameter is zero. Now add exception documentation to this method.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
```

```
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Divides an integer by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two integers.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    /// <summary>
    /// Divides a double by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two doubles.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

The `cref` attribute represents a reference to an exception that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly. The compiler will issue a warning if its value cannot be resolved.

<see>

The `<see>` tag lets you create a clickable link to a documentation page for another code element. In our next example, we'll create a clickable link between the two `Add` methods.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

The `cref` is a **required** attribute that represents a reference to a type or its member that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly.

<seealso>

You use the `<seealso>` tag in the same way you do the `<see>` tag. The only difference is that its content is typically placed in a "See Also" section. Here we'll add a `seealso` tag on the integer `Add` method to reference other methods in the class that accept integer parameters:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

The `cref` attribute represents a reference to a type or its member that is available from the current compilation environment. This can be any type defined in the project or a referenced assembly.

<param>

You use the `<param>` tag to describe a method's parameters. Here's an example on the double `Add` method: The parameter the tag describes is specified in the **required** `name` attribute.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparam>

You use `<typeparam>` tag just like the `<param>` tag but for generic type or method declarations to describe a generic parameter. Add a quick generic method to your `Math` class to check if one quantity is greater than another.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<paramref>

Sometimes you might be in the middle of describing what a method does in what could be a `<summary>` tag, and you might want to make a reference to a parameter. The `<paramref>` tag is great for just this. Let's update the summary of our double based `Add` method. Like the `<param>` tag the parameter name is specified in the **required** `name` attribute.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparamref>

You use `<typeparamref>` tag just like the `<paramref>` tag but for generic type or method declarations to describe a generic parameter. You can use the same generic method you previously created.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<list>

You use the `<list>` tag to format documentation information as an ordered list, unordered list or table. Make an unordered list of every math operation your `Math` library supports.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{

}
```

You can make an ordered list or table by changing the `type` attribute to `number` or `table`, respectively.

Putting it all together

If you've followed this tutorial and applied the tags to your code where necessary, your code should now look similar to the following:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
```

```
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Add(4.5, 5.4);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();
```

```
        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <summary>
    /// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The difference between two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Subtract(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <summary>
    /// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
    /// </summary>
    /// <returns>
    /// The difference between two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Subtract(4.5, 5.4);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <summary>
    /// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Multiply(4, 5);
    /// if (c > 100)
```

```
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <summary>
    /// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Multiply(4.5, 5.4);
    /// if (c > 100.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <summary>
    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
```

```
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <summary>
    /// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Divide(4.5, 5.4);
    /// if (c > 1.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(int, int)"/> to divide integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

From your code, you can generate a detailed documentation website complete with clickable cross-references. But you're faced with another problem: your code has become hard to read. There's so much information to sift through that this is going to be a nightmare for any developer who wants to contribute to this code. Thankfully there's an XML tag that can help you deal with this:

<include>

The `<include>` tag lets you refer to comments in a separate XML file that describe the types and members in your source code, as opposed to placing documentation comments directly in your source code file.

Now you're going to move all your XML tags into a separate XML file named `docs.xml`. Feel free to name the file whatever you want.

```
<docs>
    <members name="math">
        <Math>
            <summary>
            The main <c>Math</c> class.
            Contains all methods for performing basic math functions.
            </summary>
            <remarks>
            <para>This class can add, subtract, multiply and divide.</para>
            <para>These operations can be performed on both integers and doubles.</para>
            </remarks>
        </Math>
        <AddInt>
            <summary>
            Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The sum of two integers.
```

```
        </returns>
        <example>
        <code>
        int c = Math.Add(4, 5);
        if (c > 10)
        {
            Console.WriteLine(c);
        }
        </code>
        </example>
        <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
        See <see cref="Math.Add(double, double)"/> to add doubles.
        <seealso cref="Math.Subtract(int, int)"/>
        <seealso cref="Math.Multiply(int, int)"/>
        <seealso cref="Math.Divide(int, int)"/>
        <param name="a">An integer.</param>
        <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
        <summary>
        Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
        </summary>
        <returns>
        The sum of two doubles.
        </returns>
        <example>
        <code>
        double c = Math.Add(4.5, 5.4);
        if (c > 10)
        {
            Console.WriteLine(c);
        }
        </code>
        </example>
        <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
        See <see cref="Math.Add(int, int)"/> to add integers.
        <seealso cref="Math.Subtract(double, double)"/>
        <seealso cref="Math.Multiply(double, double)"/>
        <seealso cref="Math.Divide(double, double)"/>
        <param name="a">A double precision number.</param>
        <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
        <summary>
        Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
        </summary>
        <returns>
        The difference between two integers.
        </returns>
        <example>
        <code>
        int c = Math.Subtract(4, 5);
        if (c > 1)
        {
            Console.WriteLine(c);
        }
        </code>
        </example>
        See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
        <seealso cref="Math.Add(int, int)"/>
        <seealso cref="Math.Multiply(int, int)"/>
        <seealso cref="Math.Divide(int, int)"/>
        <param name="a">An integer.</param>
        <param name="b">An integer.</param>
    </SubtractInt>
    <SubtractDouble>
        <summary>
```

```
            Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
            </summary>
            <returns>
            The difference between two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Subtract(4.5, 5.4);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Subtract(int, int)"/> to subtract integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </SubtractDouble>
        <MultiplyInt>
            <summary>
            Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Multiply(4, 5);
            if (c > 100)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </MultiplyInt>
        <MultiplyDouble>
            <summary>
            Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Multiply(4.5, 5.4);
            if (c > 100.0)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Multiply(int, int)"/> to multiply integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </MultiplyDouble>
        <DivideInt>
```

```
        <DivideInt>
            <summary>
            Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Divide(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(double, double)"/> to divide doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <param name="a">An integer dividend.</param>
            <param name="b">An integer divisor.</param>
        </DivideInt>
        <DivideDouble>
            <summary>
            Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Divide(4.5, 5.4);
            if (c > 1.0)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(int, int)"/> to divide integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <param name="a">A double precision dividend.</param>
            <param name="b">A double precision divisor.</param>
        </DivideDouble>
    </members>
</docs>
```

In the above XML, each member's documentation comments appear directly inside a tag named after what they do. You can choose your own strategy. Now that you have your XML comments in a separate file, let's see how your code can be made more readable by using the `<include>` tag:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'/>
public class Math
{
    // Adds two integers and returns the result
```

```
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'/>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'/>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'/>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'/>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'/>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'/>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'/>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'/>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

And there you have it: our code is back to being readable, and no documentation information has been lost.

The `file` attribute represents the name of the XML file containing the documentation.

The `path` attribute represents an XPath query to the `tag name` present in the specified `file`.

The `name` attribute represents the name specifier in the tag that precedes the comments.

The `id` attribute which can be used in place of `name` represents the ID for the tag that precedes the comments.

User Defined Tags

All the tags outlined above represent those that are recognized by the C# compiler. However, a user is free to define their own tags. Tools like Sandcastle bring support for extra tags like `<event>`, `<note>` and even support documenting namespaces. Custom or in-house documentation generation tools can also be used with the standard tags and multiple output formats from HTML to PDF can be supported.

Recommendations

Documenting code is recommended for many reasons. What follows are some best practices, general use case scenarios, and things that you should know when using XML documentation tags in your C# code.

- For the sake of consistency, all publicly visible types and their members should be documented. If you must do it, do it all.
- Private members can also be documented using XML comments. However, this exposes the inner (potentially confidential) workings of your library.
- At a bare minimum, types and their members should have a `<summary>` tag because its content is needed for IntelliSense.
- Documentation text should be written using complete sentences ending with full stops.
- Partial classes are fully supported, and documentation information will be concatenated into a single entry for that type.
- The compiler verifies the syntax of the `<exception>`, `<include>`, `<param>`, `<see>`, `<seealso>` and `<typeparam>` tags.
- The compiler validates the parameters that contain file paths and references to other parts of the code.

See also

- XML Documentation Comments (C# Programming Guide)
- Recommended Tags for Documentation Comments (C# Programming Guide)

Versioning in C#

2/28/2019 • 5 minutes to read • Edit Online

In this tutorial you'll learn what versioning means in .NET. You'll also learn the factors to consider when versioning your library as well as upgrading to a new version of a library.

Authoring Libraries

As a developer who has created .NET libraries for public use, you've most likely been in situations where you have to roll out new updates. How you go about this process matters a lot as you need to ensure that there's a seamless transition of existing code to the new version of your library. Here are several things to consider when creating a new release:

Semantic Versioning

Semantic versioning (SemVer for short) is a naming convention applied to versions of your library to signify specific milestone events. Ideally, the version information you give your library should help developers determine the compatibility with their projects that make use of older versions of that same library.

The most basic approach to SemVer is the 3 component format `MAJOR.MINOR.PATCH`, where:

- `MAJOR` is incremented when you make incompatible API changes
- `MINOR` is incremented when you add functionality in a backwards-compatible manner
- `PATCH` is incremented when you make backwards-compatible bug fixes

There are also ways to specify other scenarios like pre-release versions etc. when applying version information to your .NET library.

Backwards Compatibility

As you release new versions of your library, backwards compatibility with previous versions will most likely be one of your major concerns. A new version of your library is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. A new version of your library is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Here are some things to consider when trying to maintain backwards compatibility with older versions of your library:

- Virtual methods: When you make a virtual method non-virtual in your new version it means that projects that override that method will have to be updated. This is a huge breaking change and is strongly discouraged.
- Method signatures: When updating a method behaviour requires you to change its signature as well, you should instead create an overload so that code calling into that method will still work. You can always manipulate the old method signature to call into the new method signature so that implementation remains consistent.
- Obsolete attribute: You can use this attribute in your code to specify classes or class members that are deprecated and likely to be removed in future versions. This ensures developers utilizing your library are better prepared for breaking changes.
- Optional Method Arguments: When you make previously optional method arguments compulsory or change their default value then all code that does not supply those arguments will need to be updated.

> **NOTE**
>
> Making compulsory arguments optional should have very little effect especially if it doesn't change the method's behaviour.

The easier you make it for your users to upgrade to the new version of your library, the more likely that they will upgrade sooner.

Application Configuration File

As a .NET developer there's a very high chance you've encountered the `app.config` file present in most project types. This simple configuration file can go a long way into improving the rollout of new updates. You should generally design your libraries in such a way that information that is likely to change regularly is stored in the `app.config` file, this way when such information is updated the config file of older versions just needs to be replaced with the new one without the need for recompilation of the library.

Consuming Libraries

As a developer that consumes .NET libraries built by other developers you're most likely aware that a new version of a library might not be fully compatible with your project and you might often find yourself having to update your code to work with those changes.

Lucky for you C# and the .NET ecosystem comes with features and techniques that allow us to easily update our app to work with new versions of libraries that might introduce breaking changes.

Assembly Binding Redirection

You can use the `app.config` file to update the version of a library your app uses. By adding what is called a *binding redirect* you can use the new library version without having to recompile your app. The following example shows how you would update your app's `app.config` file to use the `1.0.1` patch version of `ReferencedLibrary` instead of the `1.0.0` version it was originally compiled with.

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **NOTE**
>
> This approach will only work if the new version of `ReferencedLibrary` is binary compatible with your app. See the Backwards Compatibility section above for changes to look out for when determining compatibility.

new

You use the `new` modifier to hide inherited members of a base class. This is one way derived classes can respond to updates in base classes.

Take the following example:

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Output

```
A base method
A derived method
```

From the example above you can see how `DerivedClass` hides the `MyMethod` method present in `BaseClass`. This means that when a base class in the new version of a library adds a member that already exists in your derived class, you can simply use the `new` modifier on your derived class member to hide the base class member.

When no `new` modifier is specified, a derived class will by default hide conflicting members in a base class, although a compiler warning will be generated the code will still compile. This means that simply adding new members to an existing class makes that new version of your library both source and binary compatible with code that depends on it.

override

The `override` modifier means a derived implementation extends the implementation of a base class member rather than hides it. The base class member needs to have the `virtual` modifier applied to it.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Output

```
Base Method One: Method One
Derived Method One: Derived Method One
```

The `override` modifier is evaluated at compile time and the compiler will throw an error if it doesn't find a virtual member to override.

Your knowledge of the discussed techniques as well as your understanding of what situations to use them will go a long way to boost the ease of transition between versions of a library.

How to (C#)

8/15/2019 • 3 minutes to read • Edit Online

In the How to section of the C# Guide you can find quick answers to common questions. In some cases, articles may be listed in multiple sections. We wanted to make them easy to find for multiple search paths.

General C# concepts

There are several tips and tricks that are common C# developer practices.

- Initialize objects using an object initializer.
- Learn the differences between passing a struct and a class to a method.
- Use operator overloading.
- Implement and call a custom extension method.
- Even C# programmers may want to use the `My` namespace from VB.
- Create a new method for an `enum` type using extension methods.

Class and struct members

You create classes and structs to implement your program. These techniques are commonly used when writing classes or structs.

- Declare auto implemented properties.
- Declare and use read/write properties.
- Define constants.
- Override the `ToString` method to provide string output.
- Define abstract properties.
- Use the xml documentation features to document your code.
- Explicitly implement interface members to keep your public interface concise.
- Explicitly implement members of two interfaces.

Working with collections

These articles help you work with collections of data.

- Initialize a dictionary with a collection initializer.

Working with strings

Strings are the fundamental data type used to display or manipulate text. These articles demonstrate common practices with strings.

- Compare strings.
- Modify the contents of a string.
- Determine if a string represents a number.
- Use `String.Split` to separate strings.
- Combine multiple strings into one.
- Search for text in a string.

Convert between types

You may need to convert an object to a different type.

- Determine if a string represents a number.
- Convert between strings that represent hexadecimal numbers and the number.
- Convert a string to a `DateTime`.
- Convert a byte array to an int.
- Convert a string to a number.
- Use pattern matching, the `as` and `is` operators to safely cast to a different type.
- Define custom type conversions.
- Determine if a type is a nullable value type.
- Convert between nullable and non-nullable value types.

Equality and ordering comparisons

You may create types that define their own rules for equality or define a natural ordering among objects of that type.

- Test for reference-based equality.
- Define value-based equality for a type.

Exception handling

.NET programs report that methods did not successfully complete their work by throwing exceptions. In these articles you'll learn to work with exceptions.

- Handle exceptions using `try` and `catch`.
- Cleanup resources using `finally` clauses.
- Recover from non-CLS (Common Language Specification) exceptions.

Delegates and events

Delegates and events provide a capability for strategies that involve loosely coupled blocks of code.

- Declare, instantiate, and use delegates.
- Combine multicast delegates.

Events provide a mechanism to publish or subscribe to notifications.

- Subscribe and unsubscribe from events.
- Implement events declared in interfaces.
- Conform to .NET Framework guidelines when your code publishes events.
- Raise events defined in base classes from derived classes.
- Implement custom event accessors.

LINQ practices

LINQ enables you to write code to query any data source that supports the LINQ query expression pattern. These articles help you understand the pattern and work with different data sources.

- Query a collection.
- Use lambda expressions in a query.
- Use `var` in query expressions.
- Return subsets of element properties from a query.

- Write queries with complex filtering.
- Sort elements of a data source.
- Sort elements on multiple keys.
- Control the type of a projection.
- Count occurrences of a value in a source sequence.
- Calculate intermediate values.
- Merge data from multiple sources.
- Find the set difference between two sequences.
- Debug empty query results.
- Add custom methods to LINQ queries.

Multiple threads and async processing

Modern programs often use asynchronous operations. These articles will help you learn to use these techniques.

- Improve async performance using `System.Threading.Tasks.Task.WhenAll`.
- Make multiple web requests in parallel using `async` and `await`.
- Use a thread pool.

Command line args to your program

Typically, C# programs have command line arguments. These articles teach you to access and process those command line arguments.

- Retrieve all command line arguments with `for`.

How to: Parse Strings Using String.Split (C# Guide)

1/23/2019 • 2 minutes to read • Edit Online

The String.Split method creates an array of substrings by splitting the input string based on one or more delimiters. It is often the easiest way to separate a string on word boundaries. It is also used to split strings on other specific characters or strings.

> **NOTE**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following code splits a common phrase into an array of strings for each word.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Every instance of a separator character produces a value in the returned array. Consecutive separator characters produce the empty string as a value in the returned array. You can see this in the following example, which uses space as a separator:

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

This behavior makes it easier for formats like comma separated values (CSV) files representing tabular data. Consecutive commas represent a blank column.

You can pass an optional StringSplitOptions.RemoveEmptyEntries parameter to exclude any empty strings in the returned array. For more complicated processing of the returned collection, you can use LINQ to manipulate the result sequence.

String.Split can use multiple separator characters. The following example uses spaces, commas, periods, colons, and tabs, all passed in an array containing these separating characters, to Split. The loop at the bottom of the code displays each of the words in the returned array.

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Consecutive instances of any separator produce the empty string in the output array:

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

String.Split can take an array of strings (character sequences that act as separators for parsing the target string, instead of single characters).

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

You can try these samples by looking at the code in our GitHub repository. Or you can download the samples as a zip file.

See also

- C# Programming Guide
- Strings
- .NET Regular Expressions

How to: Concatenate Multiple Strings (C# Guide)

1/23/2019 • 3 minutes to read • Edit Online

Concatenation is the process of appending one string to the end of another string. You concatenate strings by using the `+` operator. For string literals and string constants, concatenation occurs at compile time; no run-time concatenation occurs. For string variables, concatenation occurs only at run time.

> **NOTE**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following example uses concatenation to split a long string literal into smaller strings in order to improve readability in the source code. These parts are concatenated into a single string at compile time. There is no run-time performance cost regardless of the number of strings involved.

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a " +
"query can only be represented as a string without compile-time type checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

To concatenate string variables, you can use the `+` or `+=` operators, string interpolation or the String.Format, String.Concat, String.Join or StringBuilder.Append methods. The `+` operator is easy to use and makes for intuitive code. Even if you use several `+` operators in one statement, the string content is copied only once. The following code shows examples of using the `+` and `+=` operators to concatenate strings:

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

In some expressions, it's easier to concatenate strings using string interpolation, as the following code shows:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> **NOTE**
> In string concatenation operations, the C# compiler treats a null string the same as an empty string.

Other method to concatenate strings is String.Format. This method works well when you are building a string from a small number of component strings.

In other cases you may be combining strings in a loop, where you don't know how many source strings you are combining, and the actual number of source strings may be quite large. The StringBuilder class was designed for these scenarios. The following code uses the Append method of the StringBuilder class to concatenate strings.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

You can read more about the reasons to choose string concatenation or the `StringBuilder` class

Another option to join strings from a collection is to use String.Concat method. Use String.Join method if source strings should be separated by a delimeter. The following code combines an array of words using both methods:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

At last, you can use LINQ and the Enumerable.Aggregate method to join strings from a collection. This method combines the source strings using a lambda expression. The lambda expression does the work to add each string to the existing accumulation. The following example combines an array of words by adding a space between each word in the array:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

You can try these samples by looking at the code in our GitHub repository. Or you can download the samples as a zip file.

See also

- String
- StringBuilder
- C# Programming Guide
- Strings

Parsing Date and Time Strings in .NET

6/20/2019 • 6 minutes to read • Edit Online

Parsing strings to convert them to DateTime objects requires you to specify information about how the dates and times are represented as text. Different cultures use different orders for day, month, and year. Some time representations use a 24-hour clock, others specify "AM" and "PM." Some applications need only the date. Others need only the time. Still others need to specify both the date and the time. The methods that convert strings to DateTime objects enable you to provide detailed information about the formats you expect and the elements of a date and time your application needs. There are three subtasks to correctly converting text into a DateTime:

1. You must specify the expected format of the text representing a date and time.
2. You may specify the culture for the format of a date time.
3. You may specify how missing components in the text representation are set in the date and time.

The Parse and TryParse methods convert many common representations of a date and time. The ParseExact and TryParseExact methods convert a string representation that conforms to the pattern specified by a date and time format string. (See the articles on standard date and time format strings and custom date and time format strings for details.)

The current DateTimeFormatInfo object provides more control over how text should be interpreted as a date and time. Properties of a DateTimeFormatInfo describe the date and time separators, and the names of months, days, and eras, and the format for the "AM" and "PM" designations. The current thread culture provides a DateTimeFormatInfo that represents the current culture. If you want a specific culture or custom settings, you specify the IFormatProvider parameter of a parsing method. For the IFormatProvider parameter, specify a CultureInfo object, which represents a culture, or a DateTimeFormatInfo object.

The text representing a date or time may be missing some information. For example, most people would assume the date "March 12" represents the current year. Similarly, "March 2018" represents the month of March in the year 2018. Text representing time often does only includes hours, minutes, and an AM/PM designation. Parsing methods handle this missing information by using reasonable defaults:

- When only the time is present, the date portion uses the current date.
- When only the date is present, the time portion is midnight.
- When the year isn't specified in a date, the current year is used.
- When the day of the month isn't specified, the first of the month is used.

If the date is present in the string, it must include the month and one of the day or year. If the time is present, it must include the hour, and either the minutes or the AM/PM designator.

You can specify the NoCurrentDateDefault constant to override these defaults. When you use that constant, any missing year, month, or day properties are set to the value `1`. The last example using Parse demonstrates this behavior.

In addition to a date and a time component, the string representation of a date and time can include an offset that indicates how much the time differs from Coordinated Universal Time (UTC). For example, the string "2/14/2007 5:32:00 -7:00" defines a time that is seven hours earlier than UTC. If an offset is omitted from the string representation of a time, parsing returns a DateTime object with its Kind property set to DateTimeKind.Unspecified. If an offset is specified, parsing returns a DateTime object with its Kind property set to DateTimeKind.Local and its value adjusted to the local time zone of your machine. You can modify this behavior by using a DateTimeStyles value with the parsing method.

The format provider is also used to interpret an ambiguous numeric date. It is not clear which components of the

date represented by the string "02/03/04" are the month, day, and year. The components are interpreted according to the order of similar date formats in the format provider.

Parse

The following example illustrates the use of the DateTime.Parse method to convert a `string` into a DateTime. This example uses the culture associated with the current thread. If the CultureInfo associated with the current culture cannot parse the input string, a FormatException is thrown.

> **TIP**
> All the C# samples in this article run in your browser. Press the **Run** button to see the output. You can also edit them to experiment yourself.

> **NOTE**
> These examples are available in the GitHub docs repo for both C# and VB. Or you can download the project as a zipfile for C# or VB.

```
string dateInput = "Jan 1, 2009";
DateTime parsedDate = DateTime.Parse(dateInput);
Console.WriteLine(parsedDate);
// Displays the following output on a system whose culture is en-US:
//       1/1/2009 12:00:00 AM
```

```
Dim MyString As String = "Jan 1, 2009"
Dim MyDateTime As DateTime = DateTime.Parse(MyString)
Console.WriteLine(MyDateTime)
' Displays the following output on a system whose culture is en-US:
'       1/1/2009 12:00:00 AM
```

You can also explicitly define the culture whose formatting conventions are used when you parse a string. You specify one of the standard DateTimeFormatInfo objects returned by the CultureInfo.DateTimeFormat property. The following example uses a format provider to parse a German string into a DateTime. It creates a CultureInfo representing the `de-DE` culture. That `CultureInfo` object ensures successful parsing of this particular string. This precludes whatever setting is in the CurrentCulture of the CurrentThread.

```
CultureInfo MyCultureInfo = new CultureInfo("de-DE");
string MyString = "12 Juni 2008";
DateTime MyDateTime = DateTime.Parse(MyString, MyCultureInfo);
Console.WriteLine(MyDateTime);
// The example displays the following output:
//       6/12/2008 12:00:00 AM
```

```
Dim MyCultureInfo As CultureInfo = New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo)
Console.WriteLine(MyDateTime)
' The example displays the following output:
'       6/12/2008 12:00:00 AM
```

However, although you can use overloads of the Parse method to specify custom format providers, the method does not support parsing non-standard formats. To parse a date and time expressed in a non-standard format, use

the ParseExact method instead.

The following example uses the DateTimeStyles enumeration to specify that the current date and time information should not be added to the DateTime for unspecified fields.

```
CultureInfo MyCultureInfo = new CultureInfo("de-DE");
string MyString = "12 Juni 2008";
DateTime MyDateTime = DateTime.Parse(MyString, MyCultureInfo,
                                     DateTimeStyles.NoCurrentDateDefault);
Console.WriteLine(MyDateTime);
// The example displays the following output if the current culture is en-US:
//      6/12/2008 12:00:00 AM
```

```
Dim MyCultureInfo As CultureInfo = New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo,
                           DateTimeStyles.NoCurrentDateDefault)
Console.WriteLine(MyDateTime)
' The example displays the following output if the current culture is en-US:
'       6/12/2008 12:00:00 AM
```

ParseExact

The DateTime.ParseExact method converts a string to a DateTime object if it conforms to one of the specified string patterns. When a string that is not one of the forms specified is passed to this method, a FormatException is thrown. You can specify one of the standard date and time format specifiers or a combination of the custom format specifiers. Using the custom format specifiers, it is possible for you to construct a custom recognition string. For an explanation of the specifiers, see the topics on standard date and time format strings and custom date and time format strings.

In the following example, the DateTime.ParseExact method is passed a string object to parse, followed by a format specifier, followed by a CultureInfo object. This ParseExact method can only parse strings that follow the long date pattern in the `en-US` culture.

```
CultureInfo MyCultureInfo = new CultureInfo("en-US");
string[] MyString = { " Friday, April 10, 2009", "Friday, April 10, 2009" };
foreach (string dateString in MyString)
{
    try
    {
        DateTime MyDateTime = DateTime.ParseExact(dateString, "D", MyCultureInfo);
        Console.WriteLine(MyDateTime);
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to parse '{0}'", dateString);
    }
}
// The example displays the following output:
//       Unable to parse ' Friday, April 10, 2009'
//       4/10/2009 12:00:00 AM
```

```
Dim MyCultureInfo As CultureInfo = New CultureInfo("en-US")
Dim MyString() As String = {" Friday, April 10, 2009", "Friday, April 10, 2009"}
For Each dateString As String In MyString
    Try
        Dim MyDateTime As DateTime = DateTime.ParseExact(dateString, "D",
                                                         MyCultureInfo)
        Console.WriteLine(MyDateTime)
    Catch e As FormatException
        Console.WriteLine("Unable to parse '{0}'", dateString)
    End Try
Next
' The example displays the following output:
'       Unable to parse ' Friday, April 10, 2009'
'       4/10/2009 12:00:00 AM
```

Each overload of the Parse and ParseExact methods also has an IFormatProvider parameter that provides culture-specific information about the formatting of the string. This IFormatProvider object is a CultureInfo object that represents a standard culture or a DateTimeFormatInfo object that is returned by the CultureInfo.DateTimeFormat property. ParseExact also uses an additional string or string array argument that defines one or more custom date and time formats.

See also

- Parsing Strings
- Formatting Types
- Type Conversion in .NET
- Standard date and time formats
- Custom date and time format strings

How to: search strings

3/1/2019 • 4 minutes to read • Edit Online

You can use two main strategies to search for text in strings. Methods of the String class search for specific text. Regular expressions search for patterns in text.

> **NOTE**
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The string type, which is an alias for the System.String class, provides a number of useful methods for searching the contents of a string. Among them are Contains, StartsWith, EndsWith, IndexOf, LastIndexOf. The System.Text.RegularExpressions.Regex class provides a rich vocabulary to search for patterns in text. In this article, you learn these techniques and how to choose the best method for your needs.

Does a string contain text?

The String.Contains, String.StartsWith and String.EndsWith methods search a string for specific text. The following example shows each of these methods and a variation that uses a case insensitive search:

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Starts with \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

The preceding example demonstrates an important point for using these methods. Searches are **case-sensitive** by default. You use the StringComparison.CurrentCultureIgnoreCase enum value to specify a case insensitive search.

Where does the sought text occur in a string?

The IndexOf and LastIndexOf methods also search for text in strings. These methods return the location of the text being sought. If the text isn't found, they return `-1`. The following example shows a search for the first and last occurrence of the word "methods" and displays the text in between.

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");
```

Finding specific text using regular expressions

The System.Text.RegularExpressions.Regex class can be used to search strings. These searches can range in complexity from simple to complicated text patterns.

The following code example searches for the word "the" or "their" in a sentence, ignoring case. The static method Regex.IsMatch performs the search. You give it the string to search and a search pattern. In this case, a third argument specifies case-insensitive search. For more information, see System.Text.RegularExpressions.RegexOptions.

The search pattern describes the text you search for. The following table describes each element of the search pattern. (The table below uses the single `\` which must be escaped as `\\` in a C# string).

PATTERN	MEANING
the	match the text "the"
(eir)?	match 0 or 1 occurrence of "eir"
\s	match a white-space character

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

TIP

The `string` methods are usually better choices when you are searching for an exact string. Regular expressions are better when you are searching for some pattern is a source string.

Does a string follow a pattern?

The following code uses regular expressions to validate the format of each string in an array. The validation requires that each string have the form of a telephone number in which three groups of digits are separated by dashes, the first two groups contain three digits, and the third group contains four digits. The search pattern uses the regular expression `^\\d{3}-\\d{3}-\\d{4}$`. For more information, see Regular Expression Language - Quick Reference.

PATTERN	MEANING
^	matches the beginning of the string
\d{3}	matches exactly 3 digit characters
-	matches the '-' character
\d{3}	matches exactly 3 digit characters
-	matches the '-' character
\d{4}	matches exactly 4 digit characters
$	matches the end of the string

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

This single search pattern matches many valid strings. Regular expressions are better to search for or validate against a pattern, rather than a single text string.

You can try these samples by looking at the code in our GitHub repository. Or you can download the samples as a zip file.

See also

- C# Programming Guide
- Strings
- LINQ and Strings
- System.Text.RegularExpressions.Regex
- .NET Framework Regular Expressions
- Regular Expression Language - Quick Reference
- Best practices for using strings in .NET

How to: Modify string contents in C#

6/20/2019 • 5 minutes to read • Edit Online

This article demonstrates several techniques to produce a `string` by modifying an existing `string`. All the techniques demonstrated return the result of the modifications as a new `string` object. To clearly demonstrate this, the examples all store the result in a new variable. You can then examine both the original `string` and the `string` resulting from the modification when you run each example.

> **NOTE**
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

There are several techniques demonstrated in this article. You can replace existing text. You can search for patterns and replace matching text with other text. You can treat a string as a sequence of characters. You can also use convenience methods that remove white space. You should choose the techniques that most closely match your scenario.

Replace text

The following code creates a new string by replacing existing text with a substitute.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

The preceding code demonstrates this *immutable* property of strings. You can see in the preceding example that the original string, `source`, is not modified. The String.Replace method creates a new `string` containing the modifications.

The Replace method can replace either strings or single characters. In both cases, every occurrence of the sought text is replaced. The following example replaces all ' ' characters with '_':

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

The source string is unchanged, and a new string is returned with the replacement.

Trim white space

You can use the String.Trim, String.TrimStart, and String.TrimEnd methods to remove any leading or trailing white

space. The following code shows an example of each. The source string does not change; these methods return a new string with the modified contents.

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Remove text

You can remove text from a string using the String.Remove method. This method removes a number of characters starting at a specific index. The following example shows how to use String.IndexOf followed by Remove to remove text from a string:

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Replace matching patterns

You can use regular expressions to replace text matching patterns with new text, possibly defined by a pattern. The following example uses the System.Text.RegularExpressions.Regex class to find a pattern in a source string and replace it with proper capitalization. The Regex.Replace(String, String, MatchEvaluator, RegexOptions) method takes a function that provides the logic of the replacement as one of its arguments. In this example, that function, `LocalReplaceMatchCase` is a **local function** declared inside the sample method. `LocalReplaceMatchCase` uses the System.Text.StringBuilder class to build the replacement string with proper capitalization.

Regular expressions are most useful for searching and replacing text that follows a pattern, rather than known text. See How to: search strings for more details. The search pattern, "the\s" searches for the word "the" followed by a white-space character. That part of the pattern ensures that it doesn't match "there" in the source string. For more information on regular expression language elements, see Regular Expression Language - Quick Reference.

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

The StringBuilder.ToString method returns an immutable string with the contents in the StringBuilder object.

Modifying individual characters

You can produce a character array from a string, modify the contents of the array, and then create a new string from the modified contents of the array.

The following example shows how to replace a set of characters in a string. First, it uses the ToCharArray() method to create an array of characters. It uses the IndexOf method to find the starting index of the word "fox." The next three characters are replaced with a different word. Finally, a new string is constructed from the updated character array.

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Unsafe modifications to string

Using **unsafe** code, you can modify a string "in place" after it has been created. Unsafe code bypasses many of the features of .NET designed to minimize certain types of bugs in code. You need to use unsafe code to modify a string in place because the string class is designed as an **immutable** type. Once it has been created, its value does not change. Unsafe code circumvents this property by accessing and modifying the memory used by a `string` without using normal `string` methods. The following example is provided for those rare situations where you

want to modify a string in-place using unsafe code. The example shows how to use the `fixed` keyword. The `fixed` keyword prevents the garbage collector (GC) from moving the string object in memory while code accesses the memory using the unsafe pointer. It also demonstrates one possible side effect of unsafe operations on strings that results from the way that the C# compiler stores (interns) strings internally. In general, you shouldn't use this technique unless it is absolutely necessary. You can learn more in the articles on unsafe and fixed. The API reference for Intern includes information on string interning.

```
unsafe
{
    // Compiler will store (intern)
    // these strings in same location.
    string helloOne = "Hello";
    string helloTwo = "Hello";

    // Change one string using unsafe code.
    fixed (char* p = helloOne)
    {
        p[0] = 'C';
    }

    //  Both strings have changed.
    Console.WriteLine(helloOne);
    Console.WriteLine(helloTwo);
}
```

You can try these samples by looking at the code in our GitHub repository. Or you can download the samples as a zip file.

See also

- .NET Framework Regular Expressions
- Regular Expression Language - Quick Reference

How to compare strings in C#

3/29/2019 • 10 minutes to read • Edit Online

You compare strings to answer one of two questions: "Are these two strings equal?" or "In what order should these strings be placed when sorting them?"

Those two questions are complicated by factors that affect string comparisons:

- You can choose an ordinal or linguistic comparison.
- You can choose if case matters.
- You can choose culture-specific comparisons.
- Linguistic comparisons are culture and platform-dependent.

> **NOTE**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

When you compare strings, you define an order among them. Comparisons are used to sort a sequence of strings. Once the sequence is in a known order, it is easier to search, both for software and for humans. Other comparisons may check if strings are the same. These sameness checks are similar to equality, but some differences, such as case differences, may be ignored.

Default ordinal comparisons

The most common operations:

- String.CompareTo
- String.Equals
- String.Equality

use an ordinal comparison, a case-sensitive comparison, and use the current culture. The results are shown in the following example:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal")}");
```

The default ordinal comparison does not take linguistic rules into account when comparing strings. It compares the binary value of each Char object in two strings. As a result, the default ordinal comparison is also case-sensitive.

Note that the test for equality with String.Equals and String.Equality differs from string comparison using the

String.CompareTo and Compare(String, String) methods. While the tests for equality perform a case-sensitive ordinal comparison, the comparison method perform a case-sensitive, culture-sensitive comparison using the current culture. Because the default comparison methods often perform different types of comparisons, we recommend that you always make the intent of your code clear by calling an overload that explicitly specifies the type of comparison to perform.

Case-insensitive ordinal comparisons

The String.Equals(String, StringComparison) method enables you to specify a StringComparison value of StringComparison.OrdinalIgnoreCase for a case-insensitive ordinal comparison. There is also a static String.Compare(String, String, StringComparison) method that performs a case-insensitive ordinal comparison if you specify a value of StringComparison.OrdinalIgnoreCase for the StringComparison argument. These are shown in the following code:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType: StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not
equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

When performing a case-insensitive ordinal comparison, these methods use the casing conventions of the invariant culture.

Linguistic comparisons

Strings can also be ordered using linguistic rules for the current culture. This is sometimes referred to as "word sort order." When you perform a linguistic comparison, some nonalphanumeric Unicode characters might have special weights assigned. For example, the hyphen "-" may have a very small weight assigned to it so that "co-op" and "coop" appear next to each other in sort order. In addition, some Unicode characters may be equivalent to a sequence of Char instances. The following example uses the phrase "They dance in the street." in German with the "ss" (U+0073 U+0073) in one string and 'ß' (U+00DF) in another. Linguistically (in Windows), "ss" is equal to the German Esszet: 'ß' character in both the "en-US" and "de-DE" cultures.

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

This sample demonstrates the operating system-dependent nature of linguistic comparisons. The host for the interactive window is a Linux host. The linguistic and ordinal comparisons produce the same results. If you ran this same sample on a Windows host, you would see the following output:

```
<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison
```

On Windows, the sort order of "cop", "coop", and "co-op" change when you change from a linguistic comparison to an ordinal comparison. The two German sentences also compare differently using the different comparison types.

Comparisons using specific cultures

This sample stores CultureInfo objects for the en-US and de-DE cultures. The comparisons are performed using a CultureInfo object to ensure a culture-specific comparison.

The culture used affects linguistic comparisons. The following example shows the results of comparing the two German sentences using the "en-US" culture and the "de-DE" culture:

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Culture-sensitive comparisons are typically used to compare and sort strings input by users with other strings input by users. The characters and sorting conventions of these strings might vary depending on the locale of the user's computer. Even strings that contain identical characters might sort differently depending on the culture of the current thread. In addition, try this sample code locally on a Windows machine, and you will the following results:

```
<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison
```

Linguistic comparisons are dependent on the current culture, and are OS dependent. You must take that into account when you work with string comparisons.

Linguistic sorting and searching strings in arrays

The following examples show how to sort and search for strings in an array using a linguistic comparison dependent on the current culture. You use the static Array methods that take a System.StringComparer parameter.

This example shows how to sort an array of strings using the current culture:

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Once the array is sorted, you can search for entries using a binary search. A binary search starts in the middle of the collection to determine which half of the collection would contain the sought string. Each subsequent comparison subdivides the remaining part of the collection in half. The array is sorted using the StringComparer.CurrentCulture. The local function `ShowWhere` displays information about where the string was found. If the string was not found, the returned value indicates where it would be if it were found.

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Ordinal sorting and searching in collections

The following code uses the System.Collections.Generic.List<T> collection class to store strings. The strings are sorted using the List<T>.Sort method. This method needs a delegate that compares and orders two strings. The String.CompareTo method provides that comparison function. Run the sample and observe the order. This sort operation uses an ordinal case sensitive sort. You would use the static String.Compare methods to specify different comparison rules.

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Once sorted, the list of strings can be searched using a binary search. The following sample shows how to search the sorted listed using the same comparison function. The local function `ShowWhere` shows where the sought text is or would be:

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Always make sure to use the same type of comparison for sorting and searching. Using different comparison types for sorting and searching produces unexpected results.

Collection classes such as System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue>, and System.Collections.Generic.List<T> have constructors that take a System.StringComparer parameter when the type of the elements or keys is `string`. In general, you should use these constructors whenever possible, and specify either StringComparer.Ordinal or StringComparer.OrdinalIgnoreCase.

Reference equality and string interning

None of the samples have used ReferenceEquals. This method determines if two strings are the same object. This can lead to inconsistent results in string comparisons. The following example demonstrates the *string interning* feature of C#. When a program declares two or more identical string variables, the compiler stores them all in the same location. By calling the ReferenceEquals method, you can see that the two strings actually refer to the same object in memory. Use the String.Copy method to avoid interning. After the copy has been made, the two strings have different storage locations, even though they have the same value. Run the following sample to show that strings `a` and `b` are *interned* meaning they share the same storage. The strings `a` and `c` are not.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

When you test for equality of strings, you should use the methods that explicitly specify what kind of comparison you intend to perform. Your code is much more maintainable and readable. Use the overloads of the methods of the System.String and System.Array classes that take a StringComparison enumeration parameter. You specify which type of comparison to perform. Avoid using the `==` and `!=` operators when you test for equality. The String.CompareTo instance methods always perform an ordinal case-sensitive comparison. They are primarily suited for ordering strings alphabetically.

You can intern a string or retrieve a reference to an existing interned string by calling the String.Intern method. To determine whether a string is interned, call the String.IsInterned method.

See also

- System.Globalization.CultureInfo
- System.StringComparer
- Strings
- Comparing Strings
- Globalizing and Localizing Applications

How to: safely cast by using pattern matching and the is and as operators

8/17/2019 • 4 minutes to read • Edit Online

Because objects are polymorphic, it's possible for a variable of a base class type to hold a derived type. To access the derived type's instance members, it's necessary to cast the value back to the derived type. However, a cast creates the risk of throwing an InvalidCastException. C# provides pattern matching statements that perform a cast conditionally only when it will succeed. C# also provides the is and as operators to test if a value is of a certain type.

The following code demonstrates the pattern matching `is` statement. It contains methods that test a method argument to determine if it is one of a possible set of derived types:

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        Giraffe g = new Giraffe();
        FeedMammals(g);

        TestForMammals(g);

        SuperNova sn = new SuperNova();
        TestForMammals(sn);
    }

    static void FeedMammals(Animal a)
    {
        // Use the is operator to verify the type
        // before performing a cast.
        if (a is Mammal m)
        {
            m.Eat();
        }
    }

    static void TestForMammals(object o)
    {
        // Alternatively, use the as operator and test for null
        // before referencing the variable.
        if (o is Mammal m)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            // variable 'm' is not in scope here, and can't be used.
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }
}
```

The preceding sample demonstrates a few features of pattern matching syntax. The `if (a is Mammal m)` and `if (o is Mammal m)` statements combine the test with an initialization assignment. The assignment occurs only when the test succeeds. The variable `m` is only in scope in the embedded `if` statement where it has been assigned. You cannot access `m` later in the same method. Try it in the interactive window.

You can also use the same syntax for testing if a nullable type has a value, as shown in the following sample code:

```
class Program
{
    static void Main(string[] args)
    {
        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        PatternMatchingNullable(i);

        int? j = null;
        PatternMatchingNullable(j);

        double d = 9.78654;
        PatternMatchingNullable(d);

        PatternMatchingSwitch(i);
        PatternMatchingSwitch(j);
        PatternMatchingSwitch(d);
    }

    static void PatternMatchingNullable(System.ValueType val)
    {
        if (val is int j) // Nullable types are not allowed in patterns
        {
            Console.WriteLine(j);
        }
        else if (val is null) // If val is a nullable type with no value, this expression is true
        {
            Console.WriteLine("val is a nullable type with the null value");
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }

    static void PatternMatchingSwitch(System.ValueType val)
    {
        switch (val)
        {
            case int number:
                Console.WriteLine(number);
                break;
            case long number:
                Console.WriteLine(number);
                break;
            case decimal number:
                Console.WriteLine(number);
                break;
            case float number:
                Console.WriteLine(number);
                break;
            case double number:
                Console.WriteLine(number);
                break;
            case null:
                Console.WriteLine("val is a nullable type with the null value");
                break;
            default:
                Console.WriteLine("Could not convert " + val.ToString());
                break;
        }
    }
}
```

The preceding sample demonstrates other features of pattern matching to use with conversions. You can test a variable for the null pattern by checking specifically for the `null` value. When the runtime value of the variable is `null`, an `is` statement checking for a type always returns `false`. The pattern matching `is` statement doesn't allow a nullable value type, such as `int?` or `Nullable<int>`, but you can test for any other value type.

The preceding sample also shows how you use the pattern matching `is` expression in a `switch` statement where the variable may be one of many different types.

If you want to test if a variable is a given type, but not assign it to a new variable, you can use the `is` and `as` operators for reference types and nullable types. The following code shows how to use the `is` and `as` statements that were part of the C# language before pattern matching was introduced to test if a variable is of a given type:

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }


class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        UseAsOperator(sn);

        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        UseAsWithNullable(i);

        double d = 9.78654;
        UseAsWithNullable(d);
    }

    static void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    static void UsePatternMatchingIs(Animal a)
    {
        if (a is Mammal m)
```

```
        if (a is Mammal m)
        {
            m.Eat();
        }
    }

    static void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }

    static void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }
}
```

As you can see by comparing this code with the pattern matching code, the pattern matching syntax provides more robust features by combining the test and the assignment in a single statement. Use the pattern matching syntax whenever possible.

You can try these samples by looking at the code in our GitHub repository. Or you can download the samples as a zip file.

The .NET Compiler Platform SDK

3/19/2019 • 6 minutes to read • Edit Online

Compilers build a detailed model of application code as they validate the syntax and semantics of that code. They use this model to build the executable output from the source code. The .NET Compiler Platform SDK provides access to this model. Increasingly, we rely on integrated development environment (IDE) features such as IntelliSense, refactoring, intelligent rename, "Find all references," and "Go to definition" to increase our productivity. We rely on code analysis tools to improve our code quality, and code generators to aid in application construction. As these tools get smarter, they need access to more and more of the model that only compilers create as they process application code. This is the core mission of the Roslyn APIs: opening up the black boxes and allowing tools and end users to share in the wealth of information compilers have about our code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms: APIs that you can use for code-related tasks in your tools and applications.

.NET Compiler Platform SDK concepts

The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and VB languages, and embedding of C# and VB in domain specific languages.

The .NET Compiler Platform SDK enables you to build ***analyzers*** and ***code fixes*** that find and correct coding mistakes. ***Analyzers*** understand the syntax and structure of code and detect practices that should be corrected. ***Code fixes*** provide one or more suggested fixes for addressing coding mistakes found by analyzers. Typically, an analyzer and the associated code fixes are packaged together in a single project.

Analyzers and code fixes use static analysis to understand code. They do not run the code or provide other testing benefits. They can, however, point out practices that often lead to bugs, unmaintainable code, or standard guideline validation.

The .NET Compiler Platform SDK provides a single set of APIs that enable you to examine and understand a C# or Visual Basic codebase. Because you can use this single codebase, you can write analyzers and code fixes more easily by leveraging the syntactic and semantic analysis APIs provided by the .NET Compiler Platform SDK. Freed from the large task of replicating the analysis done by the compiler, you can concentrate on the more focused task of finding and fixing common coding errors for your project or library.

A smaller benefit is that your analyzers and code fixes are smaller and use much less memory when loaded in Visual Studio than they would if you wrote your own codebase to understand the code in a project. By leveraging the same classes used by the compiler and Visual Studio, you can create your own static analysis tools. This means your team can use analyzers and code fixes without a noticeable impact on the IDE's performance.

There are three main scenarios for writing analyzers and code fixes:

1. *Enforce team coding standards*
2. *Provide guidance with library packages*
3. *Provide general guidance*

Enforce team coding standards

Many teams have coding standards that are enforced through code reviews with other team members. Analyzers and code fixes can make this process much more efficient. Code reviews happen when a developer shares their

work with others on the team. The developer will have invested all the time needed to complete a new feature before getting any comments. Weeks may go by while the developer reinforces habits that don't match the team's practices.

Analyzers run as a developer writes code. The developer gets immediate feedback that encourages following the guidance immediately. The developer builds habits to write compliant code as soon as they begin prototyping. When the feature is ready for humans to review, all the standard guidance has been enforced.

Teams can build analyzers and code fixes that look for the most common practices that violate team coding practices. These can be installed on each developer's machine to enforce the standards.

Provide guidance with library packages

There are a wealth of libraries available for .NET developers on NuGet. Some of these come from Microsoft, some from third-party companies, and others from community members and volunteers. These libraries get more adoption and higher reviews when developers can succeed with those libraries.

In addition to providing documentation, you can provide analyzers and code fixes that find and correct common mis-uses of your library. These immediate corrections will help developers succeed more quickly.

You can package analyzers and code fixes with your library on NuGet. In that scenario, every developer who installs your NuGet package will also install the analyzer package. All developers using your library will immediately get guidance from your team in the form of immediate feedback on mistakes and suggested corrections.

Provide general guidance

The .NET developer community has discovered through experience patterns that work well and patterns that are best avoided. Several community members have created analyzers that enforce those recommended patterns. As we learn more, there is always room for new ideas.

These analyzers can be uploaded to the Visual Studio Marketplace and downloaded by developers using Visual Studio. Newcomers to the language and the platform learn accepted practices quickly and become productive earlier in their .NET journey. As these become more widely used, the community adopts these practices.

Next steps

The .NET Compiler Platform SDK includes the latest language object models for code generation, analysis, and refactoring. This section provides a conceptual overview of the .NET Compiler Platform SDK. Further details can be found in the quickstarts, samples and tutorials sections.

You can learn more about the concepts in the .NET Compiler Platform SDK in these five topics:

- Explore code with the syntax visualizer
- Understand the compiler API model
- Work with syntax
- Work with semantics
- Work with a workspace

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

C# programming guide

8/19/2019 • 2 minutes to read • Edit Online

This section provides detailed information on key C# language features and features accessible to C# through the .NET Framework.

Most of this section assumes that you already know something about C# and general programming concepts. If you are a complete beginner with programming or with C#, you might want to visit the Introduction to C# Tutorials or Getting Started with C# interactive tutorial, where no prior programming knowledge is required.

For information about specific keywords, operators and preprocessor directives, see C# Reference. For information about the C# Language Specification, see C# Language Specification.

Program sections

Inside a C# Program

Main() and Command-Line Arguments

Language Sections

Statements, Expressions, and Operators

Types

Classes and Structs

Interfaces

Enumeration Types

Delegates

Arrays

Strings

Properties

Indexers

Events

Generics

Iterators

LINQ Query Expressions

Lambda Expressions

Namespaces

Nullable Types

Unsafe Code and Pointers

XML Documentation Comments

Platform Sections

Application Domains

Assemblies in .NET

Attributes

Collections

Exceptions and Exception Handling

File System and the Registry (C# Programming Guide)

Interoperability

Reflection

See also

- C# Reference
- C#

C# Reference

8/19/2019 • 2 minutes to read • Edit Online

This section provides reference material about C# keywords, operators, special characters, preprocessor directives, compiler options, and compiler errors and warnings.

In This Section

C# Keywords

Provides links to information about C# keywords and syntax.

C# Operators

Provides links to information about C# operators and syntax.

C# Special Characters

Provides links to information about special contextual characters in C# and their usage.

C# Preprocessor Directives

Provides links to information about compiler commands for embedding in C# source code.

C# Compiler Options

Includes information about compiler options and how to use them.

C# Compiler Errors

Includes code snippets that demonstrate the cause and correction of C# compiler errors and warnings.

C# Language Specification

The C# 6.0 language specification. This is a draft proposal for the C# 6.0 language. Version 5.0 has been released in December 2017 as the Standard ECMA-334 5th Edition document.

The features that have been implemented in C# versions after 6.0 are represented in language specification proposals. These documents describe the deltas to the language spec in order to add these new features.

C# 7.0 Language Proposals

There are a number of new features implemented in C# 7.0. They include pattern matching, local functions, out variable declarations, throw expressions, binary literals, and digit separators. This folder contains the specifications for each of those features.

C# 7.1 Language Proposals

There are new features added in C# 7.1. First, you can write a `Main` method that returns `Task` or `Task<int>`. This enables you to add the `async` modifier to `Main`. The `default` expression can be used without a type in locations where the type can be inferred. Also, tuple member names can be inferred. Finally, pattern matching can be used with generics.

C# 7.2 Language Proposals

C# 7.2 added a number of small features. You can pass arguments by readonly reference using the `in` keyword. There are a number of low-level changes to support compile-time safety for `Span` and related types. You can use named arguments where later arguments are positional, in some situations. The `private protected` access modifier enables you to specify that callers are limited to derived types implemented in the same assembly. The `?:` operator can resolve to a reference to a variable. You can also format hexadecimal and binary numbers using a leading digit separator.

C# 7.3 Language Proposals

C# 7.3 is another point release that includes several small updates. You can use new constraints on generic type parameters. Other changes make it easier to work with `fixed` fields, including using `stackalloc` allocations. Local variables declared with the `ref` keyword may be reasssigned to refer to new storage. You can place attributes on auto-implemented properties that target the compiler-generated backing field. Expression variables can be used in initializers. Tuples can be compared for equality (or inequality). There have also been some improvements to overload resolution.

C# 8.0 Language Proposals C# 8.0 is available in preview. The following proposals are the current versions of the specifications for those features. Some are more complete; some are still a work in progress. The features that have shipped in previews include nullable reference types, recursive pattern matching, async streams, ranges and indexes, pattern based using and using declarations, and null coalescing assignment.

Related Sections

C# Guide
Provides a portal to Visual C# documentation.

Using the Visual Studio Development Environment for C#
Provides links to conceptual and task topics that describe the IDE and Editor.

C# Programming Guide
Includes information about how to use the C# programming language.

C# Walkthroughs

8/19/2019 • 2 minutes to read • Edit Online

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

This section contains links to C# programming walkthroughs.

In This Section

Accessing the Web by Using Async and Await
Shows how to create an asynchronous solution by using async and await.

Creating a Windows Runtime Component in C# or Visual Basic and Calling it from JavaScript
Shows how to create a Windows Runtime type, package it in a Windows Runtime component, and then call the component from a Windows 8.x Store app that's built for Windows by using JavaScript.

Office Programming (C# and Visual Basic)
Shows how to create an Excel workbook and a Word document by using C# and Visual Basic.

Creating and Using Dynamic Objects (C# and Visual Basic)
Shows how to create a custom object that dynamically exposes the contents of a text file, and how to create a project that uses the `IronPython` library.

Authoring a Composite Control with Visual C#
Demonstrates creating a simple composite control and extending its functionality through inheritance.

Creating a Windows Forms Control that Takes Advantage of Visual Studio Design-Time Features
Illustrates how to create a custom designer for a custom control.

Inheriting from a Windows Forms Control with Visual C#
Demonstrates creating a simple inherited button control. This button inherits functionality from the standard Windows Forms button and exposes a custom member.

Debugging Custom Windows Forms Controls at Design Time
Describes how to debug the design-time behavior of your custom control.

Performing Common Tasks Using Smart Tags on Windows Forms Controls
Demonstrates some of the commonly performed tasks such as adding or removing a tab on a `TabControl`, docking a control to its parent, and changing the orientation of a `SplitContainer` control.

Writing Queries in C# (LINQ)
Demonstrates the C# language features that are used to write LINQ query expressions.

Manipulating Data (C#) (LINQ to SQL)
Describes a LINQ to SQL scenario for adding, modifying, and deleting data in a database.

Simple Object Model and Query (C#) (LINQ to SQL)
Demonstrates how to create an entity class and a simple query to filter the entity class.

Using Only Stored Procedures (C#) (LINQ to SQL)
Demonstrates how to use LINQ to SQL to access data by executing only stored procedures.

Querying Across Relationships (C#) (LINQ to SQL)
Demonstrates the use of LINQ to SQL associations to represent foreign-key relationships in a database.

Writing a Visualizer in C#

Shows how to write a simple visualizer by using C#.

Related Sections

Deployment Samples and Walkthroughs

Provides step-by-step examples of common deployment scenarios.

See also

- C# Programming Guide
- Visual Studio Samples

F# Guide

7/30/2019 • 2 minutes to read • Edit Online

The F# guide provides many resources to learn the F# language.

Learning F#

What is F# describes what the F# language is and what programming in it is like, with short code samples. This is recommended if you are new to F#.

Tour of F# gives an overview of major language features with lots of code samples. This is recommended if you are interested in seeing core F# features in action.

Get started with F# in Visual Studio if you're on Windows and want the full Visual Studio IDE (Integrated Development Environment) experience.

Get started with F# in Visual Studio for Mac if you're on macOS and want to use a Visual Studio IDE.

Get Started with F# in Visual Studio Code if you want a lightweight, cross-platform, and feature-packed IDE experience.

Get started with F# with the .NET Core CLI if you want to use command-line tools.

Get started with F# and Xamarin for mobile programming with F#.

F# for Azure Notebooks is a tutorial for learning F# in a free, hosted Jupyter Notebook.

References

F# Language Reference is the official, comprehensive reference for all F# language features. Each article explains the syntax and shows code samples. You can use the filter bar in the table of contents to find specific articles.

F# Core Library Reference is the API reference for the F# Core Library.

Additional guides

F# for Fun and Profit is a comprehensive and very detailed book on learning F#. Its contents and author are beloved by the F# community. The target audience is primarily developers with an object oriented programming background.

F# Programming Wikibook is a wikibook about learning F#. It is also a product of the F# community. The target audience is people who are new to F#, with a little bit of object oriented programming background.

Learn F# through videos

F# tutorial on YouTube is a great introduction to F# using Visual Studio, showing lots of great examples over the course of 1.5 hours. The target audience is Visual Studio developers who are new to F#.

Introduction to Programming with F# is a great video series that uses Visual Studio Code as the main editor. The video series starts from nothing and ends with building a text-based RPG video game. The target audience is developers who prefer Visual Studio Code (or a lightweight IDE) and are new to F#.

What's New in Visual Studio 2017 for F# For Developers is a video course that shows some of the newer features for F# in Visual Studio 2017. The target audience is Visual Studio developers who are new to F#.

Other useful resources

The F# Snippets Website contains a massive set of code snippets showing how to do just about anything in F#, ranging from absolute beginner to highly advanced snippets.

The F# Software Foundation Slack is a great place for beginners and experts alike, is highly active, and has some of world's best F# programmers available for a chat. We highly recommend joining.

The F# Software Foundation

Although Microsoft is the primary developer of the F# language and its tools in Visual Studio, F# is also backed by an independent foundation, the F# Software Foundation (FSSF).

The mission of the F# Software Foundation is to promote, protect, and advance the F# programming language, and to support and facilitate the growth of a diverse and international community of F# programmers.

To learn more and get involved, check out fsharp.org. It's free to join, and the network of F# developers in the foundation is something you don't want to miss out on!

Get Started with F#

2/28/2019 • 2 minutes to read • Edit Online

You can get started with F# on your machine or online.

Get started on your machine

There are multiple guides on how to install and use F# for the first time on your machine. You can use the following table to help in making a decision:

OS	PREFER VISUAL STUDIO	PREFER VISUAL STUDIO CODE	PREFER COMMAND LINE
Windows	Get started with Visual Studio	Get started with Visual Studio Code and Ionide	Get started with the .NET Core CLI
macOS	Get started with VS for Mac	Get started with Visual Studio Code and Ionide	Get started with the .NET Core CLI
Linux	N/A	Get started with Visual Studio Code and Ionide	Get started with the .NET Core CLI

In general, there is no specific that is better than the rest. We recommend trying all ways to use F# on your machine to see what you like the best!

Get started online

If you'd rather not install F# and .NET on your machine, you can also get started with F# in the browser:

- Introduction to F# on Azure Notebooks is a Jupyter notebook on the free Azure Notebooks service. Just sign in with a Microsoft account and you can hack away directly in the browser!
- The Fable REPL is an interactive, in-browser REPL that uses Fable to translate F# code into JavaScript. Check out the numerous samples that range from F# basics to a fully fledged video game all executing in your browser!

Install F#

4/9/2019 • 2 minutes to read • Edit Online

You can install F# in multiple ways, depending on your environment.

Install F# with Visual Studio

If you're downloading Visual Studio for the first time, it will first install the Visual Studio installer. Install the appropriate SKU of Visual Studio from the installer. If you already have it installed, click **Modify**.

You'll next see a list of Workloads. Select **ASP.NET and web development** which will install F# support and .NET Core support for ASP.NET Core projects.

Next, click **Modify** in the lower right-hand side. This will install everything you have selected. You can then open Visual Studio 2017 with F# language support by clicking **Launch**.

Install F# with Visual Studio for Mac

F# is installed by default in Visual Studio for Mac, no matter which configuration you choose.

After the install completes, choose "Start Visual Studio". You can also launch it through Finder on macOS.

Install F# with Visual Studio Code

You must have git installed and available on your PATH to make use of project templates. You can verify that it is installed correctly by typing `git --version` at a command prompt and pressing **Enter**.

- macOS
- Linux
- Windows

Mono is used for F# Interactive support. The easiest way to install Mono on macOS is via Homebrew. Simply type the following into your terminal:

```
brew install mono
```

Also install the .NET Core SDK.

You will then need Visual Studio Code installed.

Next, click the Extensions icon and search for "Ionide":

The only plugin required for F# support in Visual Studio Code is Ionide-fsharp. However, you can also install Ionide-FAKE to get FAKE support and Ionide-Paket to get Paket support. FAKE and Paket are additional F# community tools for building projects and managing dependencies, respectively.

Get started with F# in Visual Studio

7/30/2019 • 3 minutes to read • Edit Online

F# and the Visual F# tooling are supported in the Visual Studio IDE.

To begin, ensure that you have Visual Studio installed with F#.

Creating a console application

One of the most basic projects in Visual Studio is the Console Application. Here's how to do it. Once Visual Studio is open:

1. On the **File** menu, point to **New**, and then choose **Project**.
2. In the New Project dialog, under **Templates**, you should see **Visual F#**. Choose this to show the F# templates.
3. Select either **.NET Core Console app** or **Console app**.
4. Choose the **Okay** button to create the F# project! You should now see an F# project in the Solution Explorer.

Writing your code

Let's get started by writing some code first. Make sure that the `Program.fs` file is open, and then replace its contents with the following:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

In the previous code sample, a function `square` has been defined which takes an input named `x` and multiplies it by itself. Because F# uses Type Inference, the type of `x` doesn't need to be specified. The F# compiler understands the types where multiplication is valid, and will assign a type to `x` based on how `square` is called. If you hover over `square`, you should see the following:

```
val square: x:int -> int
```

This is what is known as the function's type signature. It can be read like this: "Square is a function which takes an integer named x and produces an integer". Note that the compiler gave `square` the `int` type for now - this is because multiplication is not generic across *all* types, but rather is generic across a closed set of types. The F# compiler picked `int` at this point, but it will adjust the type signature if you call `square` with a different input type, such as a `float`.

Another function, `main`, is defined, which is decorated with the `EntryPoint` attribute to tell the F# compiler that program execution should start there. It follows the same convention as other C-style programming languages, where command-line arguments can be passed to this function, and an integer code is returned (typically `0`).

It is in this function that we call the `square` function with an argument of `12`. The F# compiler then assigns the type of `square` to be `int -> int` (that is, a function which takes an `int` and produces an `int`). The call to `printfn` is a formatted printing function which uses a format string, similar to C-style programming languages, parameters which correspond to those specified in the format string, and then prints the result and a new line.

Running your code

You can run the code and see results by pressing **Ctrl**+**F5**. This runs the program without debugging and allows you to see the results. Alternatively, you can choose the **Debug** top-level menu item in Visual Studio and choose **Start Without Debugging**.

You should now see the following printed to the console window that Visual Studio popped up:

```
12 squared is 144!
```

Congratulations! You've created your first F# project in Visual Studio, written an F# function printed the results of calling that function, and run the project to see some results.

Next steps

If you haven't already, check out the Tour of F#, which covers some of the core features of the F# language. It will give you an overview of some of the capabilities of F#, and provide ample code samples that you can copy into Visual Studio and run. There are also some great external resources you can use, showcased in the F# Guide.

See also

- Tour of F#
- F# language reference
- Type inference
- Symbol and operator reference

Get started with F# in Visual Studio for Mac

7/30/2019 • 5 minutes to read • Edit Online

F# and the Visual F# tooling are supported in the Visual Studio for Mac IDE. Ensure that you have Visual Studio for Mac installed.

Creating a console application

One of the most basic projects in Visual Studio for Mac is the Console Application. Here's how to do it. Once Visual Studio for Mac is open:

1. On the **File** menu, point to **New Solution**.
2. In the New Project dialog, there are 2 different templates for Console Application. There is one under Other -> .NET which targets the .NET Framework. The other template is under .NET Core -> App which targets .NET Core. Either template should work for the purpose of this article.
3. Under console app, change C# to F# if needed. Choose the **Next** button to move forward!
4. Give your project a name, and choose the options you want for the app. Notice, the preview pane to the side of the screen that will show the directory structure that will be created based on the options selected.
5. Click **Create**. You should now see an F# project in the Solution Explorer.

Writing your code

Let's get started by writing some code first. Make sure that the `Program.fs` file is open, and then replace its contents with the following:

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

In the previous code sample, a function `square` has been defined which takes an input named `x` and multiplies it by itself. Because F# uses Type Inference, the type of `x` doesn't need to be specified. The F# compiler understands the types where multiplication is valid, and will assign a type to `x` based on how `square` is called. If you hover over `square`, you should see the following:

```
val square: x:int -> int
```

This is what is known as the function's type signature. It can be read like this: "Square is a function which takes an integer named x and produces an integer". Note that the compiler gave `square` the `int` type for now - this is because multiplication is not generic across *all* types, but rather is generic across a closed set of types. The F# compiler picked `int` at this point, but it will adjust the type signature if you call `square` with a different input type, such as a `float`.

Another function, `main`, is defined, which is decorated with the `EntryPoint` attribute to tell the F# compiler that

program execution should start there. It follows the same convention as other C-style programming languages, where command-line arguments can be passed to this function, and an integer code is returned (typically `0`).

It is in this function that we call the `square` function with an argument of `12`. The F# compiler then assigns the type of `square` to be `int -> int` (that is, a function which takes an `int` and produces an `int`). The call to `printfn` is a formatted printing function which uses a format string, similar to C-style programming languages, parameters which correspond to those specified in the format string, and then prints the result and a new line.

Running your code

You can run the code and see results by clicking on **Run** from the top level menu and then **Start Without Debugging**. This will run the program without debugging and allows you to see the results.

You should now see the following printed to the console window that Visual Studio for Mac popped up:

```
12 squared is 144!
```

Congratulations! You've created your first F# project in Visual Studio for Mac, written an F# function printed the results of calling that function, and run the project to see some results.

Using F# Interactive

One of the best features of the Visual F# tooling in Visual Studio for Mac is the F# Interactive Window. It allows you to send code over to a process where you can call that code and see the result interactively.

To begin using it, highlight the `square` function defined in your code. Next, click on **Edit** from the top level menu. Next select **Send selection to F# Interactive**. This executes the code in the F# Interactive Window. Alternatively, you can right click on the selection and choose **Send selection to F# Interactive**. You should see the F# Interactive Window appear with the following in it:

```
>

val square : x:int -> int

>
```

This shows the same function signature for the `square` function, which you saw earlier when you hovered over the function. Because `square` is now defined in the F# Interactive process, you can call it with different values:

```
> square 12;;
val it : int = 144
>square 13;;
val it : int = 169
```

This executes the function, binds the result to a new name `it`, and displays the type and value of `it`. Note that you must terminate each line with `;;`. This is how F# Interactive knows when your function call is finished. You can also define new functions in F# Interactive:

```
> let isOdd x = x % 2 <> 0;;

val isOdd : x:int -> bool

> isOdd 12;;
val it : bool = false
```

The above defines a new function, `isOdd`, which takes an `int` and checks to see if it's odd! You can call this function to see what it returns with different inputs. You can call functions within function calls:

```
> isOdd (square 15);;
val it : bool = true
```

You can also use the pipe-forward operator to pipeline the value into the two functions:

```
> 15 |> square |> isOdd;;
val it : bool = true
```

The pipe-forward operator, and more, are covered in later tutorials.

This is only a glimpse into what you can do with F# Interactive. To learn more, check out Interactive Programming with F#.

Next steps

If you haven't already, check out the Tour of F#, which covers some of the core features of the F# language. It will give you an overview of some of the capabilities of F#, and provide ample code samples that you can copy into Visual Studio for Mac and run. There are also some great external resources you can use, showcased in the F# Guide.

See also

- Visual F#
- Tour of F#
- F# language reference
- Type inference
- Symbol and operator reference

Get Started with F# in Visual Studio Code

7/30/2019 • 8 minutes to read • Edit Online

You can write F# in Visual Studio Code with the Ionide plugin to get a great cross-platform, lightweight Integrated Development Environment (IDE) experience with IntelliSense and basic code refactorings. Visit Ionide.io to learn more about the plugin.

To begin, ensure that you have F# and the Ionide plugin correctly installed.

> **NOTE**
> Ionide will generate .NET Framework F# projects, not dotnet core, which can have cross-platform compatibility issues. If you are running on **Linux** or **OSX**, a simpler way to get started is to use the command-line tools.

Creating your first project with Ionide

To create a new F# project, open Visual Studio Code in a new folder (you can name it whatever you like).

Next, open the command palette (**View > Command Palette**) and type the following:

```
> F# new project
```

This is powered by the FORGE project.

> **NOTE**
> If you don't see template options, try refreshing templates by running the following command in the Command Palette: `>F#: Refresh Project Templates`.

Select "F#: New Project" by hitting **Enter**. This takes you to the next step, which is for selecting a project template.

Pick the `classlib` template and hit **Enter**.

Next, pick a directory to create the project in. If you leave it blank, it uses the current directory.

Finally, name your project in the final step. F# uses Pascal case for project names. This article uses `ClassLibraryDemo` as the name. Once you've entered the name you want for your project, hit **Enter**.

If you followed the previous step, you should get the Visual Studio Code Workspace on the left-hand side to appear with the following:

1. The F# project itself, underneath the `ClassLibraryDemo` folder.
2. The correct directory structure for adding packages via `Paket`.
3. A cross-platform build script with `FAKE`.
4. The `paket.exe` executable that can fetch packages and resolve dependencies for you.
5. A `.gitignore` file if you wish to add this project to Git-based source control.

Writing some code

Open the *ClassLibraryDemo* folder. You should see the following files:

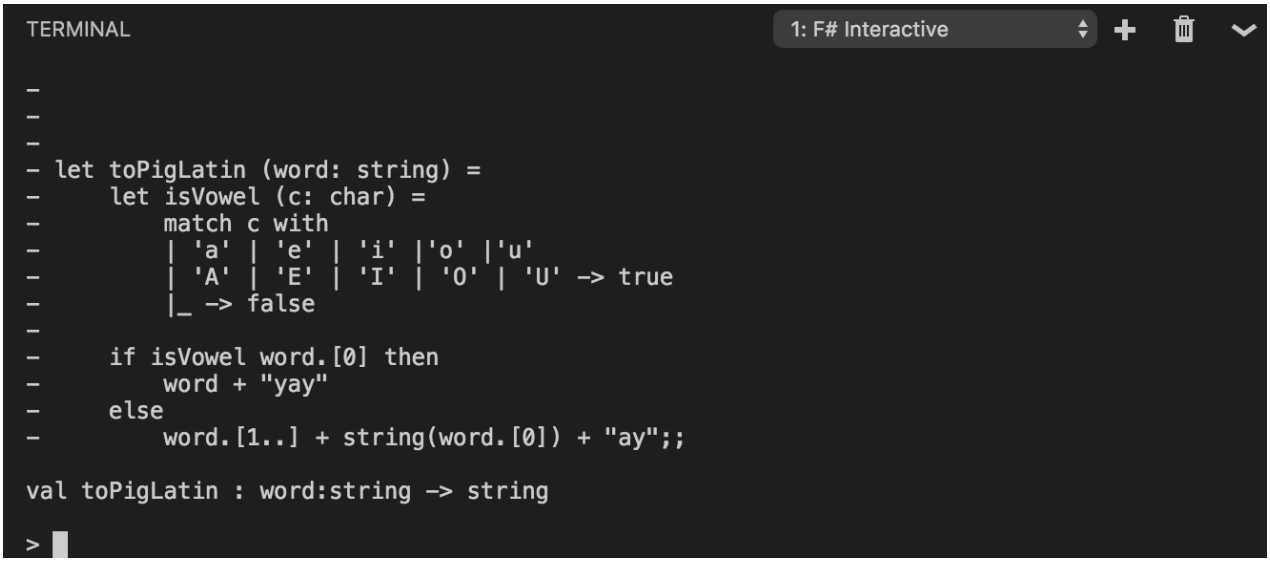
1. `ClassLibraryDemo.fs`, an F# implementation file with a class defined.
2. `ClassLibraryDemo.fsproj`, an F# project file used to build this project.
3. `Script.fsx`, an F# script file that loads the source file.
4. `paket.references`, a Paket file that specifies the project dependencies.

Open `Script.fsx`, and add the following code at the end of it:

```
let toPigLatin (word: string) =
    let isVowel (c: char) =
        match c with
        | 'a' | 'e' | 'i' |'o' |'u'
        | 'A' | 'E' | 'I' | 'O' | 'U' -> true
        |_ -> false

    if isVowel word.[0] then
        word + "yay"
    else
        word.[1..] + string(word.[0]) + "ay"
```

This function converts a word to a form of Pig Latin. The next step is to evaluate it using F# Interactive (FSI).

Highlight the entire function (it should be 11 lines long). Once it is highlighted, hold the **Alt** key and hit **Enter**. You'll notice a window pop up below, and it should show something like this:



This did three things:

1. It started the FSI process.
2. It sent the code you highlighted over the FSI process.
3. The FSI process evaluated the code you sent over.

Because what you sent over was a function, you can now call that function with FSI! In the interactive window, type the following:

```
toPigLatin "banana";;
```

You should see the following result:

```
val it : string = "ananabay"
```

Now, let's try with a vowel as the first letter. Enter the following:

```
toPigLatin "apple";;
```

You should see the following result:

```
val it : string = "appleyay"
```

The function appears to be working as expected. Congratulations, you just wrote your first F# function in Visual Studio Code and evaluated it with FSI!

> **NOTE**
> As you may have noticed, the lines in FSI are terminated with `;;`. This is because FSI allows you to enter multiple lines. The `;;` at the end lets FSI know when the code is finished.

Explaining the code

If you're not sure about what the code is actually doing, here's a step-by-step.

As you can see, `toPigLatin` is a function that takes a word as its input and converts it to a Pig-Latin representation of that word. The rules for this are as follows:

If the first character in a word starts with a vowel, add "yay" to the end of the word. If it doesn't start with a vowel, move that first character to the end of the word and add "ay" to it.

You may have noticed the following in FSI:

```
val toPigLatin : word:string -> string
```

This states that `toPigLatin` is a function that takes in a `string` as input (called `word`), and returns another `string`. This is known as the type signature of the function, a fundamental piece of F# that's key to understanding F# code. You'll also notice this if you hover over the function in Visual Studio Code.

In the body of the function, you'll notice two distinct parts:

1. An inner function, called `isVowel`, that determines if a given character (`c`) is a vowel by checking if it matches one of the provided patterns via Pattern Matching:

   ```
   let isVowel (c: char) =
       match c with
       | 'a' | 'e' | 'i' |'o' |'u'
       | 'A' | 'E' | 'I' | 'O' | 'U' -> true
       |_ -> false
   ```

2. An `if..then..else` expression that checks if the first character is a vowel, and constructs a return value out of the input characters based on if the first character was a vowel or not:

   ```
   if isVowel word.[0] then
       word + "yay"
   else
       word.[1..] + string(word.[0]) + "ay"
   ```

The flow of `toPigLatin` is thus:

Check if the first character of the input word is a vowel. If it is, attach "yay" to the end of the word. Otherwise, move that first character to the end of the word and add "ay" to it.

There's one final thing to notice about this: there's no explicit instruction to return from the function, unlike many other languages out there. This is because F# is Expression-based, and the last expression in the body of a function is the return value. Because `if..then..else` is itself an expression, the body of the `then` block or the body of the `else` block will be returned depending on the input value.

Moving your script code into the implementation file

The previous sections in this article demonstrated a common first step in writing F# code: writing an initial function and executing it interactively with FSI. This is known as REPL-driven development, where REPL stands for "Read-Evaluate-Print Loop". It's a great way to experiment with functionality until you have something working.

The next step in REPL-driven development is to move working code into an F# implementation file. It can then be compiled by the F# compiler into an assembly that can be executed.

To begin, open `ClassLibraryDemo.fs`. You'll notice that some code is already in there. Go ahead and delete the class definition, but make sure to leave the `namespace` declaration at the top.

Next, create a new `module` called `PigLatin` and copy the `toPigLatin` function into it as such:

```
namespace ClassLibraryDemo

module PigLatin =
    let toPigLatin (word: string) =
        let isVowel (c: char) =
            match c with
            | 'a' | 'e' | 'i' |'o' |'u'
            | 'A' | 'E' | 'I' | 'O' | 'U' -> true
            |_ -> false

        if isVowel word.[0] then
            word + "yay"
        else
            word.[1..] + string(word.[0]) + "ay"
```

Next, open the `Script.fsx` file again, and delete the entire `toPigLatin` function, but make sure to keep the following two lines in the file:

```
#load "ClassLibraryDemo.fs"
open ClassLibraryDemo
```

Select both lines of text and press Alt+Enter to execute these lines in FSI. These will load the contents of the Pig Latin library into the FSI process and `open` the `ClassLibraryDemo` namespace so that you have access to the functionality.

Next, in the FSI window, call the function with the `PigLatin` module that you defined earlier:

```
> PigLatin.toPigLatin "banana";;
val it : string = "ananabay"
> PigLatin.toPigLatin "apple";;
val it : string = "appleyay"
```

Success! You get the same results as before, but now loaded from an F# implementation file. The major difference here is that F# source files are compiled into assemblies that can be executed anywhere, not just in FSI.

Summary

In this article, you've learned:

1. How to set up Visual Studio Code with Ionide.
2. How to create your first F# project with Ionide.
3. How to use F# Scripting to write your first F# function in Ionide and then execute it in FSI.
4. How to migrate your script code to F# source and then call that code from FSI.

You're now equipped to write much more F# code using Visual Studio Code and Ionide.

Troubleshooting

Here are a few ways you can troubleshoot certain problems that you might run into:

1. To get the code editing features of Ionide, your F# files need to be saved to disk and inside of a folder that is open in the Visual Studio Code workspace.
2. If you've made changes to your system or installed Ionide prerequisites with Visual Studio Code open, restart Visual Studio Code.
3. Check that you can use the F# compiler and F# interactive from the command line without a fully-qualified path. You can do so by typing `fsc` in a command line for the F# compiler, and `fsi` or `fsharpi` for the Visual F# tools on Windows and Mono on Mac/Linux, respectively.
4. If you have invalid characters in your project directories, Ionide might not work. Rename your project directories if this is the case.
5. If none of the Ionide commands are working, check your Visual Studio Code keybindings to see if you're overriding them by accident.
6. If Ionide is broken on your machine and none of the above has fixed your problem, try removing the `ionide-fsharp` directory on your machine and reinstall the plugin suite.

Ionide is an open source project built and maintained by members of the F# community. Please report issues and feel free to contribute at the Ionide-VSCode: FSharp GitHub repository.

If you have an issue to report, please follow the instructions for getting logs to use when reporting an issue.

You can also ask for further help from the Ionide developers and F# community in the Ionide Gitter channel.

Next steps

To learn more about F# and the features of the language, check out Tour of F#.

Get started with F# with the .NET Core CLI

1/9/2019 • 2 minutes to read • Edit Online

This article covers how you can get started with F# on any operating system (Windows, macOS, or Linux) with the .NET Core CLI. It goes through building a multi-project solution with a class library that is called by a console application.

Prerequisites

To begin, you must install the latest .NET Core SDK.

This article assumes that you know how to use a command line and have a preferred text editor. If you don't already use it, Visual Studio Code is a great option as a text editor for F#.

Build a simple multi-project solution

Open a command prompt/terminal and use the dotnet new command to create new solution file called `FSNetCore`:

```
dotnet new sln -o FSNetCore
```

The following directory structure is produced after running the previous command:

```
FSNetCore
    ├── FSNetCore.sln
```

Write a class library

Change directories to *FSNetCore*.

Use the `dotnet new` command, create a class library project in the **src** folder named Library.

```
dotnet new classlib -lang F# -o src/Library
```

The following directory structure is produced after running the previous command:

```
└── FSNetCore
    ├── FSNetCore.sln
    └── src
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Replace the contents of `Library.fs` with the following code:

```
module Library

open Newtonsoft.Json

let getJsonNetJson value =
    sprintf "I used to be %s but now I'm %s thanks to JSON.NET!" value (JsonConvert.SerializeObject(value))
```

Add the Newtonsoft.Json NuGet package to the Library project.

```
dotnet add src/Library/Library.fsproj package Newtonsoft.Json
```

Add the `Library` project to the `FSNetCore` solution using the dotnet sln add command:

```
dotnet sln add src/Library/Library.fsproj
```

Run `dotnet build` to build the project. Unresolved dependencies will be restored when building.

Write a console application that consumes the class library

Use the `dotnet new` command, create a console application in the **src** folder named App.

```
dotnet new console -lang F# -o src/App
```

The following directory structure is produced after running the previous command:

```
└── FSNetCore
    ├── FSNetCore.sln
    └── src
        ├── App
        │   ├── App.fsproj
        │   ├── Program.fs
        └── Library
            ├── Library.fs
            └── Library.fsproj
```

Replace the contents of the `Program.fs` file with the following code:

```
open System
open Library

[<EntryPoint>]
let main argv =
    printfn "Nice command-line arguments! Here's what JSON.NET has to say about them:"

    argv
    |> Array.map getJsonNetJson
    |> Array.iter (printfn "%s")

    0 // return an integer exit code
```

Add a reference to the `Library` project using dotnet add reference.

```
dotnet add src/App/App.fsproj reference src/Library/Library.fsproj
```

Add the `App` project to the `FSNetCore` solution using the `dotnet sln add` command:

```
dotnet sln add src/App/App.fsproj
```

Restore the NuGet dependencies, `dotnet restore` and run `dotnet build` to build the project.

Change directory to the `src/App` console project and run the project passing `Hello World` as arguments:

```
cd src/App
dotnet run Hello World
```

You should see the following results:

```
Nice command-line arguments! Here's what JSON.NET has to say about them:

I used to be Hello but now I'm ""Hello"" thanks to JSON.NET!
I used to be World but now I'm ""World"" thanks to JSON.NET!
```

Next steps

Next, check out the Tour of F# to learn more about different F# features.

What is F#

7/30/2019 • 2 minutes to read • Edit Online

F# is a functional programming language that makes it easy to write correct and maintainable code.

F# programming primarily involves defining types and functions that are type-inferred and generalized automatically. This allows your focus to remain on the problem domain and manipulating its data, rather than the details of programming.

```
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name =
    sprintf "Hello, %s! Isn't F# great?" name

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]

    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn "%s" greeting)

    0
```

F# has numerous features, including:

- Lightweight syntax
- Immutable by default
- Type inference and automatic generalization
- First-class functions
- Powerful data types
- Pattern matching
- Async programming

A full set of features are documented in the F# language reference.

Rich data types

Data types such as Records and Discriminated Unions let you represent complex data and domains.

```
// Group data with Records
type SuccessfulWithdrawal = {
    Amount: decimal
    Balance: decimal
}

type FailedWithdrawal = {
    Amount: decimal
    Balance: decimal
    IsOverdraft: bool
}

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure
```

F# records and discriminated unions are non-null, immutable, and comparable by default, making them very easy to use.

Enforced correctness with functions and pattern matching

F# functions are easy to declare and powerful in practice. When combined with pattern matching, they allow you to define behavior whose correctness is enforced by the compiler.

```
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn "Successfully withdrew %f" s.Amount
    | InsufficientFunds f -> printfn "Failed: balance is %f" f.Balance
    | CardExpired d -> printfn "Failed: card expired on %O" d
    | UndisclosedFailure -> printfn "Failed: unknown :("
```

F# functions are also first-class, meaning they can be passed as parameters and returned from other functions.

Functions to define operations on objects

F# has full support for objects, which are useful data types when you need to blend data and functionality. F# functions are used to manipulate objects.

```
type Set<[<EqualityConditionOn>] 'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) =// Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<‘T>
    interface IReadOnlyCollection<‘T>

[<RequireQualifiedAccess>]
module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)
```

Rather than writing code that is object-oriented, in F#, you will often write code that treats objects as another data type for functions to manipulate. Features such as generic interfaces, object expressions, and judicious use of members are common in larger F# programs.

Next steps

To learn more about a larger set of F# features, check out the F# Tour.

Tour of F#

7/30/2019 • 31 minutes to read • Edit Online

The best way to learn about F# is to read and write F# code. This article will act as a tour through some of the key features of the F# language and give you some code snippets that you can execute on your machine. To learn about setting up a development environment, check out Getting Started.

There are two primary concepts in F#: functions and types. This tour will emphasize features of the language which fall into these two concepts.

Executing the code online

If you don't have F# installed on your machine, you can execute all of the samples in your browser with Try F# on WebAssembly. Fable is a dialect of F# that executes directly in your browser. To view the samples that follow in the REPL, check out **Samples > Learn > Tour of F#** in the left-hand menu bar of the Fable REPL.

Functions and Modules

The most fundamental pieces of any F# program are ***functions*** organized into ***modules***. Functions perform work on inputs to produce outputs, and they are organized under Modules, which are the primary way you group things in F#. They are defined using the `let` binding, which give the function a name and define its arguments.

```
module BasicFunctions =

    /// You use 'let' to define a function. This one accepts an integer argument and returns an integer.
    /// Parentheses are optional for function arguments, except for when you use an explicit type
annotation.
    let sampleFunction1 x = x*x + 3

    /// Apply the function, naming the function return result using 'let'.
    /// The variable type is inferred from the function return type.
    let result1 = sampleFunction1 4573

    // This line uses '%d' to print the result as an integer. This is type-safe.
    // If 'result1' were not of type 'int', then the line would fail to compile.
    printfn "The result of squaring the integer 4573 and adding 3 is %d" result1

    /// When needed, annotate the type of a parameter name using '(argument:type)'.  Parentheses are
required.
    let sampleFunction2 (x:int) = 2*x*x - x/5 + 3

    let result2 = sampleFunction2 (7 + 4)
    printfn "The result of applying the 2nd sample function to (7 + 4) is %d" result2

    /// Conditionals use if/then/elif/else.
    ///
    /// Note that F# uses white space indentation-aware syntax, similar to languages like Python.
    let sampleFunction3 x =
        if x < 100.0 then
            2.0*x*x - x/5.0 + 3.0
        else
            2.0*x*x + x/5.0 - 37.0

    let result3 = sampleFunction3 (6.5 + 4.5)

    // This line uses '%f' to print the result as a float.  As with '%d' above, this is type-safe.
    printfn "The result of applying the 3rd sample function to (6.5 + 4.5) is %f" result3
```

`let` bindings are also how you bind a value to a name, similar to a variable in other languages. `let` bindings are ***immutable*** by default, which means that once a value or function is bound to a name, it cannot be changed in-place. This is in contrast to variables in other languages, which are ***mutable***, meaning their values can be changed at any point in time. If you require a mutable binding, you can use `let mutable ...` syntax.

```
module Immutability =

    /// Binding a value to a name via 'let' makes it immutable.
    ///
    /// The second line of code fails to compile because 'number' is immutable and bound.
    /// Re-defining 'number' to be a different value is not allowed in F#.
    let number = 2
    // let number = 3

    /// A mutable binding.  This is required to be able to mutate the value of 'otherNumber'.
    let mutable otherNumber = 2

    printfn "'otherNumber' is %d" otherNumber

    // When mutating a value, use '<-' to assign a new value.
    //
    // Note that '=' is not the same as this.  '=' is used to test equality.
    otherNumber <- otherNumber + 1

    printfn "'otherNumber' changed to be %d" otherNumber
```

Numbers, Booleans, and Strings

As a .NET language, F# supports the same underlying primitive types that exist in .NET.

Here is how various numeric types are represented in F#:

```
module IntegersAndNumbers =

    /// This is a sample integer.
    let sampleInteger = 176

    /// This is a sample floating point number.
    let sampleDouble = 4.1

    /// This computed a new number by some arithmetic.  Numeric types are converted using
    /// functions 'int', 'double' and so on.
    let sampleInteger2 = (sampleInteger/4 + 5 - 7) * 4 + int sampleDouble

    /// This is a list of the numbers from 0 to 99.
    let sampleNumbers = [ 0 .. 99 ]

    /// This is a list of all tuples containing all the numbers from 0 to 99 and their squares.
    let sampleTableOfSquares = [ for i in 0 .. 99 -> (i, i*i) ]

    // The next line prints a list that includes tuples, using '%A' for generic printing.
    printfn "The table of squares from 0 to 99 is:\n%A" sampleTableOfSquares
```

Here's what Boolean values and performing basic conditional logic looks like:

```
module Booleans =

    /// Booleans values are 'true' and 'false'.
    let boolean1 = true
    let boolean2 = false

    /// Operators on booleans are 'not', '&&' and '||'.
    let boolean3 = not boolean1 && (boolean2 || false)

    // This line uses '%b'to print a boolean value.  This is type-safe.
    printfn "The expression 'not boolean1 && (boolean2 || false)' is %b" boolean3
```

And here's what basic string manipulation looks like:

```
module StringManipulation =

    /// Strings use double quotes.
    let string1 = "Hello"
    let string2  = "world"

    /// Strings can also use @ to create a verbatim string literal.
    /// This will ignore escape characters such as '\', '\n', '\t', etc.
    let string3 = @"C:\Program Files\"

    /// String literals can also use triple-quotes.
    let string4 = """The computer said "hello world" when I told it to!"""

    /// String concatenation is normally done with the '+' operator.
    let helloWorld = string1 + " " + string2

    // This line uses '%s' to print a string value.  This is type-safe.
    printfn "%s" helloWorld

    /// Substrings use the indexer notation.  This line extracts the first 7 characters as a substring.
    /// Note that like many languages, Strings are zero-indexed in F#.
    let substring = helloWorld.[0..6]
    printfn "%s" substring
```

Tuples

Tuples are a big deal in F#. They are a grouping of unnamed, but ordered values, that can be treated as values themselves. Think of them as values which are aggregated from other values. They have many uses, such as conveniently returning multiple values from a function, or grouping values for some ad-hoc convenience.

```
module Tuples =

    /// A simple tuple of integers.
    let tuple1 = (1, 2, 3)

    /// A function that swaps the order of two values in a tuple.
    ///
    /// F# Type Inference will automatically generalize the function to have a generic type,
    /// meaning that it will work with any type.
    let swapElems (a, b) = (b, a)

    printfn "The result of swapping (1, 2) is %A" (swapElems (1,2))

    /// A tuple consisting of an integer, a string,
    /// and a double-precision floating point number.
    let tuple2 = (1, "fred", 3.1415)

    printfn "tuple1: %A\ttuple2: %A" tuple1 tuple2
```

As of F# 4.1, you can also create `struct` tuples. These also interoperate fully with C#7/Visual Basic 15 tuples, which are also `struct` tuples:

```
/// Tuples are normally objects, but they can also be represented as structs.
///
/// These interoperate completely with structs in C# and Visual Basic.NET; however,
/// struct tuples are not implicitly convertible with object tuples (often called reference tuples).
///
/// The second line below will fail to compile because of this.  Uncomment it to see what happens.
let sampleStructTuple = struct (1, 2)
//let thisWillNotCompile: (int*int) = struct (1, 2)

// Although you can
let convertFromStructTuple (struct(a, b)) = (a, b)
let convertToStructTuple (a, b) = struct(a, b)

printfn "Struct Tuple: %A\nReference tuple made from the Struct Tuple: %A" sampleStructTuple
(sampleStructTuple |> convertFromStructTuple)
```

It's important to note that because `struct` tuples are value types, they cannot be implicitly converted to reference tuples, or vice versa. You must explicitly convert between a reference and struct tuple.

Pipelines and Composition

Pipe operators such as `|>` are used extensively when processing data in F#. These operators are functions that allow you to establish "pipelines" of functions in a flexible manner. The following example walks through how you can take advantage of these operators to build a simple functional pipeline:

```
module PipelinesAndComposition =

    /// Squares a value.
    let square x = x * x

    /// Adds 1 to a value.
    let addOne x = x + 1

    /// Tests if an integer value is odd via modulo.
    let isOdd x = x % 2 <> 0

    /// A list of 5 numbers.  More on lists later.
    let numbers = [ 1; 2; 3; 4; 5 ]

    /// Given a list of integers, it filters out the even numbers,
    /// squares the resulting odds, and adds 1 to the squared odds.
    let squareOddValuesAndAddOne values =
        let odds = List.filter isOdd values
        let squares = List.map square odds
        let result = List.map addOne squares
        result

    printfn "processing %A through 'squareOddValuesAndAddOne' produces: %A" numbers
(squareOddValuesAndAddOne numbers)

    /// A shorter way to write 'squareOddValuesAndAddOne' is to nest each
    /// sub-result into the function calls themselves.
    ///
    /// This makes the function much shorter, but it's difficult to see the
    /// order in which the data is processed.
    let squareOddValuesAndAddOneNested values =
        List.map addOne (List.map square (List.filter isOdd values))

    printfn "processing %A through 'squareOddValuesAndAddOneNested' produces: %A" numbers
(squareOddValuesAndAddOneNested numbers)

    /// A preferred way to write 'squareOddValuesAndAddOne' is to use F# pipe operators.
    /// This allows you to avoid creating intermediate results, but is much more readable
    /// than nesting function calls like 'squareOddValuesAndAddOneNested'
    let squareOddValuesAndAddOnePipeline values =
        values
        |> List.filter isOdd
        |> List.map square
        |> List.map addOne

    printfn "processing %A through 'squareOddValuesAndAddOnePipeline' produces: %A" numbers
(squareOddValuesAndAddOnePipeline numbers)

    /// You can shorten 'squareOddValuesAndAddOnePipeline' by moving the second `List.map` call
    /// into the first, using a Lambda Function.
    ///
    /// Note that pipelines are also being used inside the lambda function.  F# pipe operators
    /// can be used for single values as well.  This makes them very powerful for processing data.
    let squareOddValuesAndAddOneShorterPipeline values =
        values
        |> List.filter isOdd
        |> List.map(fun x -> x |> square |> addOne)

    printfn "processing %A through 'squareOddValuesAndAddOneShorterPipeline' produces: %A" numbers
(squareOddValuesAndAddOneShorterPipeline numbers)
```

The previous sample made use of many features of F#, including list processing functions, first-class functions, and partial application. Although a deep understanding of each of those concepts can become somewhat advanced, it should be clear how easily functions can be used to process data when building pipelines.

Lists, Arrays, and Sequences

Lists, Arrays, and Sequences are three primary collection types in the F# core library.

Lists are ordered, immutable collections of elements of the same type. They are singly-linked lists, which means they are meant for enumeration, but a poor choice for random access and concatenation if they're large. This in contrast to Lists in other popular languages, which typically do not use a singly-linked list to represent Lists.

```
module Lists =

    /// Lists are defined using [ ... ].  This is an empty list.
    let list1 = [ ]

    /// This is a list with 3 elements.  ';' is used to separate elements on the same line.
    let list2 = [ 1; 2; 3 ]

    /// You can also separate elements by placing them on their own lines.
    let list3 = [
        1
        2
        3
    ]

    /// This is a list of integers from 1 to 1000
    let numberList = [ 1 .. 1000 ]

    /// Lists can also be generated by computations. This is a list containing
    /// all the days of the year.
    let daysList =
        [ for month in 1 .. 12 do
              for day in 1 .. System.DateTime.DaysInMonth(2017, month) do
                  yield System.DateTime(2017, month, day) ]

    // Print the first 5 elements of 'daysList' using 'List.take'.
    printfn "The first 5 days of 2017 are: %A" (daysList |> List.take 5)

    /// Computations can include conditionals.  This is a list containing the tuples
    /// which are the coordinates of the black squares on a chess board.
    let blackSquares =
        [ for i in 0 .. 7 do
              for j in 0 .. 7 do
                  if (i+j) % 2 = 1 then
                      yield (i, j) ]

    /// Lists can be transformed using 'List.map' and other functional programming combinators.
    /// This definition produces a new list by squaring the numbers in numberList, using the pipeline
    /// operator to pass an argument to List.map.
    let squares =
        numberList
        |> List.map (fun x -> x*x)

    /// There are many other list combinations. The following computes the sum of the squares of the
    /// numbers divisible by 3.
    let sumOfSquares =
        numberList
        |> List.filter (fun x -> x % 3 = 0)
        |> List.sumBy (fun x -> x * x)

    printfn "The sum of the squares of numbers up to 1000 that are divisible by 3 is: %d" sumOfSquares
```

Arrays are fixed-size, *mutable* collections of elements of the same type. They support fast random access of elements, and are faster than F# lists because they are just contiguous blocks of memory.

```
module Arrays =

    /// This is The empty array.  Note that the syntax is similar to that of Lists, but uses `[| ... |]`
instead.
    let array1 = [| |]

    /// Arrays are specified using the same range of constructs as lists.
    let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

    /// This is an array of numbers from 1 to 1000.
    let array3 = [| 1 .. 1000 |]

    /// This is an array containing only the words "hello" and "world".
    let array4 =
        [| for word in array2 do
               if word.Contains("l") then
                   yield word |]

    /// This is an array initialized by index and containing the even numbers from 0 to 2000.
    let evenNumbers = Array.init 1001 (fun n -> n * 2)

    /// Sub-arrays are extracted using slicing notation.
    let evenNumbersSlice = evenNumbers.[0..500]

    /// You can loop over arrays and lists using 'for' loops.
    for word in array4 do
        printfn "word: %s" word

    // You can modify the contents of an array element by using the left arrow assignment operator.
    //
    // To learn more about this operator, see: https://docs.microsoft.com/dotnet/fsharp/language-
reference/values/index#mutable-variables
    array2.[1] <- "WORLD!"

    /// You can transform arrays using 'Array.map' and other functional programming operations.
    /// The following calculates the sum of the lengths of the words that start with 'h'.
    let sumOfLengthsOfWords =
        array2
        |> Array.filter (fun x -> x.StartsWith "h")
        |> Array.sumBy (fun x -> x.Length)

    printfn "The sum of the lengths of the words in Array 2 is: %d" sumOfLengthsOfWords
```

Sequences are a logical series of elements, all of the same type. These are a more general type than Lists and Arrays, capable of being your "view" into any logical series of elements. They also stand out because they can be ***lazy***, which means that elements can be computed only when they are needed.

```
module Sequences =

    /// This is the empty sequence.
    let seq1 = Seq.empty

    /// This a sequence of values.
    let seq2 = seq { yield "hello"; yield "world"; yield "and"; yield "hello"; yield "world"; yield "again"
}

    /// This is an on-demand sequence from 1 to 1000.
    let numbersSeq = seq { 1 .. 1000 }

    /// This is a sequence producing the words "hello" and "world"
    let seq3 =
        seq { for word in seq2 do
                  if word.Contains("l") then
                      yield word }

    /// This sequence producing the even numbers up to 2000.
    let evenNumbers = Seq.init 1001 (fun n -> n * 2)

    let rnd = System.Random()

    /// This is an infinite sequence which is a random walk.
    /// This example uses yield! to return each element of a subsequence.
    let rec randomWalk x =
        seq { yield x
              yield! randomWalk (x + rnd.NextDouble() - 0.5) }

    /// This example shows the first 100 elements of the random walk.
    let first100ValuesOfRandomWalk =
        randomWalk 5.0
        |> Seq.truncate 100
        |> Seq.toList

    printfn "First 100 elements of a random walk: %A" first100ValuesOfRandomWalk
```

Recursive Functions

Processing collections or sequences of elements is typically done with recursion in F#. Although F# has support for loops and imperative programming, recursion is preferred because it is easier to guarantee correctness.

> **NOTE**
>
> The following example makes use of the pattern matching via the `match` expression. This fundamental construct is covered later in this article.

```
module RecursiveFunctions =

    /// This example shows a recursive function that computes the factorial of an
    /// integer. It uses 'let rec' to define a recursive function.
    let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

    printfn "Factorial of 6 is: %d" (factorial 6)

    /// Computes the greatest common factor of two integers.
    ///
    /// Since all of the recursive calls are tail calls,
    /// the compiler will turn the function into a loop,
    /// which improves performance and reduces memory consumption.
    let rec greatestCommonFactor a b =
        if a = 0 then b
        elif a < b then greatestCommonFactor a (b - a)
        else greatestCommonFactor (a - b) b

    printfn "The Greatest Common Factor of 300 and 620 is %d" (greatestCommonFactor 300 620)

    /// This example computes the sum of a list of integers using recursion.
    let rec sumList xs =
        match xs with
        | []    -> 0
        | y::ys -> y + sumList ys

    /// This makes 'sumList' tail recursive, using a helper function with a result accumulator.
    let rec private sumListTailRecHelper accumulator xs =
        match xs with
        | []    -> accumulator
        | y::ys -> sumListTailRecHelper (accumulator+y) ys

    /// This invokes the tail recursive helper function, providing '0' as a seed accumulator.
    /// An approach like this is common in F#.
    let sumListTailRecursive xs = sumListTailRecHelper 0 xs

    let oneThroughTen = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

    printfn "The sum 1-10 is %d" (sumListTailRecursive oneThroughTen)
```

F# also has full support for Tail Call Optimization, which is a way to optimize recursive calls so that they are just as fast as a loop construct.

Record and Discriminated Union Types

Record and Union types are two fundamental data types used in F# code, and are generally the best way to represent data in an F# program. Although this makes them similar to classes in other languages, one of their primary differences is that they have structural equality semantics. This means that they are "natively" comparable and equality is straightforward - just check if one is equal to the other.

Records are an aggregate of named values, with optional members (such as methods). If you're familiar with C# or Java, then these should feel similar to POCOs or POJOs - just with structural equality and less ceremony.

```
module RecordTypes =

    /// This example shows how to define a new record type.
    type ContactCard =
        { Name     : string
          Phone    : string
          Verified : bool }

    /// This example shows how to instantiate a record type.
    let contact1 =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false }

    /// You can also do this on the same line with ';' separators.
    let contactOnSameLine = { Name = "Alf"; Phone = "(206) 555-0157"; Verified = false }

    /// This example shows how to use "copy-and-update" on record values. It creates
    /// a new record value that is a copy of contact1, but has different values for
    /// the 'Phone' and 'Verified' fields.
    ///
    /// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/copy-and-update-
record-expressions
    let contact2 =
        { contact1 with
            Phone = "(206) 555-0112"
            Verified = true }

    /// This example shows how to write a function that processes a record value.
    /// It converts a 'ContactCard' object to a string.
    let showContactCard (c: ContactCard) =
        c.Name + " Phone: " + c.Phone + (if not c.Verified then " (unverified)" else "")

    printfn "Alf's Contact Card: %s" (showContactCard contact1)

    /// This is an example of a Record with a member.
    type ContactCardAlternate =
        { Name     : string
          Phone    : string
          Address  : string
          Verified : bool }

        /// Members can implement object-oriented members.
        member this.PrintedContactCard =
            this.Name + " Phone: " + this.Phone + (if not this.Verified then " (unverified)" else "") +
this.Address

    let contactAlternate =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false
          Address = "111 Alf Street" }

    // Members are accessed via the '.' operator on an instantiated type.
    printfn "Alf's alternate contact card is %s" contactAlternate.PrintedContactCard
```

As of F# 4.1, you can also represent Records as `struct`s. This is done with the `[<Struct>]` attribute:

```
/// Records can also be represented as structs via the 'Struct' attribute.
/// This is helpful in situations where the performance of structs outweighs
/// the flexibility of reference types.
[<Struct>]
type ContactCardStruct =
    { Name     : string
      Phone    : string
      Verified : bool }
```

Discriminated Unions (DUs) are values which could be a number of named forms or cases. Data stored in the type can be one of several distinct values.

```
module DiscriminatedUnions =

    /// The following represents the suit of a playing card.
    type Suit =
        | Hearts
        | Clubs
        | Diamonds
        | Spades

    /// A Discriminated Union can also be used to represent the rank of a playing card.
    type Rank =
        /// Represents the rank of cards 2 .. 10
        | Value of int
        | Ace
        | King
        | Queen
        | Jack

        /// Discriminated Unions can also implement object-oriented members.
        static member GetAllRanks() =
            [ yield Ace
              for i in 2 .. 10 do yield Value i
              yield Jack
              yield Queen
              yield King ]

    /// This is a record type that combines a Suit and a Rank.
    /// It's common to use both Records and Discriminated Unions when representing data.
    type Card = { Suit: Suit; Rank: Rank }

    /// This computes a list representing all the cards in the deck.
    let fullDeck =
        [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
              for rank in Rank.GetAllRanks() do
                  yield { Suit=suit; Rank=rank } ]

    /// This example converts a 'Card' object to a string.
    let showPlayingCard (c: Card) =
        let rankString =
            match c.Rank with
            | Ace -> "Ace"
            | King -> "King"
            | Queen -> "Queen"
            | Jack -> "Jack"
            | Value n -> string n
        let suitString =
            match c.Suit with
            | Clubs -> "clubs"
            | Diamonds -> "diamonds"
            | Spades -> "spades"
            | Hearts -> "hearts"
        rankString  + " of " + suitString

    /// This example prints all the cards in a playing deck.
    let printAllCards() =
        for card in fullDeck do
            printfn "%s" (showPlayingCard card)
```

You can also use DUs as *Single-Case Discriminated Unions*, to help with domain modeling over primitive types. Often times, strings and other primitive types are used to represent something, and are thus given a particular meaning. However, using only the primitive representation of the data can result in mistakenly assigning an incorrect value! Representing each type of information as a distinct single-case union can enforce correctness in this scenario.

```
// Single-case DUs are often used for domain modeling.  This can buy you extra type safety
// over primitive types such as strings and ints.
//
// Single-case DUs cannot be implicitly converted to or from the type they wrap.
// For example, a function which takes in an Address cannot accept a string as that input,
// or vice versa.
type Address = Address of string
type Name = Name of string
type SSN = SSN of int

// You can easily instantiate a single-case DU as follows.
let address = Address "111 Alf Way"
let name = Name "Alf"
let ssn = SSN 1234567890

/// When you need the value, you can unwrap the underlying value with a simple function.
let unwrapAddress (Address a) = a
let unwrapName (Name n) = n
let unwrapSSN (SSN s) = s

// Printing single-case DUs is simple with unwrapping functions.
printfn "Address: %s, Name: %s, and SSN: %d" (address |> unwrapAddress) (name |> unwrapName) (ssn |>
unwrapSSN)
```

As the above sample demonstrates, to get the underlying value in a single-case Discriminated Union, you must explicitly unwrap it.

Additionally, DUs also support recursive definitions, allowing you to easily represent trees and inherently recursive data. For example, here's how you can represent a Binary Search Tree with `exists` and `insert` functions.

```
/// Discriminated Unions also support recursive definitions.
///
/// This represents a Binary Search Tree, with one case being the Empty tree,
/// and the other being a Node with a value and two subtrees.
type BST<'T> =
    | Empty
    | Node of value:'T * left: BST<'T> * right: BST<'T>

/// Check if an item exists in the binary search tree.
/// Searches recursively using Pattern Matching.  Returns true if it exists; otherwise, false.
let rec exists item bst =
    match bst with
    | Empty -> false
    | Node (x, left, right) ->
        if item = x then true
        elif item < x then (exists item left) // Check the left subtree.
        else (exists item right) // Check the right subtree.

/// Inserts an item in the Binary Search Tree.
/// Finds the place to insert recursively using Pattern Matching, then inserts a new node.
/// If the item is already present, it does not insert anything.
let rec insert item bst =
    match bst with
    | Empty -> Node(item, Empty, Empty)
    | Node(x, left, right) as node ->
        if item = x then node // No need to insert, it already exists; return the node.
        elif item < x then Node(x, insert item left, right) // Call into left subtree.
        else Node(x, left, insert item right) // Call into right subtree.
```

Because DUs allow you to represent the recursive structure of the tree in the data type, operating on this recursive structure is straightforward and guarantees correctness. It is also supported in pattern matching, as shown below.

Additionally, you can represent DUs as `struct`s with the `[<Struct>]` attribute:

```
/// Discriminated Unions can also be represented as structs via the 'Struct' attribute.
/// This is helpful in situations where the performance of structs outweighs
/// the flexibility of reference types.
///
/// However, there are two important things to know when doing this:
///     1. A struct DU cannot be recursively-defined.
///     2. A struct DU must have unique names for each of its cases.
[<Struct>]
type Shape =
    | Circle of radius: float
    | Square of side: float
    | Triangle of height: float * width: float
```

However, there are two key things to keep in mind when doing so:

1. A struct DU cannot be recursively-defined.
2. A struct DU must have unique names for each of its cases.

Failure to follow the above will result in a compilation error.

Pattern Matching

Pattern Matching is the F# language feature which enables correctness for operating on F# types. In the above samples, you probably noticed quite a bit of `match x with ...` syntax. This construct allows the compiler, which can understand the "shape" of data types, to force you to account for all possible cases when using a data type through what is known as Exhaustive Pattern Matching. This is incredibly powerful for correctness, and can be cleverly used to "lift" what would normally be a runtime concern into compile-time.

```
module PatternMatching =

    /// A record for a person's first and last name
    type Person = {
        First : string
        Last  : string
    }

    /// A Discriminated Union of 3 different kinds of employees
    type Employee =
        | Engineer of engineer: Person
        | Manager of manager: Person * reports: List<Employee>
        | Executive of executive: Person * reports: List<Employee> * assistant: Employee

    /// Count everyone underneath the employee in the management hierarchy,
    /// including the employee. The matches bind names to the properties
    /// of the cases so that those names can be used inside the match branches.
    /// Note that the names used for binding do not need to be the same as the
    /// names given in the DU definition above.
    let rec countReports(emp : Employee) =
        1 + match emp with
            | Engineer(person) ->
                0
            | Manager(person, reports) ->
                reports |> List.sumBy countReports
            | Executive(person, reports, assistant) ->
                (reports |> List.sumBy countReports) + countReports assistant


    /// Find all managers/executives named "Dave" who do not have any reports.
    /// This uses the 'function' shorthand to as a lambda expression.
    let rec findDaveWithOpenPosition(emps : List<Employee>) =
        emps
        |> List.filter(function
                       | Manager({First = "Dave"}, []) -> true // [] matches an empty list.
                       | Executive({First = "Dave"}, [], _) -> true
                       | _ -> false) // '_' is a wildcard pattern that matches anything.
                                     // This handles the "or else" case.
```

Something you may have noticed is the use of the `_` pattern. This is known as the Wildcard Pattern, which is a way of saying "I don't care what something is". Although convenient, you can accidentally bypass Exhaustive Pattern Matching and no longer benefit from compile-time enforcements if you aren't careful in using `_`. It is best used when you don't care about certain pieces of a decomposed type when pattern matching, or the final clause when you have enumerated all meaningful cases in a pattern matching expression.

Active Patterns are another powerful construct to use with pattern matching. They allow you to partition input data into custom forms, decomposing them at the pattern match call site. They can also be parameterized, thus allowing to define the partition as a function. Expanding the previous example to support Active Patterns looks something like this:

```
// Active Patterns are another powerful construct to use with pattern matching.
// They allow you to partition input data into custom forms, decomposing them at the pattern match call
site.
//
// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/active-patterns
let (|Int|_|) = parseInt
let (|Double|_|) = parseDouble
let (|Date|_|) = parseDateTimeOffset
let (|TimeSpan|_|) = parseTimeSpan

/// Pattern Matching via 'function' keyword and Active Patterns often looks like this.
let printParseResult = function
    | Int x -> printfn "%d" x
    | Double x -> printfn "%f" x
    | Date d -> printfn "%s" (d.ToString())
    | TimeSpan t -> printfn "%s" (t.ToString())
    | _ -> printfn "Nothing was parse-able!"

// Call the printer with some different values to parse.
printParseResult "12"
printParseResult "12.045"
printParseResult "12/28/2016"
printParseResult "9:01PM"
printParseResult "banana!"
```

Optional Types

One special case of Discriminated Union types is the Option Type, which is so useful that it's a part of the F# core library.

The Option Type is a type which represents one of two cases: a value, or nothing at all. It is used in any scenario where a value may or may not result from a particular operation. This then forces you to account for both cases, making it a compile-time concern rather than a runtime concern. These are often used in APIs where `null` is used to represent "nothing" instead, thus eliminating the need to worry about `NullReferenceException` in many circumstances.

```
/// Option values are any kind of value tagged with either 'Some' or 'None'.
/// They are used extensively in F# code to represent the cases where many other
/// languages would use null references.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/options
module OptionValues =

    /// First, define a zip code defined via Single-case Discriminated Union.
    type ZipCode = ZipCode of string

    /// Next, define a type where the ZipCode is optional.
    type Customer = { ZipCode: ZipCode option }

    /// Next, define an interface type that represents an object to compute the shipping zone for the
customer's zip code,
    /// given implementations for the 'getState' and 'getShippingZone' abstract methods.
    type IShippingCalculator =
        abstract GetState : ZipCode -> string option
        abstract GetShippingZone : string -> int

    /// Next, calculate a shipping zone for a customer using a calculator instance.
    /// This uses combinators in the Option module to allow a functional pipeline for
    /// transforming data with Optionals.
    let CustomerShippingZone (calculator: IShippingCalculator, customer: Customer) =
        customer.ZipCode
        |> Option.bind calculator.GetState
        |> Option.map calculator.GetShippingZone
```

Units of Measure

One unique feature of F#'s type system is the ability to provide context for numeric literals through Units of Measure.

Units of Measure allow you to associate a numeric type to a unit, such as Meters, and have functions perform work on units rather than numeric literals. This enables the compiler to verify that the types of numeric literals passed in make sense under a certain context, thus eliminating runtime errors associated with that kind of work.

```
/// Units of measure are a way to annotate primitive numeric types in a type-safe way.
/// You can then perform type-safe arithmetic on these values.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/units-of-measure
module UnitsOfMeasure =

    /// First, open a collection of common unit names
    open Microsoft.FSharp.Data.UnitSystems.SI.UnitNames

    /// Define a unitized constant
    let sampleValue1 = 1600.0<meter>

    /// Next, define a new unit type
    [<Measure>]
    type mile =
        /// Conversion factor mile to meter.
        static member asMeter = 1609.34<meter/mile>

    /// Define a unitized constant
    let sampleValue2  = 500.0<mile>

    /// Compute  metric-system constant
    let sampleValue3 = sampleValue2 * mile.asMeter

    // Values using Units of Measure can be used just like the primitive numeric type for things like
printing.
    printfn "After a %f race I would walk %f miles which would be %f meters" sampleValue1 sampleValue2
sampleValue3
```

The F# Core library defines many SI unit types and unit conversions. To learn more, check out the Microsoft.FSharp.Data.UnitSystems.SI Namespace.

Classes and Interfaces

F# also has full support for .NET classes, Interfaces, Abstract Classes, Inheritance, and so on.

Classes are types that represent .NET objects, which can have properties, methods, and events as its Members.

```
/// Classes are a way of defining new object types in F#, and support standard Object-oriented constructs.
/// They can have a variety of members (methods, properties, events, etc.)
///
/// To learn more about Classes, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/classes
///
/// To learn more about Members, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/members
module DefiningClasses =

    /// A simple two-dimensional Vector class.
    ///
    /// The class's constructor is on the first line,
    /// and takes two arguments: dx and dy, both of type 'double'.
    type Vector2D(dx : double, dy : double) =

        /// This internal field stores the length of the vector, computed when the
        /// object is constructed
        let length = sqrt (dx*dx + dy*dy)

        // 'this' specifies a name for the object's self-identifier.
        // In instance methods, it must appear before the member name.
        member this.DX = dx

        member this.DY = dy

        member this.Length = length

        /// This member is a method.  The previous members were properties.
        member this.Scale(k) = Vector2D(k * this.DX, k * this.DY)

    /// This is how you instantiate the Vector2D class.
    let vector1 = Vector2D(3.0, 4.0)

    /// Get a new scaled vector object, without modifying the original object.
    let vector2 = vector1.Scale(10.0)

    printfn "Length of vector1: %f\nLength of vector2: %f" vector1.Length vector2.Length
```

Defining generic classes is also very straightforward.

```
/// Generic classes allow types to be defined with respect to a set of type parameters.
/// In the following, 'T is the type parameter for the class.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/generics/
module DefiningGenericClasses =

    type StateTracker<'T>(initialElement: 'T) =

        /// This internal field store the states in a list.
        let mutable states = [ initialElement ]

        /// Add a new element to the list of states.
        member this.UpdateState newState =
            states <- newState :: states  // use the '<-' operator to mutate the value.

        /// Get the entire list of historical states.
        member this.History = states

        /// Get the latest state.
        member this.Current = states.Head

    /// An 'int' instance of the state tracker class. Note that the type parameter is inferred.
    let tracker = StateTracker 10

    // Add a state
    tracker.UpdateState 17
```

To implement an Interface, you can use either `interface ... with` syntax or an Object Expression.

```
/// Interfaces are object types with only 'abstract' members.
/// Object types and object expressions can implement interfaces.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/interfaces
module ImplementingInterfaces =

    /// This is a type that implements IDisposable.
    type ReadFile() =

        let file = new System.IO.StreamReader("readme.txt")

        member this.ReadLine() = file.ReadLine()

        // This is the implementation of IDisposable members.
        interface System.IDisposable with
            member this.Dispose() = file.Close()


    /// This is an object that implements IDisposable via an Object Expression
    /// Unlike other languages such as C# or Java, a new type definition is not needed
    /// to implement an interface.
    let interfaceImplementation =
        { new System.IDisposable with
            member this.Dispose() = printfn "disposed" }
```

Which Types to Use

The presence of Classes, Records, Discriminated Unions, and Tuples leads to an important question: which should you use? Like most everything in life, the answer depends on your circumstances.

Tuples are great for returning multiple values from a function, and using an ad-hoc aggregate of values as a value itself.

Records are a "step up" from Tuples, having named labels and support for optional members. They are great for

a low-ceremony representation of data in-transit through your program. Because they have structural equality, they are easy to use with comparison.

Discriminated Unions have many uses, but the core benefit is to be able to utilize them in conjunction with Pattern Matching to account for all possible "shapes" that a data can have.

Classes are great for a huge number of reasons, such as when you need to represent information and also tie that information to functionality. As a rule of thumb, when you have functionality which is conceptually tied to some data, using Classes and the principles of Object-Oriented Programming is a big benefit. Classes are also the preferred data type when interoperating with C# and Visual Basic, as these languages use classes for nearly everything.

Next Steps

Now that you've seen some of the primary features of the language, you should be ready to write your first F# programs! Check out Getting Started to learn how to set up your development environment and write some code.

The next steps for learning more can be whatever you like, but we recommend Introduction to Functional Programming in F# to get comfortable with core Functional Programming concepts. These will be essential in building robust programs in F#.

Also, check out the F# Language Reference to see a comprehensive collection of conceptual content on F#.

Introduction to Functional Programming in F#

3/19/2019 • 8 minutes to read • Edit Online

Functional programming is a style of programming that emphasizes the use of functions and immutable data. Typed functional programming is when functional programming is combined with static types, such as with F#. In general, the following concepts are emphasized in functional programming:

- Functions as the primary constructs you use
- Expressions instead of statements
- Immutable values over variables
- Declarative programming over imperative programming

Throughout this series, you'll explore concepts and patterns in functional programming using F#. Along the way, you'll learn some F# too.

Terminology

Functional programming, like other programming paradigms, comes with a vocabulary that you will eventually need to learn. Here are some common terms you'll see all of the time:

- **Function** - A function is a construct that will produce an output when given an input. More formally, it *maps* an item from one set to another set. This formalism is lifted into the concrete in many ways, especially when using functions that operate on collections of data. It is the most basic (and important) concept in functional programming.
- **Expression** - An expression is a construct in code that produces a value. In F#, this value must be bound or explicitly ignored. An expression can be trivially replaced by a function call.
- **Purity** - Purity is a property of a function such that its return value is always the same for the same arguments, and that its evaluation has no side effects. A pure function depends entirely on its arguments.
- **Referential Transparency** - Referential Transparency is a property of expressions such that they can be replaced with their output without affecting a program's behavior.
- **Immutability** - Immutability means that a value cannot be changed in-place. This is in contrast with variables, which can change in place.

Examples

The following examples demonstrate these core concepts.

Functions

The most common and fundamental construct in functional programming is the function. Here's a simple function that adds 1 to an integer:

```
let addOne x = x + 1
```

Its type signature is as follows:

```
val addOne: x:int -> int
```

The signature can be read as, "`addOne` accepts an `int` named `x` and will produce an `int`". More formally,

`addOne` is *mapping* a value from the set of integers to the set of integers. The `->` token signifies this mapping. In F#, you can usually look at the function signature to get a sense for what it does.

So, why is the signature important? In typed functional programming, the implementation of a function is often less important than the actual type signature! The fact that `addOne` adds the value 1 to an integer is interesting at runtime, but when you are constructing a program, the fact that it accepts and returns an `int` is what informs how you will actually use this function. Furthermore, once you use this function correctly (with respect to its type signature), diagnosing any problems can be done only within the body of the `addOne` function. This is the impetus behind typed functional programming.

Expressions

Expressions are constructs that evaluate to a value. In contrast to statements, which perform an action, expressions can be thought of performing an action that gives back a value. Expressions are almost always used in favor of statements in functional programming.

Consider the previous function, `addOne`. The body of `addOne` is an expression:

```
// 'x + 1' is an expression!
let addOne x = x + 1
```

It is the result of this expression that defines the result type of the `addOne` function. For example, the expression that makes up this function could be changed to be a different type, such as a `string`:

```
let addOne x = x.ToString() + "1"
```

The signature of the function is now:

```
val addOne: x:'a -> string
```

Since any type in F# can have `ToString()` called on it, the type of `x` has been made generic (called Automatic Generalization), and the resultant type is a `string`.

Expressions are not just the bodies of functions. You can have expressions that produce a value you use elsewhere. A common one is `if`:

```
// Checks if 'x' is odd by using the mod operator
let isOdd x = x % 2 <> 0

let addOneIfOdd input =
    let result =
        if isOdd input then
            input + 1
        else
            input

    result
```

The `if` expression produces a value called `result`. Note that you could omit `result` entirely, making the `if` expression the body of the `addOneIfOdd` function. The key thing to remember about expressions is that they produce a value.

There is a special type, `unit`, that is used when there is nothing to return. For example, consider this simple function:

```
let printString (str: string) =
    printfn "String is: %s" str
```

The signature looks like this:

```
val printString: str:string -> unit
```

The `unit` type indicates that there is no actual value being returned. This is useful when you have a routine that must "do work" despite having no value to return as a result of that work.

This is in sharp contrast to imperative programming, where the equivalent `if` construct is a statement, and producing values is often done with mutating variables. For example, in C#, the code might be written like this:

```
bool IsOdd(int x) => x % 2 != 0;

int AddOneIfOdd(int input)
{
    var result = input;

    if (IsOdd(input))
    {
        result = input + 1;
    }

    return result;
}
```

It's worth noting that C# and other C-style languages do support the ternary expression, which allows for expression-based conditional programming.

In functional programming, it is rare to mutate values with statements. Although some functional languages support statements and mutation, it is not common to use these concepts in functional programming.

Pure functions

As previously mentioned, pure functions are functions that:

- Always evaluate to the same value for the same input.
- Have no side effects.

It is helpful to think of mathematical functions in this context. In mathematics, functions depend only on their arguments and do not have any side effects. In the mathematical function `f(x) = x + 1`, the value of `f(x)` depends only on the value of `x`. Pure functions in functional programming are the same way.

When writing a pure function, the function must depend only on its arguments and not perform any action that results in a side effect.

Here is an example of a non-pure function because it depends on global, mutable state:

```
let mutable value = 1

let addOneToValue x = x + value
```

The `addOneToValue` function is clearly impure, because `value` could be changed at any time to have a different value than 1. This pattern of depending on a global value is to be avoided in functional programming.

Here is another example of a non-pure function, because it performs a side effect:

```
let addOneToValue x =
    printfn "x is %d" x
    x + 1
```

Although this function does not depend on a global value, it writes the value of `x` to the output of the program. Although there is nothing inherently wrong with doing this, it does mean that the function is not pure. If another part of your program depends on something external to the program, such as the output buffer, then calling this function can affect that other part of your program.

Removing the `printfn` statement makes the function pure:

```
let addOneToValue x = x + 1
```

Although this function is not inherently *better* than the previous version with the `printfn` statement, it does guarantee that all this function does is return a value. Calling this function any number of times produces the same result: it just produces a value. The predictability given by purity is something many functional programmers strive for.

Immutability

Finally, one of the most fundamental concepts of typed functional programming is immutability. In F#, all values are immutable by default. That means they cannot be mutated in-place unless you explicitly mark them as mutable.

In practice, working with immutable values means that you change your approach to programming from, "I need to change something", to "I need to produce a new value".

For example, adding 1 to a value means producing a new value, not mutating the existing one:

```
let value = 1
let secondValue = value + 1
```

In F#, the following code does **not** mutate the `value` function; instead, it performs an equality check:

```
let value = 1
value = value + 1 // Produces a 'bool' value!
```

Some functional programming languages do not support mutation at all. In F#, it is supported, but it is not the default behavior for values.

This concept extends even further to data structures. In functional programming, immutable data structures such as sets (and many more) have a different implementation than you might initially expect. Conceptually, something like adding an item to a set does not change the set, it produces a *new* set with the added value. Under the covers, this is often accomplished by a different data structure that allows for efficiently tracking a value so that the appropriate representation of the data can be given as a result.

This style of working with values and data structures is critical, as it forces you to treat any operation that modifies something as if it creates a new version of that thing. This allows for things like equality and comparability to be consistent in your programs.

Next steps

The next section will thoroughly cover functions, exploring different ways you can use them in functional programming.

First-class functions explores functions deeply, showing how you can use them in various contexts.

Further reading

The Thinking Functionally series is another great resource to learn about functional programming with F#. It covers fundamentals of functional programming in a pragmatic and easy-to-read way, using F# features to illustrate the concepts.

First-class functions

7/30/2019 • 22 minutes to read • Edit Online

A defining characteristic of functional programming languages is the elevation of functions to first-class status. You should be able to do with a function whatever you can do with values of the other built-in types, and be able to do so with a comparable degree of effort.

Typical measures of first-class status include the following:

- Can you bind functions to identifiers? That is, can you give them names?
- Can you store functions in data structures, such as in a list?
- Can you pass a function as an argument in a function call?
- Can you return a function from a function call?

The last two measures define what are known as *higher-order operations* or *higher-order functions*. Higher-order functions accept functions as arguments and return functions as the value of function calls. These operations support such mainstays of functional programming as mapping functions and composition of functions.

Give the Value a Name

If a function is a first-class value, you must be able to name it, just as you can name integers, strings, and other built-in types. This is referred to in functional programming literature as binding an identifier to a value. F# uses `let` bindings to bind names to values: `let <identifier> = <value>`. The following code shows two examples.

```
// Integer and string.
let num = 10
let str = "F#"
```

You can name a function just as easily. The following example defines a function named `squareIt` by binding the identifier `squareIt` to the lambda expression `fun n -> n * n`. Function `squareIt` has one parameter, `n`, and it returns the square of that parameter.

```
let squareIt = fun n -> n * n
```

F# provides the following more concise syntax to achieve the same result with less typing.

```
let squareIt2 n = n * n
```

The examples that follow mostly use the first style, `let <function-name> = <lambda-expression>`, to emphasize the similarities between the declaration of functions and the declaration of other types of values. However, all the named functions can also be written with the concise syntax. Some of the examples are written in both ways.

Store the Value in a Data Structure

A first-class value can be stored in a data structure. The following code shows examples that store values in lists and in tuples.

```
// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )
```

To verify that a function name stored in a tuple does in fact evaluate to a function, the following example uses the `fst` and `snd` operators to extract the first and second elements from tuple `funAndArgTuple`. The first element in the tuple is `squareIt` and the second element is `num`. Identifier `num` is bound in a previous example to integer 10, a valid argument for the `squareIt` function. The second expression applies the first element in the tuple to the second element in the tuple: `squareIt num`.

```
// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))
```

Similarly, just as identifier `num` and integer 10 can be used interchangeably, so can identifier `squareIt` and lambda expression `fun n -> n * n`.

```
// Make a tuple of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))
```

Pass the Value as an Argument

If a value has first-class status in a language, you can pass it as an argument to a function. For example, it is common to pass integers and strings as arguments. The following code shows integers and strings passed as arguments in F#.

```
// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)
```

If functions have first-class status, you must be able to pass them as arguments in the same way. Remember that this is the first characteristic of higher-order functions.

In the following example, function `applyIt` has two parameters, `op` and `arg`. If you send in a function that has one parameter for `op` and an appropriate argument for the function to `arg`, the function returns the result of applying `op` to `arg`. In the following example, both the function argument and the integer argument are sent in the same way, by using their names.

```
// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)
```

The ability to send a function as an argument to another function underlies common abstractions in functional programming languages, such as map or filter operations. A map operation, for example, is a higher-order function that captures the computation shared by functions that step through a list, do something to each element, and then return a list of the results. You might want to increment each element in a list of integers, or to square each element, or to change each element in a list of strings to uppercase. The error-prone part of the computation is the recursive process that steps through the list and builds a list of the results to return. That part is captured in the mapping function. All you have to write for a particular application is the function that you want to apply to each list element individually (adding, squaring, changing case). That function is sent as an argument to the mapping

function, just as `squareIt` is sent to `applyIt` in the previous example.

F# provides map methods for most collection types, including lists, arrays, and sequences. The following examples use lists. The syntax is `List.map <the function> <the list>`.

```
// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot
```

For more information, see Lists.

Return the Value from a Function Call

Finally, if a function has first-class status in a language, you must be able to return it as the value of a function call, just as you return other types, such as integers and strings.

The following function calls return integers and display them.

```
// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)
```

The following function call returns a string.

```
// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()
```

The following function call, declared inline, returns a Boolean value. The value displayed is `True`.

```
System.Console.WriteLine((fun n -> n % 2 = 1) 15)
```

The ability to return a function as the value of a function call is the second characteristic of higher-order functions. In the following example, `checkFor` is defined to be a function that takes one argument, `item`, and returns a new function as its value. The returned function takes a list as its argument, `lst`, and searches for `item` in `lst`. If `item` is present, the function returns `true`. If `item` is not present, the function returns `false`. As in the previous section, the following code uses a provided list function, List.exists, to search the list.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

The following code uses `checkFor` to create a new function that takes one argument, a list, and searches for 7 in the list.

```
// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)
```

The following example uses the first-class status of functions in F# to declare a function, `compose`, that returns a composition of two function arguments.

```
// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                          op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                      op1 (op2 n)
    funToReturn
```

> **NOTE**
>
> For an even shorter version, see the following section, "Curried Functions."

The following code sends two functions as arguments to `compose`, both of which take a single argument of the same type. The return value is a new function that is a composition of the two function arguments.

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

NOTE

F# provides two operators, `<<` and `>>`, that compose functions. For example, `let squareAndDouble2 = doubleIt << squareIt` is equivalent to `let squareAndDouble = compose doubleIt squareIt` in the previous example.

The following example of returning a function as the value of a function call creates a simple guessing game. To create a game, call `makeGame` with the value that you want someone to guess sent in for `target`. The return value from function `makeGame` is a function that takes one argument (the guess) and reports whether the guess is correct.

```
let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                   if guess = target then
                      System.Console.WriteLine("You win!")
                   else
                      System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game
```

The following code calls `makeGame`, sending the value `7` for `target`. Identifier `playGame` is bound to the returned lambda expression. Therefore, `playGame` is a function that takes as its one argument a value for `guess`.

```
let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!
```

Curried Functions

Many of the examples in the previous section can be written more concisely by taking advantage of the implicit *currying* in F# function declarations. Currying is a process that transforms a function that has more than one parameter into a series of embedded functions, each of which has a single parameter. In F#, functions that have more than one parameter are inherently curried. For example, `compose` from the previous section can be written as shown in the following concise style, with three parameters.

```
let compose4 op1 op2 n = op1 (op2 n)
```

However, the result is a function of one parameter that returns a function of one parameter that in turn returns another function of one parameter, as shown in `compose4curried`.

```
let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)
```

You can access this function in several ways. Each of the following examples returns and displays 18. You can replace `compose4` with `compose4curried` in any of the examples.

```
// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)
```

To verify that the function still works as it did before, try the original test cases again.

```
let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)
```

> **NOTE**
>
> You can restrict currying by enclosing parameters in tuples. For more information, see "Parameter Patterns" in Parameters and Arguments.

The following example uses implicit currying to write a shorter version of `makeGame`. The details of how `makeGame` constructs and returns the `game` function are less explicit in this format, but you can verify by using the original test cases that the result is the same.

```
let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

For more information about currying, see "Partial Application of Arguments" in Functions.

Identifier and Function Definition Are Interchangeable

The variable name `num` in the previous examples evaluates to the integer 10, and it is no surprise that where `num` is valid, 10 is also valid. The same is true of function identifiers and their values: anywhere the name of the function can be used, the lambda expression to which it is bound can be used.

The following example defines a `Boolean` function called `isNegative`, and then uses the name of the function and the definition of the function interchangeably. The next three examples all return and display `False`.

```
let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)
```

To take it one step further, substitute the value that `applyIt` is bound to for `applyIt`.

```
System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)
```

Functions Are First-Class Values in F#

The examples in the previous sections demonstrate that functions in F# satisfy the criteria for being first-class values in F#:

- You can bind an identifier to a function definition.

```
let squareIt = fun n -> n * n
```

- You can store a function in a data structure.

```
let funTuple2 = ( BMICalculator, fun n -> n * n )
```

- You can pass a function as an argument.

```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]
```

- You can return a function as the value of a function call.

```
let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn
```

For more information about F#, see the F# Language Reference.

Example

Description

The following code contains all the examples in this topic.

Code

```
// ** GIVE THE VALUE A NAME **


// Integer and string.
let num = 10
let str = "F#"



let squareIt = fun n -> n * n



let squareIt2 n = n * n



// ** STORE THE VALUE IN A DATA STRUCTURE **


// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator
```

```
// Index (BMI) calculator.
let BMICalculator = fun ht wt ->
                    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]


// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )



// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))



// Make a list of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))



// ** PASS THE VALUE AS AN ARGUMENT **


// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)



// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
```

```
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)



// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot



// ** RETURN THE VALUE FROM A FUNCTION CALL **


// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)


// The following function call returns a string:

// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()



System.Console.WriteLine((fun n -> n % 2 = 1) 15)



let checkFor item =
    let functionToReturn = fun lst ->
                           List.exists (fun a -> a = item) lst
    functionToReturn



// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
```

```
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)



// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expresson syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
                            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
                        op1 (op2 n)
    funToReturn



// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)



let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
                   if guess = target then
                      System.Console.WriteLine("You win!")
                   else
                      System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game



let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
```

```
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!



// ** CURRIED FUNCTIONS **


let compose4 op1 op2 n = op1 (op2 n)



let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)



// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)



let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)



let makeGame2 target guess =
    if guess = target then
       System.Console.WriteLine("You win!")
    else
       System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'
```

```
// ** IDENTIFIER AND FUNCTION DEFINITION ARE INTERCHANGEABLE **


let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)



System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0)  10)



// ** FUNCTIONS ARE FIRST-CLASS VALUES IN F# **

//let squareIt = fun n -> n * n


let funTuple2 = ( BMICalculator, fun n -> n * n )



let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]


//let checkFor item =
//    let functionToReturn = fun lst ->
//                           List.exists (fun a -> a = item) lst
//    functionToReturn
```

See also

- Lists
- Tuples
- Functions
- `let` Bindings
- Lambda Expressions: The `fun` Keyword

Async Programming in F#

4/24/2019 • 6 minutes to read • Edit Online

NOTE

Some inaccuracies have been discovered in this article. It is being rewritten. See Issue #666 to learn about the changes.

Async programming in F# can be accomplished through a language-level programming model designed to be easy to use and natural to the language.

The core of async programming in F# is `Async<'T>`, a representation of work that can be triggered to run in the background, where `'T` is either the type returned via the special `return` keyword or `unit` if the async workflow has no result to return.

The key concept to understand is that an async expression's type is `Async<'T>`, which is merely a *specification* of work to be done in an asynchronous context. It is not executed until you explicitly start it with one of the starting functions (such as `Async.RunSynchronously`). Although this is a different way of thinking about doing work, it ends up being quite simple in practice.

For example, say you wanted to download the HTML from dotnetfoundation.org without blocking the main thread. You can accomplish it like this:

```
open System
open System.Net

let fetchHtmlAsync url =
    async {
        let uri = Uri(url)
        use webClient = new WebClient()

        // Execution of fetchHtmlAsync won't continue until the result
        // of AsyncDownloadString is bound.
        let! html = webClient.AsyncDownloadString(uri)
        return html
    }

let html = "https://dotnetfoundation.org" |> fetchHtmlAsync |> Async.RunSynchronously
printfn "%s" html
```

And that's it! Aside from the use of `async`, `let!`, and `return`, this is just normal F# code.

There are a few syntactical constructs which are worth noting:

- `let!` binds the result of an async expression (which runs on another context).
- `use!` works just like `let!`, but disposes its bound resources when it goes out of scope.
- `do!` will await an async workflow which doesn't return anything.
- `return` simply returns a result from an async expression.
- `return!` executes another async workflow and returns its return value as a result.

Additionally, normal `let`, `use`, and `do` keywords can be used alongside the async versions just as they would in a normal function.

How to start Async Code in F#

As mentioned earlier, async code is a specification of work to be done in another context which needs to be explicitly started. Here are two primary ways to accomplish this:

1. `Async.RunSynchronously` will start an async workflow on another thread and await its result.

```
open System
open System.Net

let fetchHtmlAsync url =
    async {
        let uri = Uri(url)
        use webClient = new WebClient()
        let! html = webClient.AsyncDownloadString(uri)
        return html
    }

// Execution will pause until fetchHtmlAsync finishes
let html = "https://dotnetfoundation.org" |> fetchHtmlAsync |> Async.RunSynchronously

// you actually have the result from fetchHtmlAsync now!
printfn "%s" html
```

2. `Async.Start` will start an async workflow on another thread, and will **not** await its result.

```
open System
open System.Net

let uploadDataAsync url data =
    async {
        let uri = Uri(url)
        use webClient = new WebClient()
        webClient.UploadStringAsync(uri, data)
    }

let workflow = uploadDataAsync "https://url-to-upload-to.com" "hello, world!"

// Execution will continue after calling this!
Async.Start(workflow)

printfn "%s" "uploadDataAsync is running in the background..."
```

There are other ways to start an async workflow available for more specific scenarios. They are detailed in the Async reference.

A Note on Threads

The phrase "on another thread" is mentioned above, but it is important to know that **this does not mean that async workflows are a facade for multithreading**. The workflow actually "jumps" between threads, borrowing them for a small amount of time to do useful work. When an async workflow is effectively "waiting" (for example, waiting for a network call to return something), any thread it was borrowing at the time is freed up to go do useful work on something else. This allows async workflows to utilize the system they run on as effectively as possible, and makes them especially strong for high-volume I/O scenarios.

How to Add Parallelism to Async Code

Sometimes you may need to perform multiple asynchronous jobs in parallel, collect their results, and interpret them in some way. `Async.Parallel` allows you to do this without needing to use the Task Parallel Library, which would involve needing to coerce `Task<'T>` and `Async<'T>` types.

The following example will use `Async.Parallel` to download the HTML from four popular sites in parallel, wait for those tasks to complete, and then print the HTML which was downloaded.

```
open System
open System.Net

let urlList =
    [ "https://www.microsoft.com"
      "https://www.google.com"
      "https://www.amazon.com"
      "https://www.facebook.com" ]

let fetchHtmlAsync url =
    async {
        let uri = Uri(url)
        use webClient = new WebClient()
        let! html = webClient.AsyncDownloadString(uri)
        return html
    }

let getHtmlList urls =
    urls
    |> Seq.map fetchHtmlAsync   // Build an Async<'T> for each site
    |> Async.Parallel           // Returns an Async<'T []>
    |> Async.RunSynchronously   // Wait for the result of the parallel work

let htmlList = getHtmlList urlList

// We now have the downloaded HTML for each site!
for html in htmlList do
    printfn "%s" html
```

Important Info and Advice

- Append "Async" to the end of any functions you'll consume

Although this is just a naming convention, it does make things like API discoverability easier. Particularly if there are synchronous and asynchronous versions of the same routine, it's a good idea to explicitly state which is asynchronous via the name.

- Listen to the compiler!

F#'s compiler is very strict, making it nearly impossible to do something troubling like run "async" code synchronously. If you come across a warning, that's a sign that the code won't execute how you think it will. If you can make the compiler happy, your code will most likely execute as expected.

For the C#/VB Programmer Looking Into F#

This section assumes you're familiar with the async model in C#/VB. If you are not, Async Programming in C# is a starting point.

There is a fundamental difference between the C#/VB async model and the F# async model.

When you call a function which returns a `Task` or `Task<'T>`, that job has already begun execution. The handle returned represents an already-running asynchronous job. In contrast, when you call an async function in F#, the `Async<'a>` returned represents a job which will be **generated** at some point. Understanding this model is powerful, because it allows for asynchronous jobs in F# to be chained together easier, performed conditionally, and be started with a finer grain of control.

There are a few other similarities and differences worth noting.

Similarities

- `let!`, `use!`, and `do!` are analogous to `await` when calling an async job from within an `async{ }` block.

  The three keywords can only be used within an `async { }` block, similar to how `await` can only be invoked inside an `async` method. In short, `let!` is for when you want to capture and use a result, `use!` is the same but for something whose resources should get cleaned after it's used, and `do!` is for when you want to wait for an async workflow with no return value to finish before moving on.

- F# supports data-parallelism in a similar way.

  Although it operates very differently, `Async.Parallel` corresponds to `Task.WhenAll` for the scenario of wanting the results of a set of async jobs when they all complete.

Differences

- Nested `let!` is not allowed, unlike nested `await`

  Unlike `await`, which can be nested indefinitely, `let!` cannot and must have its result bound before using it inside of another `let!`, `do!`, or `use!`.

- Cancellation support is simpler in F# than in C#/VB.

  Supporting cancellation of a task midway through its execution in C#/VB requires checking the `IsCancellationRequested` property or calling `ThrowIfCancellationRequested()` on a `CancellationToken` object that's passed into the async method.

In contrast, F# async workflows are more naturally cancellable. Cancellation is a simple three-step process.

1. Create a new `CancellationTokenSource`.
2. Pass it into a starting function.
3. Call `Cancel` on the token.

Example:

```
open System.Threading

// Create a workflow which will loop forever.
let workflow =
    async {
        while true do
            printfn "Working..."
            do! Async.Sleep 1000
    }

let tokenSource = new CancellationTokenSource()

// Start the workflow in the background
Async.Start (workflow, tokenSource.Token)

// Executing the next line will stop the workflow
tokenSource.Cancel()
```

And that's it!

Further resources:

- Async Workflows on MSDN
- Asynchronous Sequences for F#
- F# Data HTTP Utilities

Type Providers

5/15/2019 • 2 minutes to read • Edit Online

An F# type provider is a component that provides types, properties, and methods for use in your program. Type Providers generate what are known as **Provided Types**, which are generated by the F# compiler and are based on an external data source.

For example, an F# Type Provider for SQL can generate types representing tables and columns in a relational database. In fact, this is what the SQLProvider Type Provider does.

Provided Types depend on input parameters to a Type Provider. Such input can be a sample data source (such as a JSON schema file), a URL pointing directly to an external service, or a connection string to a data source. A Type Provider can also ensure that groups of types are only expanded on demand; that is, they are expanded if the types are actually referenced by your program. This allows for the direct, on-demand integration of large-scale information spaces such as online data markets in a strongly typed way.

Generative and Erased Type Providers

Type Providers come in two forms: Generative and Erased.

Generative Type Providers produce types that can be written as .NET types into the assembly in which they are produced. This allows them to be consumed from code in other assemblies. This means that the typed representation of the data source must generally be one that is feasible to represent with .NET types.

Erasing Type Providers produce types that can only be consumed in the assembly or project they are generated from. The types are ephemeral; that is, they are not written into an assembly and cannot be consumed by code in other assemblies. They can contain *delayed* members, allowing you to use provided types from a potentially infinite information space. They are useful for using a small subset of a large and interconnected data source.

Commonly used Type Providers

The following widely-used libraries contain Type Providers for different uses:

- FSharp.Data includes Type Providers for JSON, XML, CSV, and HTML document formats and resources.
- SQLProvider provides strongly-typed access to relation databases through object mapping and F# LINQ queries against these data sources.
- FSharp.Data.SqlClient has a set of type providers for compile-time checked embedding of T-SQL in F#.
- Azure Storage Type provider provides types for Azure Blobs, Tables, and Queues, allowing you to access these resources without needing to specify resource names as strings throughout your program.
- FSharp.Data.GraphQL contains the **GraphQLProvider**, which provides types based on a GraphQL server specified by URL.

Where necessary, you can create your own custom type providers, or reference type providers that have been created by others. For example, assume your organization has a data service providing a large and growing number of named data sets, each with its own stable data schema. You may choose to create a type provider that reads the schemas and presents the latest available data sets to the programmer in a strongly typed way.

See also

- Tutorial: Create a Type Provider
- F# Language Reference

- Visual F#

Tutorial: Create a Type Provider

3/8/2019 • 36 minutes to read • Edit Online

The type provider mechanism in F# is a significant part of its support for information rich programming. This tutorial explains how to create your own type providers by walking you through the development of several simple type providers to illustrate the basic concepts. For more information about the type provider mechanism in F#, see Type Providers.

The F# ecosystem contains a range of type providers for commonly used Internet and enterprise data services. For example:

- FSharp.Data includes type providers for JSON, XML, CSV and HTML document formats.
- SQLProvider provides strongly-typed access to SQL databases through a object mapping and F# LINQ queries against these data sources.
- FSharp.Data.SqlClient has a set of type providers for compile-time checked embedding of T-SQL in F#.
- FSharp.Data.TypeProviders is an older set of type providers for use only with .NET Framework programming for accessing SQL, Entity Framework, OData and WSDL data services.

Where necessary, you can create custom type providers, or you can reference type providers that others have created. For example, your organization could have a data service that provides a large and growing number of named data sets, each with its own stable data schema. You can create a type provider that reads the schemas and presents the current data sets to the programmer in a strongly typed way.

Before You Start

The type provider mechanism is primarily designed for injecting stable data and service information spaces into the F# programming experience.

This mechanism isn't designed for injecting information spaces whose schema changes during program execution in ways that are relevant to program logic. Also, the mechanism isn't designed for intra-language meta-programming, even though that domain contains some valid uses. You should use this mechanism only where necessary and where the development of a type provider yields very high value.

You should avoid writing a type provider where a schema isn't available. Likewise, you should avoid writing a type provider where an ordinary (or even an existing) .NET library would suffice.

Before you start, you might ask the following questions:

- Do you have a schema for your information source? If so, what's the mapping into the F# and .NET type system?
- Can you use an existing (dynamically typed) API as a starting point for your implementation?
- Will you and your organization have enough uses of the type provider to make writing it worthwhile? Would a normal .NET library meet your needs?
- How much will your schema change?
- Will it change during coding?
- Will it change between coding sessions?
- Will it change during program execution?

Type providers are best suited to situations where the schema is stable at runtime and during the lifetime of compiled code.

A Simple Type Provider

This sample is Samples.HelloWorldTypeProvider, similar to the samples in the `examples` directory of the F# Type Provider SDK. The provider makes available a "type space" that contains 100 erased types, as the following code shows by using F# signature syntax and omitting the details for all except `Type1`. For more information about erased types, see Details About Erased Provided Types later in this topic.

```
namespace Samples.HelloWorldTypeProvider

type Type1 =
    /// This is a static property.
    static member StaticProperty : string

    /// This constructor takes no arguments.
    new : unit -> Type1

    /// This constructor takes one argument.
    new : data:string -> Type1

    /// This is an instance property.
    member InstanceProperty : int

    /// This is an instance method.
    member InstanceMethod : x:int -> char

    nested type NestedType =
        /// This is StaticProperty1 on NestedType.
        static member StaticProperty1 : string
        …
        /// This is StaticProperty100 on NestedType.
        static member StaticProperty100 : string

type Type2 =
…
…

type Type100 =
…
```

Note that the set of types and members provided is statically known. This example doesn't leverage the ability of providers to provide types that depend on a schema. The implementation of the type provider is outlined in the following code, and the details are covered in later sections of this topic.

> **WARNING**
> There may be differences between this code and the online samples.

```
namespace Samples.FSharp.HelloWorldTypeProvider

open System
open System.Reflection
open ProviderImplementation.ProvidedTypes
open FSharp.Core.CompilerServices
open FSharp.Quotations

// This type defines the type provider. When compiled to a DLL, it can be added
// as a reference to an F# command-line compilation, script, or project.
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =

  // Inheriting from this type provides implementations of ITypeProvider
  // in terms of the provided types below.
  inherit TypeProviderForNamespaces(config)

  let namespaceName = "Samples.HelloWorldTypeProvider"
  let thisAssembly = Assembly.GetExecutingAssembly()

  // Make one provided type, called TypeN.
  let makeOneProvidedType (n:int) =
  …
  // Now generate 100 types
  let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

  // And add them to the namespace
  do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()
```

To use this provider, open a separate instance of Visual Studio, create an F# script, and then add a reference to the provider from your script by using #r as the following code shows:

```
#r @".\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]
[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35
```

Then look for the types under the `Samples.HelloWorldTypeProvider` namespace that the type provider generated.

Before you recompile the provider, make sure that you have closed all instances of Visual Studio and F# Interactive that are using the provider DLL. Otherwise, a build error will occur because the output DLL will be locked.

To debug this provider by using print statements, make a script that exposes a problem with the provider, and then use the following code:

```
fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

To debug this provider by using Visual Studio, open the Developer Command Prompt for Visual Studio with administrative credentials, and run the following command:

```
devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

As an alternative, open Visual Studio, open the Debug menu, choose `Debug/Attach to process…`, and attach to another `devenv` process where you're editing your script. By using this method, you can more easily target particular logic in the type provider by interactively typing expressions into the second instance (with full IntelliSense and other features).

You can disable Just My Code debugging to better identify errors in generated code. For information about how to enable or disable this feature, see Navigating through Code with the Debugger. Also, you can also set first-chance exception catching by opening the `Debug` menu and then choosing `Exceptions` or by choosing the Ctrl+Alt+E keys to open the `Exceptions` dialog box. In that dialog box, under `Common Language Runtime Exceptions`, select the `Thrown` check box.

Implementation of the Type Provider

This section walks you through the principal sections of the type provider implementation. First, you define the type for the custom type provider itself:

```
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =
```

This type must be public, and you must mark it with the TypeProvider attribute so that the compiler will recognize the type provider when a separate F# project references the assembly that contains the type. The *config* parameter is optional, and, if present, contains contextual configuration information for the type provider instance that the F# compiler creates.

Next, you implement the ITypeProvider interface. In this case, you use the `TypeProviderForNamespaces` type from the `ProvidedTypes` API as a base type. This helper type can provide a finite collection of eagerly provided namespaces, each of which directly contains a finite number of fixed, eagerly provided types. In this context, the provider *eagerly* generates types even if they aren't needed or used.

```
inherit TypeProviderForNamespaces(config)
```

Next, define local private values that specify the namespace for the provided types, and find the type provider assembly itself. This assembly is used later as the logical parent type of the erased types that are provided.

```
let namespaceName = "Samples.HelloWorldTypeProvider"
let thisAssembly = Assembly.GetExecutingAssembly()
```

Next, create a function to provide each of the types Type1…Type100. This function is explained in more detail later in this topic.

```
let makeOneProvidedType (n:int) = …
```

Next, generate the 100 provided types:

```
let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]
```

Next, add the types as a provided namespace:

```
do this.AddNamespace(namespaceName, types)
```

Finally, add an assembly attribute that indicates that you are creating a type provider DLL:

```
[<assembly:TypeProviderAssembly>]
do()
```

Providing One Type And Its Members

The `makeOneProvidedType` function does the real work of providing one of the types.

```
let makeOneProvidedType (n:int) =
…
```

This step explains the implementation of this function. First, create the provided type (for example, Type1, when n = 1, or Type57, when n = 57).

```
// This is the provided type. It is an erased provided type and, in compiled code,
// will appear as type 'obj'.
let t = ProvidedTypeDefinition(thisAssembly, namespaceName,
                               "Type" + string n,
                               baseType = Some typeof<obj>)
```

You should note the following points:

- This provided type is erased. Because you indicate that the base type is `obj`, instances will appear as values of type obj in compiled code.
- When you specify a non-nested type, you must specify the assembly and namespace. For erased types, the assembly should be the type provider assembly itself.

Next, add XML documentation to the type. This documentation is delayed, that is, computed on-demand if the host compiler needs it.

```
t.AddXmlDocDelayed (fun () -> sprintf "This provided type %s" ("Type" + string n))
```

Next you add a provided static property to the type:

```
let staticProp = ProvidedProperty(propertyName = "StaticProperty",
                                  propertyType = typeof<string>,
                                  isStatic = true,
                                  getterCode = (fun args -> <@@ "Hello!" @@>))
```

Getting this property will always evaluate to the string "Hello!". The `GetterCode` for the property uses an F# quotation, which represents the code that the host compiler generates for getting the property. For more information about quotations, see Code Quotations (F#).

Add XML documentation to the property.

```
staticProp.AddXmlDocDelayed(fun () -> "This is a static property")
```

Now attach the provided property to the provided type. You must attach a provided member to one and only one type. Otherwise, the member will never be accessible.

```
t.AddMember staticProp
```

Now create a provided constructor that takes no parameters.

```
let ctor = ProvidedConstructor(parameters = [ ],
                               invokeCode = (fun args -> <@@ "The object data" :> obj @@>))
```

The `InvokeCode` for the constructor returns an F# quotation, which represents the code that the host compiler generates when the constructor is called. For example, you can use the following constructor:

```
new Type10()
```

An instance of the provided type will be created with underlying data "The object data". The quoted code includes a conversion to obj because that type is the erasure of this provided type (as you specified when you declared the provided type).

Add XML documentation to the constructor, and add the provided constructor to the provided type:

```
ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

t.AddMember ctor
```

Create a second provided constructor that takes one parameter:

```
let ctor2 =
ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],
                    invokeCode = (fun args -> <@@ (%%(args.[0]) : string) :> obj @@>))
```

The `InvokeCode` for the constructor again returns an F# quotation, which represents the code that the host compiler generated for a call to the method. For example, you can use the following constructor:

```
new Type10("ten")
```

An instance of the provided type is created with underlying data "ten". You may have already noticed that the `InvokeCode` function returns a quotation. The input to this function is a list of expressions, one per constructor parameter. In this case, an expression that represents the single parameter value is available in `args.[0]`. The code for a call to the constructor coerces the return value to the erased type `obj`. After you add the second provided constructor to the type, you create a provided instance property:

```
let instanceProp =
    ProvidedProperty(propertyName = "InstanceProperty",
                     propertyType = typeof<int>,
                     getterCode= (fun args ->
                        <@@ ((%%(args.[0]) : obj) :?> string).Length @@>))
instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")
t.AddMember instanceProp
```

Getting this property will return the length of the string, which is the representation object. The `GetterCode` property returns an F# quotation that specifies the code that the host compiler generates to get the property. Like `InvokeCode`, the `GetterCode` function returns a quotation. The host compiler calls this function with a list of arguments. In this case, the arguments include just the single expression that represents the instance upon which

the getter is being called, which you can access by using `args.[0]`.The implementation of `GetterCode` then splices into the result quotation at the erased type `obj`, and a cast is used to satisfy the compiler's mechanism for checking types that the object is a string. The next part of `makeOneProvidedType` provides an instance method with one parameter.

```
let instanceMeth =
    ProvidedMethod(methodName = "InstanceMethod",
                   parameters = [ProvidedParameter("x",typeof<int>)],
                   returnType = typeof<char>,
                   invokeCode = (fun args ->
                       <@@ ((%%(args.[0]) : obj) :?> string).Chars(%%(args.[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")
// Add the instance method to the type.
t.AddMember instanceMeth
```

Finally, create a nested type that contains 100 nested properties. The creation of this nested type and its properties is delayed, that is, computed on-demand.

```
t.AddMembersDelayed(fun () ->
  let nestedType = ProvidedTypeDefinition("NestedType", Some typeof<obj>)

  nestedType.AddMembersDelayed (fun () ->
    let staticPropsInNestedType =
      [ for i in 1 .. 100 do
          let valueOfTheProperty = "I am string "  + string i

          let p =
            ProvidedProperty(propertyName = "StaticProperty" + string i,
              propertyType = typeof<string>,
              isStatic = true,
              getterCode= (fun args -> <@@ valueOfTheProperty @@>))

          p.AddXmlDocDelayed(fun () ->
              sprintf "This is StaticProperty%d on NestedType" i)

          yield p ]

    staticPropsInNestedType)

  [nestedType])
```

Details about Erased Provided Types

The example in this section provides only *erased provided types*, which are particularly useful in the following situations:

- When you are writing a provider for an information space that contains only data and methods.
- When you are writing a provider where accurate runtime-type semantics aren't critical for practical use of the information space.
- When you are writing a provider for an information space that is so large and interconnected that it isn't technically feasible to generate real .NET types for the information space.

In this example, each provided type is erased to type `obj`, and all uses of the type will appear as type `obj` in compiled code. In fact, the underlying objects in these examples are strings, but the type will appear as `System.Object` in .NET compiled code. As with all uses of type erasure, you can use explicit boxing, unboxing, and casting to subvert erased types. In this case, a cast exception that isn't valid may result when the object is used. A provider runtime can define its own private representation type to help protect against false representations. You can't define erased types in F# itself. Only provided types may be erased. You must understand the ramifications,

both practical and semantic, of using either erased types for your type provider or a provider that provides erased types. An erased type has no real .NET type. Therefore, you cannot do accurate reflection over the type, and you might subvert erased types if you use runtime casts and other techniques that rely on exact runtime type semantics. Subversion of erased types frequently results in type cast exceptions at runtime.

Choosing Representations for Erased Provided Types

For some uses of erased provided types, no representation is required. For example, the erased provided type might contain only static properties and members and no constructors, and no methods or properties would return an instance of the type. If you can reach instances of an erased provided type, you must consider the following questions:

What is the erasure of a provided type?

- The erasure of a provided type is how the type appears in compiled .NET code.
- The erasure of a provided erased class type is always the first non-erased base type in the inheritance chain of the type.
- The erasure of a provided erased interface type is always `System.Object`.

What are the representations of a provided type?

- The set of possible objects for an erased provided type are called its representations. In the example in this document, the representations of all the erased provided types `Type1..Type100` are always string objects.

All representations of a provided type must be compatible with the erasure of the provided type. (Otherwise, either the F# compiler will give an error for a use of the type provider, or unverifiable .NET code that isn't valid will be generated. A type provider isn't valid if it returns code that gives a representation that isn't valid.)

You can choose a representation for provided objects by using either of the following approaches, both of which are very common:

- If you're simply providing a strongly typed wrapper over an existing .NET type, it often makes sense for your type to erase to that type, use instances of that type as representations, or both. This approach is appropriate when most of the existing methods on that type still make sense when using the strongly typed version.
- If you want to create an API that differs significantly from any existing .NET API, it makes sense to create runtime types that will be the type erasure and representations for the provided types.

The example in this document uses strings as representations of provided objects. Frequently, it may be appropriate to use other objects for representations. For example, you may use a dictionary as a property bag:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))
```

As an alternative, you may define a type in your type provider that will be used at runtime to form the representation, along with one or more runtime operations:

```
type DataObject() =
    let data = Dictionary<string,obj>()
    member x.RuntimeOperation() = data.Count
```

Provided members can then construct instances of this object type:

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))
```

In this case, you may (optionally) use this type as the type erasure by specifying this type as the `baseType` when constructing the `ProvidedTypeDefinition`:

```
ProvidedTypeDefinition(…, baseType = Some typeof<DataObject> )
…
ProvidedConstructor(…, InvokeCode = (fun args -> <@@ new DataObject() @@>), …)
```

Key Lessons

The previous section explained how to create a simple erasing type provider that provides a range of types, properties, and methods. This section also explained the concept of type erasure, including some of the advantages and disadvantages of providing erased types from a type provider, and discussed representations of erased types.

A Type Provider That Uses Static Parameters

The ability to parameterize type providers by static data enables many interesting scenarios, even in cases when the provider doesn't need to access any local or remote data. In this section, you'll learn some basic techniques for putting together such a provider.

Type Checked Regex Provider

Imagine that you want to implement a type provider for regular expressions that wraps the .NET Regex libraries in an interface that provides the following compile-time guarantees:

- Verifying whether a regular expression is valid.
- Providing named properties on matches that are based on any group names in the regular expression.

This section shows you how to use type providers to create a `RegexTyped` type that the regular expression pattern parameterizes to provide these benefits. The compiler will report an error if the supplied pattern isn't valid, and the type provider can extract the groups from the pattern so that you can access them by using named properties on matches. When you design a type provider, you should consider how its exposed API should look to end users and how this design will translate to .NET code. The following example shows how to use such an API to get the components of the area code:

```
type T = RegexTyped< @"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)">
let reg = T()
let result = T.IsMatch("425-555-2345")
let r = reg.Match("425-555-2345").Group_AreaCode.Value //r equals "425"
```

The following example shows how the type provider translates these calls:

```
let reg = new Regex(@"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)")
let result = reg.IsMatch("425-123-2345")
let r = reg.Match("425-123-2345").Groups.["AreaCode"].Value //r equals "425"
```

Note the following points:

- The standard Regex type represents the parameterized `RegexTyped` type.
- The `RegexTyped` constructor results in a call to the Regex constructor, passing in the static type argument for the pattern.

- The results of the `Match` method are represented by the standard Match type.
- Each named group results in a provided property, and accessing the property results in a use of an indexer on a match's `Groups` collection.

The following code is the core of the logic to implement such a provider, and this example omits the addition of all members to the provided type. For information about each added member, see the appropriate section later in this topic. For the full code, download the sample from the F# 3.0 Sample Pack on the CodePlex website.

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

          match parameterValues with
          | [| :? string as pattern|] ->

            // Create an instance of the regular expression.
            //
            // This will fail with System.ArgumentException if the regular expression is not valid.
            // The exception will escape the type provider and be reported in client code.
            let r = System.Text.RegularExpressions.Regex(pattern)

            // Declare the typed regex provided type.
            // The type erasure of this type is 'obj', even though the representation will always be a Regex
            // This, combined with hiding the object methods, makes the IntelliSense experience simpler.
            let ty =
              ProvidedTypeDefinition(
                thisAssembly,
                rootNamespace,
                typeName,
                baseType = Some baseTy)

            ...

            ty
          | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Note the following points:

- The type provider takes two static parameters: the `pattern`, which is mandatory, and the `options`, which are optional (because a default value is provided).

- After the static arguments are supplied, you create an instance of the regular expression. This instance will throw an exception if the Regex is malformed, and this error will be reported to users.
- Within the `DefineStaticParameters` callback, you define the type that will be returned after the arguments are supplied.
- This code sets `HideObjectMethods` to true so that the IntelliSense experience will remain streamlined. This attribute causes the `Equals`, `GetHashCode`, `Finalize`, and `GetType` members to be suppressed from IntelliSense lists for a provided object.
- You use `obj` as the base type of the method, but you'll use a `Regex` object as the runtime representation of this type, as the next example shows.
- The call to the `Regex` constructor throws a ArgumentException when a regular expression isn't valid. The compiler catches this exception and reports an error message to the user at compile time or in the Visual Studio editor. This exception enables regular expressions to be validated without running an application.

The type defined above isn't useful yet because it doesn't contain any meaningful methods or properties. First, add a static `IsMatch` method:

```
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%%args.[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified input string."
ty.AddMember isMatch
```

The previous code defines a method `IsMatch`, which takes a string as input and returns a `bool`. The only tricky part is the use of the `args` argument within the `InvokeCode` definition. In this example, `args` is a list of quotations that represents the arguments to this method. If the method is an instance method, the first argument represents the `this` argument. However, for a static method, the arguments are all just the explicit arguments to the method. Note that the type of the quoted value should match the specified return type (in this case, `bool`). Also note that this code uses the `AddXmlDoc` method to make sure that the provided method also has useful documentation, which you can supply through IntelliSense.

Next, add an instance Match method. However, this method should return a value of a provided `Match` type so that the groups can be accessed in a strongly typed fashion. Thus, you first declare the `Match` type. Because this type depends on the pattern that was supplied as a static argument, this type must be nested within the parameterized type definition:

```
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

ty.AddMember matchTy
```

You then add one property to the Match type for each group. At runtime, a match is represented as a Match value, so the quotation that defines the property must use the Groups indexed property to get the relevant group.

```
for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
    let prop =
      ProvidedProperty(
        propertyName = group,
        propertyType = typeof<Group>,
        getterCode = fun args -> <@@ ((%%args.[0]:obj) :?> Match).Groups.[group] @@>)
        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
    matchTy.AddMember prop
```

Again, note that you're adding XML documentation to the provided property. Also note that a property can be read if a `GetterCode` function is provided, and the property can be written if a `SetterCode` function is provided, so the resulting property is read only.

Now you can create an instance method that returns a value of this `Match` type:

```
let matchMethod =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%%args.[0]:obj) :?> Regex).Match(%%args.[1]) :> obj @@>)

matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this regular expression"

ty.AddMember matchMeth
```

Because you are creating an instance method, `args.[0]` represents the `RegexTyped` instance on which the method is being called, and `args.[1]` is the input argument.

Finally, provide a constructor so that instances of the provided type can be created.

```
let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern, options) :> obj @@>)

ctor.AddXmlDoc("Initializes a regular expression instance.")

ty.AddMember ctor
```

The constructor merely erases to the creation of a standard .NET Regex instance, which is again boxed to an object because `obj` is the erasure of the provided type. With that change, the sample API usage that specified earlier in the topic works as expected. The following code is complete and final:

```
namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types.
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
```

```
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.

                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.

                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,
                        typeName,
                        baseType = Some baseTy)

                ty.AddXmlDoc "A strongly typed interface to the regular expression '%s'"

                // Provide strongly typed version of Regex.IsMatch static method.
                let isMatch =
                    ProvidedMethod(
                        methodName = "IsMatch",
                        parameters = [ProvidedParameter("input", typeof<string>)],
                        returnType = typeof<bool>,
                        isStatic = true,
                        invokeCode = fun args -> <@@ Regex.IsMatch(%%args.[0], pattern) @@>)

                isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified
input string"

                ty.AddMember isMatch

                // Provided type for matches
                // Again, erase to obj even though the representation will always be a Match
                let matchTy =
                    ProvidedTypeDefinition(
                        "MatchType",
                        baseType = Some baseTy,
                        hideObjectMethods = true)

                // Nest the match type within parameterized Regex type.
                ty.AddMember matchTy

                // Add group properties to match type
                for group in r.GetGroupNames() do
                    // Ignore the group named 0, which represents all input.
                    if group <> "0" then
                        let prop =
                          ProvidedProperty(
                            propertyName = group,
                            propertyType = typeof<Group>,
                            getterCode = fun args -> <@@ ((%%args.[0]:obj) :?> Match).Groups.[group] @@>)
                        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
                        matchTy.AddMember(prop)

                // Provide strongly typed version of Regex.Match instance method.
                let matchMeth =
                  ProvidedMethod(
                    methodName = "Match",
                    parameters = [ProvidedParameter("input", typeof<string>)],
```

```
                    returnType = matchTy,
                    invokeCode = fun args -> <@@ ((%%args.[0]:obj) :?> Regex).Match(%%args.[1]) :> obj @@>)
                matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this
regular expression"

                ty.AddMember matchMeth

                // Declare a constructor.
                let ctor =
                  ProvidedConstructor(
                    parameters = [],
                    invokeCode = fun args -> <@@ Regex(pattern) :> obj @@>)

                // Add documentation to the constructor.
                ctor.AddXmlDoc "Initializes a regular expression instance"

                ty.AddMember ctor

                ty
            | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()
```

Key Lessons

This section explained how to create a type provider that operates on its static parameters. The provider checks the static parameter and provides operations based on its value.

A Type Provider That Is Backed By Local Data

Frequently you might want type providers to present APIs based on not only static parameters but also information from local or remote systems. This section discusses type providers that are based on local data, such as local data files.

Simple CSV File Provider

As a simple example, consider a type provider for accessing scientific data in Comma Separated Value (CSV) format. This section assumes that the CSV files contain a header row followed by floating point data, as the following table illustrates:

DISTANCE (METER)	TIME (SECOND)
50.0	3.7
100.0	5.2
150.0	6.4

This section shows how to provide a type that you can use to get rows with a `Distance` property of type `float<meter>` and a `Time` property of type `float<second>`. For simplicity, the following assumptions are made:

- Header names are either unit-less or have the form "Name (unit)" and don't contain commas.
- Units are all System International (SI) units as the Microsoft.FSharp.Data.UnitSystems.SI.UnitNames Module (F#) module defines.
- Units are all simple (for example, meter) rather than compound (for example, meter/second).
- All columns contain floating point data.

A more complete provider would loosen these restrictions.

Again the first step is to consider how the API should look. Given an `info.csv` file with the contents from the previous table (in comma-separated format), users of the provider should be able to write code that resembles the following example:

```
let info = new MiniCsv<"info.csv">()
for row in info.Data do
let time = row.Time
printfn "%f" (float time)
```

In this case, the compiler should convert these calls into something like the following example:

```
let info = new CsvFile("info.csv")
for row in info.Data do
let (time:float) = row.[1]
printfn "%f" (float time)
```

The optimal translation will require the type provider to define a real `CsvFile` type in the type provider's assembly. Type providers often rely on a few helper types and methods to wrap important logic. Because measures are erased at runtime, you can use a `float[]` as the erased type for a row. The compiler will treat different columns as having different measure types. For example, the first column in our example has type `float<meter>`, and the second has `float<second>`. However, the erased representation can remain quite simple.

The following code shows the core of the implementation.

```
// Simple type wrapping CSV data
type CsvFile(filename) =
    // Cache the sequence of all data lines (all lines but the first)
    let data =
        seq { for line in File.ReadAllLines(filename) |> Seq.skip 1 do
                 yield line.Split(',') |> Array.map float }
        |> Seq.cache
    member __.Data = data

[<TypeProvider>]
type public MiniCsvProvider(cfg:TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces(cfg)

    // Get the assembly and namespace used to house the provided types.
    let asm = System.Reflection.Assembly.GetExecutingAssembly()
    let ns = "Samples.FSharp.MiniCsvProvider"

    // Create the main provided type.
    let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

    // Parameterize the type by the file to use as a template.
    let filename = ProvidedStaticParameter("filename", typeof<string>)
    do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

        // Resolve the filename relative to the resolution folder.
        let resolvedFilename = Path.Combine(cfg.ResolutionFolder, filename)

        // Get the first line from the file.
        let headerLine = File.ReadLines(resolvedFilename) |> Seq.head

        // Define a provided type for each row, erasing to a float[].
        let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))

        // Extract header names from the file, splitting on commas.
        // use Regex matching to get the position in the row at which the field occurs
        let headers = Regex.Matches(headerLine, "[^,]+")
```

```
        // Add one property per CSV field.
        for i in 0 .. headers.Count - 1 do
            let headerText = headers.[i].Value

            // Try to decompose this header into a name and unit.
            let fieldName, fieldTy =
                let m = Regex.Match(headerText, @"(?<field>.+) \((?<unit>.+)\)")
                if m.Success then

                    let unitName = m.Groups.["unit"].Value
                    let units = ProvidedMeasureBuilder.Default.SI unitName
                    m.Groups.["field"].Value, ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>,
[units])

                else
                    // no units, just treat it as a normal float
                    headerText, typeof<float>

            let prop =
                ProvidedProperty(fieldName, fieldTy,
                    getterCode = fun [row] -> <@@ (%%row:float[]).[i] @@>)

            // Add metadata that defines the property's location in the referenced file.
            prop.AddDefinitionLocation(1, headers.[i].Index + 1, filename)
            rowTy.AddMember(prop)

        // Define the provided type, erasing to CsvFile.
        let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

        // Add a parameterless constructor that loads the file that was used to define the schema.
        let ctor0 =
            ProvidedConstructor([],
                invokeCode = fun [] -> <@@ CsvFile(resolvedFilename) @@>)
        ty.AddMember ctor0

        // Add a constructor that takes the file name to load.
        let ctor1 = ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],
            invokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>)
        ty.AddMember ctor1

        // Add a more strongly typed Data property, which uses the existing property at runtime.
        let prop =
            ProvidedProperty("Data", typedefof<seq<_>>.MakeGenericType(rowTy),
                getterCode = fun [csvFile] -> <@@ (%%csvFile:CsvFile).Data @@>)
        ty.AddMember prop

        // Add the row type as a nested type.
        ty.AddMember rowTy
        ty)

    // Add the type to the namespace.
    do this.AddNamespace(ns, [csvTy])
```

Note the following points about the implementation:

- Overloaded constructors allow either the original file or one that has an identical schema to be read. This pattern is common when you write a type provider for local or remote data sources, and this pattern allows a local file to be used as the template for remote data.
- You can use the TypeProviderConfig value that's passed in to the type provider constructor to resolve relative file names.
- You can use the `AddDefinitionLocation` method to define the location of the provided properties. Therefore, if you use `Go To Definition` on a provided property, the CSV file will open in Visual Studio.

- You can use the `ProvidedMeasureBuilder` type to look up the SI units and to generate the relevant `float<_>` types.

Key Lessons

This section explained how to create a type provider for a local data source with a simple schema that's contained in the data source itself.

Going Further

The following sections include suggestions for further study.

A Look at the Compiled Code for Erased Types

To give you some idea of how the use of the type provider corresponds to the code that's emitted, look at the following function by using the `HelloWorldTypeProvider` that's used earlier in this topic.

```
let function1 () =
    let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")
    obj1.InstanceProperty
```

Here's an image of the resulting code decompiled by using ildasm.exe:

```
.class public abstract auto ansi sealed Module1
extends [mscorlib]System.Object
{
.custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAtt
ribute::.ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags)
= ( 01 00 07 00 00 00 00 00 )
.method public static int32  function1() cil managed
{
// Code size       24 (0x18)
.maxstack  3
.locals init ([0] object obj1)
IL_0000:  nop
IL_0001:  ldstr      "some data"
IL_0006:  unbox.any  [mscorlib]System.Object
IL_000b:  stloc.0
IL_000c:  ldloc.0
IL_000d:  call       !!0 [FSharp.Core_2]Microsoft.FSharp.Core.LanguagePrimit
ives/IntrinsicFunctions::UnboxGeneric<string>(object)
IL_0012:  callvirt   instance int32 [mscorlib_3]System.String::get_Length()
IL_0017:  ret
} // end of method Module1::function1

} // end of class Module1
```

As the example shows, all mentions of the type `Type1` and the `InstanceProperty` property have been erased, leaving only operations on the runtime types involved.

Design and Naming Conventions for Type Providers

Observe the following conventions when authoring type providers.

Providers for Connectivity Protocols In general, names of most provider DLLs for data and service connectivity protocols, such as OData or SQL connections, should end in `TypeProvider` or `TypeProviders`. For example, use a DLL name that resembles the following string:

```
Fabrikam.Management.BasicTypeProviders.dll
```

Ensure that your provided types are members of the corresponding namespace, and indicate the connectivity

protocol that you implemented:

```
Fabrikam.Management.BasicTypeProviders.WmiConnection<…>
Fabrikam.Management.BasicTypeProviders.DataProtocolConnection<…>
```

Utility Providers for General Coding. For a utility type provider such as that for regular expressions, the type provider may be part of a base library, as the following example shows:

```
#r "Fabrikam.Core.Text.Utilities.dll"
```

In this case, the provided type would appear at an appropriate point according to normal .NET design conventions:

```
open Fabrikam.Core.Text.RegexTyped

let regex = new RegexTyped<"a+b+a+b+">()
```

Singleton Data Sources. Some type providers connect to a single dedicated data source and provide only data. In this case, you should drop the `TypeProvider` suffix and use normal conventions for .NET naming:

```
#r "Fabrikam.Data.Freebase.dll"

let data = Fabrikam.Data.Freebase.Astronomy.Asteroids
```

For more information, see the `GetConnection` design convention that's described later in this topic.

Design Patterns for Type Providers

The following sections describe design patterns you can use when authoring type providers.

The GetConnection Design Pattern

Most type providers should be written to use the `GetConnection` pattern that's used by the type providers in FSharp.Data.TypeProviders.dll, as the following example shows:

```
#r "Fabrikam.Data.WebDataStore.dll"

type Service = Fabrikam.Data.WebDataStore<…static connection parameters…>

let connection = Service.GetConnection(…dynamic connection parameters…)

let data = connection.Astronomy.Asteroids
```

Type Providers Backed By Remote Data and Services

Before you create a type provider that's backed by remote data and services, you must consider a range of issues that are inherent in connected programming. These issues include the following considerations:

- schema mapping
- liveness and invalidation in the presence of schema change
- schema caching
- asynchronous implementations of data access operations
- supporting queries, including LINQ queries
- credentials and authentication

This topic doesn't explore these issues further.

Additional Authoring Techniques

When you write your own type providers, you might want to use the following additional techniques.

Creating Types and Members On-Demand

The ProvidedType API has delayed versions of AddMember.

```
type ProvidedType =
    member AddMemberDelayed  : (unit -> MemberInfo)      -> unit
    member AddMembersDelayed : (unit -> MemberInfo list) -> unit
```

These versions are used to create on-demand spaces of types.

Providing Array types and Generic Type Instantiations

You make provided members (whose signatures include array types, byref types, and instantiations of generic types) by using the normal `MakeArrayType`, `MakePointerType`, and `MakeGenericType` on any instance of Type, including `ProvidedTypeDefinitions`.

> **NOTE**
>
> In some cases you may have to use the helper in `ProvidedTypeBuilder.MakeGenericType`. See the Type Provider SDK documentation for more details.

Providing Unit of Measure Annotations

The ProvidedTypes API provides helpers for providing measure annotations. For example, to provide the type `float<kg>`, use the following code:

```
let measures = ProvidedMeasureBuilder.Default
let kg = measures.SI "kilogram"
let m = measures.SI "meter"
let float_kg = measures.AnnotateType(typeof<float>,[kg])
```

To provide the type `Nullable<decimal<kg/m^2>>`, use the following code:

```
let kgpm2 = measures.Ratio(kg, measures.Square m)
let dkgpm2 = measures.AnnotateType(typeof<decimal>,[kgpm2])
let nullableDecimal_kgpm2 = typedefof<System.Nullable<_>>.MakeGenericType [|dkgpm2 |]
```

Accessing Project-Local or Script-Local Resources

Each instance of a type provider can be given a `TypeProviderConfig` value during construction. This value contains the "resolution folder" for the provider (that is, the project folder for the compilation or the directory that contains a script), the list of referenced assemblies, and other information.

Invalidation

Providers can raise invalidation signals to notify the F# language service that the schema assumptions may have changed. When invalidation occurs, a typecheck is redone if the provider is being hosted in Visual Studio. This signal will be ignored when the provider is hosted in F# Interactive or by the F# Compiler (fsc.exe).

Caching Schema Information

Providers must often cache access to schema information. The cached data should be stored by using a file name that's given as a static parameter or as user data. An example of schema caching is the `LocalSchemaFile` parameter in the type providers in the `FSharp.Data.TypeProviders` assembly. In the implementation of these providers, this

static parameter directs the type provider to use the schema information in the specified local file instead of accessing the data source over the network. To use cached schema information, you must also set the static parameter `ForceUpdate` to `false`. You could use a similar technique to enable online and offline data access.

Backing Assembly

When you compile a `.dll` or `.exe` file, the backing .dll file for generated types is statically linked into the resulting assembly. This link is created by copying the Intermediate Language (IL) type definitions and any managed resources from the backing assembly into the final assembly. When you use F# Interactive, the backing .dll file isn't copied and is instead loaded directly into the F# Interactive process.

Exceptions and Diagnostics from Type Providers

All uses of all members from provided types may throw exceptions. In all cases, if a type provider throws an exception, the host compiler attributes the error to a specific type provider.

- Type provider exceptions should never result in internal compiler errors.
- Type providers can't report warnings.
- When a type provider is hosted in the F# compiler, an F# development environment, or F# Interactive, all exceptions from that provider are caught. The Message property is always the error text, and no stack trace appears. If you're going to throw an exception, you can throw the following examples: `System.NotSupportedException`, `System.IO.IOException`, `System.Exception`.

Providing Generated Types

So far, this document has explained how to provide erased types. You can also use the type provider mechanism in F# to provide generated types, which are added as real .NET type definitions into the users' program. You must refer to generated provided types by using a type definition.

```
open Microsoft.FSharp.TypeProviders

type Service = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

The ProvidedTypes-0.2 helper code that is part of the F# 3.0 release has only limited support for providing generated types. The following statements must be true for a generated type definition:

- `isErased` must be set to `false`.
- The generated type must be added to a newly constructed `ProvidedAssembly()`, which represents a container for generated code fragments.
- The provider must have an assembly that has an actual backing .NET .dll file with a matching .dll file on disk.

Rules and Limitations

When you write type providers, keep the following rules and limitations in mind.

Provided types must be reachable

All provided types should be reachable from the non-nested types. The non-nested types are given in the call to the `TypeProviderForNamespaces` constructor or a call to `AddNamespace`. For example, if the provider provides a type `StaticClass.P : T`, you must ensure that T is either a non-nested type or nested under one.

For example, some providers have a static class such as `DataTypes` that contain these `T1, T2, T3, ...` types. Otherwise, the error says that a reference to type T in assembly A was found, but the type couldn't be found in that assembly. If this error appears, verify that all your subtypes can be reached from the provider types. Note: These `T1, T2, T3...` types are referred to as the *on-the-fly* types. Remember to put them in an accessible namespace or

a parent type.

Limitations of the Type Provider Mechanism

The type provider mechanism in F# has the following limitations:

- The underlying infrastructure for type providers in F# doesn't support provided generic types or provided generic methods.
- The mechanism doesn't support nested types with static parameters.

Development Tips

You might find the following tips helpful during the development process:

Run two instances of Visual Studio

You can develop the type provider in one instance and test the provider in the other because the test IDE will take a lock on the .dll file that prevents the type provider from being rebuilt. Thus, you must close the second instance of Visual Studio while the provider is built in the first instance, and then you must reopen the second instance after the provider is built.

Debug type providers by using invocations of fsc.exe

You can invoke type providers by using the following tools:

- fsc.exe (The F# command line compiler)
- fsi.exe (The F# Interactive compiler)
- devenv.exe (Visual Studio)

You can often debug type providers most easily by using fsc.exe on a test script file (for example, script.fsx). You can launch a debugger from a command prompt.

```
devenv /debugexe fsc.exe script.fsx
```

You can use print-to-stdout logging.

See also

- Type Providers
- The Type Provider SDK

Type Provider Security

5/15/2019 • 2 minutes to read • Edit Online

Type providers are assemblies (DLLs) referenced by your F# project or script that contain code to connect to external data sources and surface this type information to the F# type environment. Typically, code in referenced assemblies is only run when you compile and then execute the code (or in the case of a script, send the code to F# Interactive). However, a type provider assembly will run inside Visual Studio when the code is merely browsed in the editor. This happens because type providers need to run to add extra information to the editor, such as Quick Info tooltips, IntelliSense completions, and so on. As a result, there are extra security considerations for type provider assemblies, since they run automatically inside the Visual Studio process.

Security Warning Dialog

When using a particular type provider assembly for the first time, Visual Studio displays a security dialog that warns you that the type provider is about to run. Before Visual Studio loads the type provider, it gives you the opportunity to decide if you trust this particular provider. If you trust the source of the type provider, then select "I trust this type provider." If you do not trust the source of the type provider, then select "I do not trust this type provider." Trusting the provider enables it to run inside Visual Studio and provide IntelliSense and build features. But if the type provider itself is malicious, running its code could compromise your machine.

If your project contains code that references type providers that you chose in the dialog not to trust, then at compile time, the compiler will report an error that indicates that the type provider is untrusted. Any types that are dependent on the untrusted type provider are indicated by red squiggles. It is safe to browse the code in the editor.

If you decide to change the trust setting directly in Visual Studio, perform the following steps.

To change the trust settings for type providers

1. On the `Tools` menu, select `Options`, and expand the `F# Tools` node.
2. Select `Type Providers`, and in the list of type providers, select the check box for type providers you trust, and clear the check box for those you don't trust.

See also

- Type Providers

Troubleshooting Type Providers

5/15/2019 • 2 minutes to read • Edit Online

This topic describes and provides potential solutions for the problems that you are most likely to encounter when you use type providers.

Possible Problems with Type Providers

If you encounter a problem when you work with type providers, you can review the following table for the most common solutions.

PROBLEM	SUGGESTED ACTIONS
Schema Changes. Type providers work best when the data source schema is stable. If you add a data table or column or make another change to that schema, the type provider doesn't automatically recognize these changes.	Clean or rebuild the project. To clean the project, choose **Build**, **Clean** *ProjectName* on the menu bar. To rebuild the project, choose **Build**, **Rebuild** *ProjectName* on the menu bar. These actions reset all type provider state and force the provider to reconnect to the data source and obtain updated schema information.
Connection Failure. The URL or connection string is incorrect, the network is down, or the data source or service is unavailable.	For a web service or OData service, you can try the URL in Internet Explorer to verify whether the URL is correct and the service is available. For a database connection string, you can use the data connection tools in **Server Explorer** to verify whether the connection string is valid and the database is available. After you restore your connection, you should then clean or rebuild the project so that the type provider will reconnect to the network.
Not Valid Credentials. You must have valid permissions for the data source or web service.	For a SQL connection, the username and the password that are specified in the connection string or configuration file must be valid for the database. If you are using Windows Authentication, you must have access to the database. The database administrator can identify what permissions you need for access to each database and each element within a database. For a web service or a data service, you must have appropriate credentials. Most type providers provide a DataContext object, which contains a Credentials property that you can set with the appropriate username and access key.
Not Valid Path. A path to a file was not valid.	Verify whether the path is correct and the file exists. In addition, you must either quote any backlashes in the path appropriately or use a verbatim string or triple-quoted string.

See also

- Type Providers

Interactive Programming with F#

5/15/2019 • 4 minutes to read • Edit Online

NOTE

This article currently describes the experience for Windows only. It will be rewritten.

NOTE

The API reference link will take you to MSDN. The docs.microsoft.com API reference is not complete.

F# Interactive (fsi.exe) is used to run F# code interactively at the console, or to execute F# scripts. In other words, F# interactive executes a REPL (Read, Evaluate, Print Loop) for the F# language.

To run F# Interactive from the console, run fsi.exe. You will find fsi.exe in:

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\<sku>\Common7\IDE\CommonExtensions\Microsoft\FSharp
```

where `sku` is either `Community`, `Professional`, or `Enterprise`.

For information about command line options available, see F# Interactive Options.

To run F# Interactive through Visual Studio, you can click the appropriate toolbar button labeled **F# Interactive**, or use the keys **Ctrl+Alt+F**. Doing this will open the interactive window, a tool window running an F# Interactive session. You can also select some code that you want to run in the interactive window and hit the key combination **ALT+ENTER**. F# Interactive starts in a tool window labeled **F# Interactive**. When you use this key combination, make sure that the editor window has the focus.

Whether you are using the console or Visual Studio, a command prompt appears and the interpreter awaits your input. You can enter code just as you would in a code file. To compile and execute the code, enter two semicolons (**;;**) to terminate a line or several lines of input.

F# Interactive attempts to compile the code and, if successful, it executes the code and prints the signature of the types and values that it compiled. If errors occur, the interpreter prints the error messages.

Code entered in the same session has access to any constructs entered previously, so you can build up programs. An extensive buffer in the tool window allows you to copy the code into a file if needed.

When run in Visual Studio, F# Interactive runs independently of your project, so, for example, you cannot use constructs defined in your project in F# Interactive unless you copy the code for the function into the interactive window.

If you have a project open that references some libraries, you can reference these in F# Interactive through **Solution Explorer**. To reference a library in F# Interactive, expand the **References** node, open the shortcut menu for the library, and choose **Send to F# Interactive**.

You can control the F# Interactive command line arguments (options) by adjusting the settings. On the **Tools** menu, select **Options...**, and then expand **F# Tools**. The two settings that you can change are the F# Interactive options and the **64-bit F# Interactive** setting, which is relevant only if you are running F# Interactive on a 64-bit machine. This setting determines whether you want to run the dedicated 64-bit version of fsi.exe or fsianycpu.exe, which uses the machine architecture to determine whether to run as a 32-bit or 64-bit process.

Scripting with F#

Scripts use the file extension **.fsx** or **.fsscript**. Instead of compiling source code and then later running the compiled assembly, you can just run **fsi.exe** and specify the filename of the script of F# source code, and F# interactive reads the code and executes it in real time.

Differences Between the Interactive, Scripting and Compiled Environments

When you are compiling code in F# Interactive, whether you are running interactively or running a script, the symbol **INTERACTIVE** is defined. When you are compiling code in the compiler, the symbol **COMPILED** is defined. Thus, if code needs to be different in compiled and interactive modes, you can use preprocessor directives for conditional compilation to determine which to use.

Some directives are available when you are executing scripts in F# Interactive that are not available when you are executing the compiler. The following table summarizes directives that are available when you are using F# Interactive.

DIRECTIVE	DESCRIPTION
#help	Displays information about available directives.
#I	Specifies an assembly search path in quotation marks.
#load	Reads a source file, compiles it, and runs it.
#quit	Terminates an F# Interactive session.
#r	References an assembly.
#time ["on"|"off"]	By itself, **#time** toggles whether to display performance information. When it is enabled, F# Interactive measures real time, CPU time, and garbage collection information for each section of code that is interpreted and executed.

When you specify files or paths in F# Interactive, a string literal is expected. Therefore, files and paths must be in quotation marks, and the usual escape characters apply. Also, you can use the @ character to cause F# Interactive to interpret a string that contains a path as a verbatim string. This causes F# Interactive to ignore any escape characters.

One of the differences between compiled and interactive mode is the way you access command line arguments. In compiled mode, use **System.Environment.GetCommandLineArgs**. In scripts, use **fsi.CommandLineArgs**.

The following code illustrates how to create a function that reads the command line arguments in a script and also demonstrates how to reference another assembly from a script. The first code file, **MyAssembly.fs**, is the code for the assembly being referenced. Compile this file with the command line: **fsc -a MyAssembly.fs** and then execute the second file as a script with the command line: **fsi --exec file1.fsx** test

```
// MyAssembly.fs
module MyAssembly
let myFunction x y = x + 2 * y
```

```
// file1.fsx
#r "MyAssembly.dll"

printfn "Command line arguments: "

for arg in fsi.CommandLineArgs do
    printfn "%s" arg

printfn "%A" (MyAssembly.myFunction 10 40)
```

The output is as follows:

```
Command line arguments:
file1.fsx
test
90
```

Related Topics

TITLE	DESCRIPTION
F# Interactive Options	Describes command-line syntax and options for the F# Interactive, fsi.exe.
F# Interactive Library Reference	Describes library functionality available when executing code in F# interactive.

F# Language Reference

8/14/2019 • 8 minutes to read • Edit Online

This section is a reference to the F# language, a multi-paradigm programming language targeting .NET. The F# language supports functional, object-oriented and imperative programming models.

F# Tokens

The following table shows reference topics that provide tables of keywords, symbols and literals used as tokens in F#.

TITLE	DESCRIPTION
Keyword Reference	Contains links to information about all F# language keywords.
Symbol and Operator Reference	Contains a table of symbols and operators that are used in the F# language.
Literals	Describes the syntax for literal values in F# and how to specify type information for F# literals.

F# Language Concepts

The following table shows reference topics available that describe language concepts.

TITLE	DESCRIPTION
Functions	Functions are the fundamental unit of program execution in any programming language. As in other languages, an F# function has a name, can have parameters and take arguments, and has a body. F# also supports functional programming constructs such as treating functions as values, using unnamed functions in expressions, composition of functions to form new functions, curried functions, and the implicit definition of functions by way of the partial application of function arguments.
F# Types	Describes the types that are used in F# and how F# types are named and described.
Type Inference	Describes how the F# compiler infers the types of values, variables, parameters and return values.
Automatic Generalization	Describes generic constructs in F#.
Inheritance	Describes inheritance, which is used to model the "is-a" relationship, or subtyping, in object-oriented programming.
Members	Describes members of F# object types.

TITLE	DESCRIPTION
Parameters and Arguments	Describes language support for defining parameters and passing arguments to functions, methods, and properties. It includes information about how to pass by reference.
Operator Overloading	Describes how to overload arithmetic operators in a class or record type, and at the global level.
Casting and Conversions	Describes support for type conversions in F#.
Access Control	Describes access control in F#. Access control means declaring what clients are able to use certain program elements, such as types, methods, functions and so on.
Pattern Matching	Describes patterns, which are rules for transforming input data that are used throughout the F# language to extract compare data with a pattern, decompose data into constituent parts, or extract information from data in various ways.
Active Patterns	Describes active patterns. Active patterns enable you to define named partitions that subdivide input data. You can use active patterns to decompose data in a customized manner for each partition.
Assertions	Describes the `assert` expression, which is a debugging feature that you can use to test an expression. Upon failure in Debug mode, an assertion generates a system error dialog box.
Exception Handling	Contains information about exception handling support in the F# language.
attributes	Describes attributes, which enable metadata to be applied to a programming construct.
Resource Management: The `use` Keyword	Describes the keywords `use` and `using`, which can control the initialization and release of resources
namespaces	Describes namespace support in F#. A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.
Modules	Describes modules. An F# module is a grouping of F# code, such as values, types, and function values, in an F# program. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.
Import Declarations: The `open` Keyword	Describes how `open` works. An import declaration specifies a module or namespace whose elements you can reference without using a fully qualified name.

TITLE	DESCRIPTION
Signatures	Describes signatures and signature files. A signature file contains information about the public signatures of a set of F# program elements, such as types, namespaces, and modules. It can be used to specify the accessibility of these program elements.
XML Documentation	Describes support for generating documentation files for XML doc comments, also known as triple slash comments. You can produce documentation from code comments in F# just as in other .NET languages.
Verbose Syntax	Describes the syntax for F# constructs when lightweight syntax is not enabled. Verbose syntax is indicated by the `#light "off"` directive at the top of the code file.

F# Types

The following table shows reference topics available that describe types supported by the F# language.

TITLE	DESCRIPTION
values	Describes values, which are immutable quantities that have a specific type; values can be integral or floating point numbers, characters or text, lists, sequences, arrays, tuples, discriminated unions, records, class types, or function values.
Basic Types	Describes the fundamental basic types that are used in the F# language. It also provides the corresponding .NET types and the minimum and maximum values for each type.
Unit Type	Describes the `unit` type, which is a type that indicates the absence of a specific value; the `unit` type has only a single value, which acts as a placeholder when no other value exists or is needed.
Strings	Describes strings in F#. The `string` type represents immutable text, as a sequence of Unicode characters. `string` is an alias for `System.String` in the .NET Framework.
Tuples	Describes tuples, which are groupings of unnamed but ordered values of possibly different types.
F# Collection Types	An overview of the F# functional collection types, including types for arrays, lists, sequences (seq), maps, and sets.
Lists	Describes lists. A list in F# is an ordered, immutable series of elements all of the same type.

TITLE	DESCRIPTION
Options	Describes the option type. An option in F# is used when a value may or may not exist. An option has an underlying type and may either hold a value of that type or it may not have a value.
Sequences	Describes sequences. A sequence is a logical series of elements all of one type. Individual sequence elements are only computed if required, so the representation may be smaller than a literal element count indicates.
Arrays	Describes arrays. Arrays are fixed-size, zero-based, mutable sequences of consecutive data elements, all of the same type.
Records	Describes records. Records represent simple aggregates of named values, optionally with members.
Discriminated Unions	Describes discriminated unions, which provides support for values which may be one of a variety of named cases, each with possibly different values and types.
Enumerations	Describes enumerations are types that have a defined set of named values. You can use them in place of literals to make code more readable and maintainable.
Reference Cells	Describes reference cells, which are storage locations that enable you to create mutable variables with reference semantics.
Type Abbreviations	Describes type abbreviations, which are alternate names for types.
Classes	Describes classes, which are types that represent objects that can have properties, methods, and events.
Structures	Describes structures, which are compact object types that can be more efficient than a class for types that have a small amount of data and simple behavior.
Interfaces	Describes interfaces, which specify sets of related members that other classes implement.
Abstract Classes	Describes abstract classes, which are classes that leave some or all members unimplemented, so that implementations can be provided by derived classes.
Type Extensions	Describes type extensions, which let you add new members to a previously defined object type.
Flexible Types	Describes flexible types. A flexible type annotation is an indication that a parameter, variable or value has a type that is compatible with type specified, where compatibility is determined by position in an object-oriented hierarchy of classes or interfaces.

TITLE	DESCRIPTION
Delegates	Describes delegates, which represent a function call as an object.
Units of Measure	Describes units of measure. Floating point values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on.
Type Providers	Describes type provides and provides links to walkthroughs on using the built-in type providers to access databases and web services.

F# Expressions

The following table lists topics that describe F# expressions.

TITLE	DESCRIPTION
Conditional Expressions: `if...then...else`	Describes the `if...then...else` expression, which runs different branches of code and also evaluates to a different value depending on the Boolean expression given.
Match Expressions	Describes the `match` expression, which provides branching control that is based on the comparison of an expression with a set of patterns.
Loops: `for...to` Expression	Describes the `for...to` expression, which is used to iterate in a loop over a range of values of a loop variable.
Loops: `for...in` Expression	Describes the `for...in` expression, a looping construct that is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.
Loops: `while...do` Expression	Describes the `while...do` expression, which is used to perform iterative execution (looping) while a specified test condition is true.
Object Expressions	Describes object expressions, which are expressions that create new instances of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.
Lazy Expressions	Describes lazy expressions, which are computations that are not evaluated immediately, but are instead evaluated when the result is actually needed.

TITLE	DESCRIPTION
Computation Expressions	Describes computation expressions in F#, which provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. They can be used to provide a convenient syntax for *monads*, a functional programming feature that can be used to manage data, control and side effects in functional programs. One type of computation expression, the asynchronous workflow, provides support for asynchronous and parallel computations. For more information, see Asynchronous Workflows.
Asynchronous Workflows	Describes asynchronous workflows, a language feature that lets you write asynchronous code in a way that is very close to the way you would naturally write synchronous code.
Code Quotations	Describes code quotations, a language feature that enables you to generate and work with F# code expressions programmatically.
Query Expressions	Describes query expressions, a language feature that implements LINQ for F# and enables you to write queries against a data source or enumerable collection.

Compiler-supported Constructs

The following table lists topics that describe special compiler-supported constructs.

TOPIC	DESCRIPTION
Compiler Options	Describes the command-line options for the F# compiler.
Compiler Directives	Describes processor directives and compiler directives.
Source Line, File, and Path Identifiers	Describes the identifiers `__LINE__`, `__SOURCE_DIRECTORY__` and `__SOURCE_FILE__`, which are built-in values that enable you to access the source line number, directory and file name in your code.

See also

- Visual F#

Keyword Reference

8/2/2019 • 7 minutes to read • Edit Online

This topic contains links to information about all F# language keywords.

F# Keyword Table

The following table shows all F# keywords in alphabetical order, together with brief descriptions and links to relevant topics that contain more information.

KEYWORD	LINK	DESCRIPTION
`abstract`	Members Abstract Classes	Indicates a method that either has no implementation in the type in which it is declared or that is virtual and has a default implementation.
`and`	`let` Bindings Records Members Constraints	Used in mutually recursive bindings and records, in property declarations, and with multiple constraints on generic parameters.
`as`	Classes Pattern Matching	Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.
`assert`	Assertions	Used to verify code during debugging.
`base`	Classes Inheritance	Used as the name of the base class object.
`begin`	Verbose Syntax	In verbose syntax, indicates the start of a code block.
`class`	Classes	In verbose syntax, indicates the start of a class definition.
`default`	Members	Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.
`delegate`	Delegates	Used to declare a delegate.

KEYWORD	LINK	DESCRIPTION
`do`	do Bindings Loops: `for...to` Expression Loops: `for...in` Expression Loops: `while...do` Expression	Used in looping constructs or to execute imperative code.
`done`	Verbose Syntax	In verbose syntax, indicates the end of a block of code in a looping expression.
`downcast`	Casting and Conversions	Used to convert to a type that is lower in the inheritance chain.
`downto`	Loops: `for...to` Expression	In a `for` expression, used when counting in reverse.
`elif`	Conditional Expressions: `if...then...else`	Used in conditional branching. A short form of `else if`.
`else`	Conditional Expressions: `if...then...else`	Used in conditional branching.
`end`	Structures Discriminated Unions Records Type Extensions Verbose Syntax	In type definitions and type extensions, indicates the end of a section of member definitions. In verbose syntax, used to specify the end of a code block that starts with the `begin` keyword.
`exception`	Exception Handling Exception Types	Used to declare an exception type.
`extern`	External Functions	Indicates that a declared program element is defined in another binary or assembly.
`false`	Primitive Types	Used as a Boolean literal.
`finally`	Exceptions: The `try...finally` Expression	Used together with `try` to introduce a block of code that executes regardless of whether an exception occurs.
`fixed`	Fixed	Used to "pin" a pointer on the stack to prevent it from being garbage collected.
`for`	Loops: `for...to` Expression Loops: for...in Expression	Used in looping constructs.

KEYWORD	LINK	DESCRIPTION
`fun`	Lambda Expressions: The `fun` Keyword	Used in lambda expressions, also known as anonymous functions.
`function`	Match Expressions Lambda Expressions: The fun Keyword	Used as a shorter alternative to the `fun` keyword and a `match` expression in a lambda expression that has pattern matching on a single argument.
`global`	Namespaces	Used to reference the top-level .NET namespace.
`if`	Conditional Expressions: `if...then...else`	Used in conditional branching constructs.
`in`	Loops: for...in Expression Verbose Syntax	Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.
`inherit`	Inheritance	Used to specify a base class or base interface.
`inline`	Functions Inline Functions	Used to indicate a function that should be integrated directly into the caller's code.
`interface`	Interfaces	Used to declare and implement interfaces.
`internal`	Access Control	Used to specify that a member is visible inside an assembly but not outside it.
`lazy`	Lazy Expressions	Used to specify an expression that is to be performed only when a result is needed.
`let`	`let` Bindings	Used to associate, or bind, a name to a value or function.
`let!`	Asynchronous Workflows Computation Expressions	Used in asynchronous workflows to bind a name to the result of an asynchronous computation, or, in other computation expressions, used to bind a name to a result, which is of the computation type.
`match`	Match Expressions	Used to branch by comparing a value to a pattern.
`match!`	Computation Expressions	Used to inline a call to a computation expression and pattern match on its result.
`member`	Members	Used to declare a property or method in an object type.

KEYWORD	LINK	DESCRIPTION
`module`	Modules	Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.
`mutable`	let Bindings	Used to declare a variable, that is, a value that can be changed.
`namespace`	Namespaces	Used to associate a name with a group of related types and modules, to logically separate it from other code.
`new`	Constructors Constraints	Used to declare, define, or invoke a constructor that creates or that can create an object. Also used in generic parameter constraints to indicate that a type must have a certain constructor.
`not`	Symbol and Operator Reference Constraints	Not actually a keyword. However, `not struct` in combination is used as a generic parameter constraint.
`null`	Null Values Constraints	Indicates the absence of an object. Also used in generic parameter constraints.
`of`	Discriminated Unions Delegates Exception Types	Used in discriminated unions to indicate the type of categories of values, and in delegate and exception declarations.
`open`	Import Declarations: The `open` Keyword	Used to make the contents of a namespace or module available without qualification.
`or`	Symbol and Operator Reference Constraints	Used with Boolean conditions as a Boolean `or` operator. Equivalent to `||`. Also used in member constraints.
`override`	Members	Used to implement a version of an abstract or virtual method that differs from the base version.
`private`	Access Control	Restricts access to a member to code in the same type or module.
`public`	Access Control	Allows access to a member from outside the type.
`rec`	Functions	Used to indicate that a function is recursive.

KEYWORD	LINK	DESCRIPTION
`return`	Asynchronous Workflows Computation Expressions	Used to indicate a value to provide as the result of a computation expression.
`return!`	Computation Expressions Asynchronous Workflows	Used to indicate a computation expression that, when evaluated, provides the result of the containing computation expression.
`select`	Query Expressions	Used in query expressions to specify what fields or columns to extract. Note that this is a contextual keyword, which means that it is not actually a reserved word and it only acts like a keyword in appropriate context.
`static`	Members	Used to indicate a method or property that can be called without an instance of a type, or a value member that is shared among all instances of a type.
`struct`	Structures Tuples Constraints	Used to declare a structure type. Used to specify a struct tuple. Also used in generic parameter constraints. Used for OCaml compatibility in module definitions.
`then`	Conditional Expressions: `if...then...else` Constructors	Used in conditional expressions. Also used to perform side effects after object construction.
`to`	Loops: `for...to` Expression	Used in `for` loops to indicate a range.
`true`	Primitive Types	Used as a Boolean literal.
`try`	Exceptions: The try...with Expression Exceptions: The try...finally Expression	Used to introduce a block of code that might generate an exception. Used together with `with` or `finally`.

KEYWORD	LINK	DESCRIPTION
`type`	F# Types Classes Records Structures Enumerations Discriminated Unions Type Abbreviations Units of Measure	Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.
`upcast`	Casting and Conversions	Used to convert to a type that is higher in the inheritance chain.
`use`	Resource Management: The `use` Keyword	Used instead of `let` for values that require `Dispose` to be called to free resources.
`use!`	Computation Expressions Asynchronous Workflows	Used instead of `let!` in asynchronous workflows and other computation expressions for values that require `Dispose` to be called to free resources.
`val`	Explicit Fields: The `val` Keyword Signatures Members	Used in a signature to indicate a value, or in a type to declare a member, in limited situations.
`void`	Primitive Types	Indicates the .NET `void` type. Used when interoperating with other .NET languages.
`when`	Constraints	Used for Boolean conditions (*when guards*) on pattern matches and to introduce a constraint clause for a generic type parameter.
`while`	Loops: `while...do` Expression	Introduces a looping construct.
`with`	Match Expressions Object Expressions Copy and Update Record Expressions Type Extensions Exceptions: The `try...with` Expression	Used together with the `match` keyword in pattern matching expressions. Also used in object expressions, record copying expressions, and type extensions to introduce member definitions, and to introduce exception handlers.

KEYWORD	LINK	DESCRIPTION
`yield`	Sequences	Used in a sequence expression to produce a value for a sequence.
`yield!`	Computation Expressions Asynchronous Workflows	Used in a computation expression to append the result of a given computation expression to a collection of results for the containing computation expression.

The following tokens are reserved in F# because they are keywords in the OCaml language:

- `asr`
- `land`
- `lor`
- `lsl`
- `lsr`
- `lxor`
- `mod`
- `sig`

If you use the `--mlcompatibility` compiler option, the above keywords are available for use as identifiers.

The following tokens are reserved as keywords for future expansion of the F# language:

- `atomic`
- `break`
- `checked`
- `component`
- `const`
- `constraint`
- `constructor`
- `continue`
- `eager`
- `event`
- `external`
- `functor`
- `include`
- `method`
- `mixin`
- `object`
- `parallel`
- `process`
- `protected`
- `pure`
- `sealed`
- `tailcall`
- `trait`

- `virtual`
- `volatile`

See also

- F# Language Reference
- Symbol and Operator Reference
- Compiler Options

Symbol and Operator Reference

6/21/2019 • 8 minutes to read • Edit Online

NOTE

The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

This topic includes a table of symbols and operators that are used in the F# language.

Table of Symbols and Operators

The following table describes symbols used in the F# language, provides links to topics that provide more information, and provides a brief description of some of the uses of the symbol. Symbols are ordered according to the ASCII character set ordering.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`!`	Reference Cells Computation Expressions	• Dereferences a reference cell. • After a keyword, indicates a modified version of the keyword's behavior as controlled by a workflow.
`!=`	Not applicable.	• Not used in F#. Use `<>` for inequality operations.
`"`	Literals Strings	• Delimits a text string.
`"""`	Strings	Delimits a verbatim text string. Differs from `@"..."` in that a you can indicate a quotation mark character by using a single quote in the string.
`#`	Compiler Directives Flexible Types	• Prefixes a preprocessor or compiler directive, such as `#light`. • When used with a type, indicates a *flexible type*, which refers to a type or any one of its derived types.
`$`	No more information available.	• Used internally for certain compiler-generated variable and function names.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`%`	Arithmetic Operators Code Quotations	• Computes the integer remainder. • Used for splicing expressions into typed code quotations.
`%%`	Code Quotations	• Used for splicing expressions into untyped code quotations.
`%?`	Nullable Operators	• Computes the integer remainder, when the right side is a nullable type.
`&`	Match Expressions	• Computes the address of a mutable value, for use when interoperating with other languages. • Used in AND patterns.
`&&`	Boolean Operators	• Computes the Boolean AND operation.
`&&&`	Bitwise Operators	• Computes the bitwise AND operation.
`'`	Literals Automatic Generalization	• Delimits a single-character literal. • Indicates a generic type parameter.
``` ``...`` ```	No more information available.	• Delimits an identifier that would otherwise not be a legal identifier, such as a language keyword.
`( )`	Unit Type	• Represents the single value of the unit type.
`(...)`	Tuples Operator Overloading	• Indicates the order in which expressions are evaluated. • Delimits a tuple. • Used in operator definitions.
`(*...*)`		• Delimits a comment that could span multiple lines.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`(|...|)`	Active Patterns	• Delimits an active pattern. Also called *banana clips*.
`*`	Arithmetic Operators Tuples Units of Measure	• When used as a binary operator, multiplies the left and right sides. • In types, indicates pairing in a tuple. • Used in units of measure types.
`*?`	Nullable Operators	• Multiplies the left and right sides, when the right side is a nullable type.
`**`	Arithmetic Operators	• Computes the exponentiation operation (`x ** y` means `x` to the power of `y`).
`+`	Arithmetic Operators	• When used as a binary operator, adds the left and right sides. • When used as a unary operator, indicates a positive quantity. (Formally, it produces the same value with the sign unchanged.)
`+?`	Nullable Operators	• Adds the left and right sides, when the right side is a nullable type.
`,`	Tuples	• Separates the elements of a tuple, or type parameters.
`-`	Arithmetic Operators	• When used as a binary operator, subtracts the right side from the left side. • When used as a unary operator, performs a negation operation.
`-?`	Nullable Operators	• Subtracts the right side from the left side, when the right side is a nullable type.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`->`	Functions Match Expressions	• In function types, delimits arguments and return values. • Yields an expression (in sequence expressions); equivalent to the `yield` keyword. • Used in match expressions
`.`	Members Primitive Types	• Accesses a member, and separates individual names in a fully qualified name. • Specifies a decimal point in floating point numbers.
`..`	Loops: `for...in` Expression	• Specifies a range.
`.. ..`	Loops: `for...in` Expression	• Specifies a range together with an increment.
`.[...]`	Arrays	• Accesses an array element.
`/`	Arithmetic Operators Units of Measure	• Divides the left side (numerator) by the right side (denominator). • Used in units of measure types.
`/?`	Nullable Operators	• Divides the left side by the right side, when the right side is a nullable type.
`//`		• Indicates the beginning of a single-line comment.
`///`	XML Documentation	• Indicates an XML comment.
`:`	Functions	• In a type annotation, separates a parameter or member name from its type.
`::`	Lists Match Expressions	• Creates a list. The element on the left side is prepended to the list on the right side. • Used in pattern matching to separate the parts of a list.
`:=`	Reference Cells	• Assigns a value to a reference cell.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`:>`	Casting and Conversions	• Converts a type to type that is higher in the hierarchy.
`:?`	Match Expressions	• Returns `true` if the value matches the specified type (including if it is a subtype); otherwise, returns `false` (type test operator).
`:?>`	Casting and Conversions	• Converts a type to a type that is lower in the hierarchy.
`;`	Verbose Syntax Lists Records	• Separates expressions (used mostly in verbose syntax). • Separates elements of a list. • Separates fields of a record.
`<`	Arithmetic Operators	• Computes the less-than operation.
`<?`	Nullable Operators	Computes the less than operation, when the right side is a nullable type.
`<<`	Functions	• Composes two functions in reverse order; the second one is executed first (backward composition operator).
`<<<`	Bitwise Operators	• Shifts bits in the quantity on the left side to the left by the number of bits specified on the right side.
`<-`	Values	• Assigns a value to a variable.
`<...>`	Automatic Generalization	• Delimits type parameters.
`<>`	Arithmetic Operators	• Returns `true` if the left side is not equal to the right side; otherwise, returns false.
`<>?`	Nullable Operators	• Computes the "not equal" operation when the right side is a nullable type.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`<=`	Arithmetic Operators	• Returns `true` if the left side is less than or equal to the right side; otherwise, returns `false`.
`<=?`	Nullable Operators	• Computes the "less than or equal to" operation when the right side is a nullable type.
`<|`	Functions	• Passes the result of the expression on the right side to the function on left side (backward pipe operator).
`<||`	Operators.(<||)<'T1,'T2,'U> Function	• Passes the tuple of two arguments on the right side to the function on left side.
`<|||`	Operators.(<|||)<'T1,'T2,'T3,'U> Function	• Passes the tuple of three arguments on the right side to the function on left side.
`<@...@>`	Code Quotations	• Delimits a typed code quotation.
`<@@...@@>`	Code Quotations	• Delimits an untyped code quotation.
`=`	Arithmetic Operators	• Returns `true` if the left side is equal to the right side; otherwise, returns `false`.
`=?`	Nullable Operators	• Computes the "equal" operation when the right side is a nullable type.
`==`	Not applicable.	• Not used in F#. Use `=` for equality operations.
`>`	Arithmetic Operators	• Returns `true` if the left side is greater than the right side; otherwise, returns `false`.
`>?`	Nullable Operators	• Computes the "greater than" operation when the right side is a nullable type.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`>>`	Functions	• Composes two functions (forward composition operator).
`>>>`	Bitwise Operators	• Shifts bits in the quantity on the left side to the right by the number of places specified on the right side.
`>=`	Arithmetic Operators	• Returns `true` if the left side is greater than or equal to the right side; otherwise, returns `false`.
`>=?`	Nullable Operators	• Computes the "greater than or equal" operation when the right side is a nullable type.
`?`	Parameters and Arguments	• Specifies an optional argument. • Used as an operator for dynamic method and property calls. You must provide your own implementation.
`? ... <- ...`	No more information available.	• Used as an operator for setting dynamic properties. You must provide your own implementation.
`?>=`, `?>`, `?<=`, `?<`, `?=`, `?<>`, `?+`, `?-`, `?*`, `?/`	Nullable Operators	• Equivalent to the corresponding operators without the ? prefix, where a nullable type is on the left.
`>=?`, `>?`, `<=?`, `<?`, `=?`, `<>?`, `+?`, `-?`, `*?`, `/?`	Nullable Operators	• Equivalent to the corresponding operators without the ? suffix, where a nullable type is on the right.
`?>=?`, `?>?`, `?<=?`, `?<?`, `?=?`, `?<>?`, `?+?`, `?-?`, `?*?`, `?/?`	Nullable Operators	• Equivalent to the corresponding operators without the surrounding question marks, where both sides are nullable types.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`@`	Lists Strings	• Concatenates two lists. • When placed before a string literal, indicates that the string is to be interpreted verbatim, with no interpretation of escape characters.
`[...]`	Lists	• Delimits the elements of a list.
`[|...|]`	Arrays	• Delimits the elements of an array.
`[<...>]`	Attributes	• Delimits an attribute.
`\`	Strings	• Escapes the next character; used in character and string literals.
`^`	Statically Resolved Type Parameters Strings	• Specifies type parameters that must be resolved at compile time, not at runtime. • Concatenates strings.
`^^^`	Bitwise Operators	• Computes the bitwise exclusive OR operation.
`_`	Match Expressions Generics	• Indicates a wildcard pattern. • Specifies an anonymous generic parameter.
`` ` ``	Automatic Generalization	• Used internally to indicate a generic type parameter.
`{...}`	Sequences Records	• Delimits sequence expressions and computation expressions. • Used in record definitions.
`|`	Match Expressions	• Delimits individual match cases, individual discriminated union cases, and enumeration values.
`||`	Boolean Operators	• Computes the Boolean OR operation.

SYMBOL OR OPERATOR	LINKS	DESCRIPTION
`|||`	Bitwise Operators	• Computes the bitwise OR operation.
`|>`	Functions	• Passes the result of the left side to the function on the right side (forward pipe operator).
`||>`	Operators.(||>)<'T1,'T2,'U> Function	• Passes the tuple of two arguments on the left side to the function on the right side.
`|||>`	Operators.(|||>)<'T1,'T2,'T3,'U> Function	• Passes the tuple of three arguments on the left side to the function on the right side.
`~~`	Operator Overloading	• Used to declare an overload for the unary negation operator.
`~~~`	Bitwise Operators	• Computes the bitwise NOT operation.
`~-`	Operator Overloading	• Used to declare an overload for the unary minus operator.
`~+`	Operator Overloading	• Used to declare an overload for the unary plus operator.

Operator Precedence

The following table shows the order of precedence of operators and other expression keywords in the F# language, in order from lowest precedence to the highest precedence. Also listed is the associativity, if applicable.

OPERATOR	ASSOCIATIVITY
`as`	Right
`when`	Right
`|` (pipe)	Left
`;`	Right
`let`	Nonassociative
`function`, `fun`, `match`, `try`	Nonassociative

OPERATOR	ASSOCIATIVITY
`if`	Nonassociative
`not`	Right
`->`	Right
`:=`	Right
`,`	Nonassociative
`or` , `||`	Left
`&` , `&&`	Left
`:>` , `:?>`	Right
`<` *op*, `>` *op*, `=` , `|` *op*, `&` *op*, `&` (including `<<<` , `>>>` , `|||` , `&&&`)	Left
`^` *op* (including `^^^`)	Right
`::`	Right
`:?`	Not associative
`-` *op*, `+` *op*	Applies to infix uses of these symbols
`*` *op*, `/` *op*, `%` *op*	Left
`**` *op*	Right
`f x` (function application)	Left
`|` (pattern match)	Right
prefix operators (`+` *op*, `-` *op*, `%` , `%%` , `&` , `&&` , `!` *op*, `~` *op*)	Left
`.`	Left
`f(x)`	Left
`f<` *types* `>`	Left

F# supports custom operator overloading. This means that you can define your own operators. In the previous table, *op* can be any valid (possibly empty) sequence of operator characters, either built-in or user-defined. Thus, you can use this table to determine what sequence of characters to use for a custom operator to achieve the

desired level of precedence. Leading `.` characters are ignored when the compiler determines precedence.

See also

- F# Language Reference
- Operator Overloading

Arithmetic Operators

7/30/2019 • 3 minutes to read • Edit Online

This topic describes arithmetic operators that are available in the F# language.

Summary of Binary Arithmetic Operators

The following table summarizes the binary arithmetic operators that are available for unboxed integral and floating-point types.

BINARY OPERATOR	NOTES
`+` (addition, plus)	Unchecked. Possible overflow condition when numbers are added together and the sum exceeds the maximum absolute value supported by the type.
`-` (subtraction, minus)	Unchecked. Possible underflow condition when unsigned types are subtracted, or when floating-point values are too small to be represented by the type.
`*` (multiplication, times)	Unchecked. Possible overflow condition when numbers are multiplied and the product exceeds the maximum absolute value supported by the type.
`/` (division, divided by)	Division by zero causes a DivideByZeroException for integral types. For floating-point types, division by zero gives you the special floating-point values `+Infinity` or `-Infinity`. There is also a possible underflow condition when a floating-point number is too small to be represented by the type.
`%` (remainder, rem)	Returns the remainder of a division operation. The sign of the result is the same as the sign of the first operand.
`**` (exponentiation, to the power of)	Possible overflow condition when the result exceeds the maximum absolute value for the type. The exponentiation operator works only with floating-point types.

Summary of Unary Arithmetic Operators

The following table summarizes the unary arithmetic operators that are available for integral and floating-point types.

UNARY OPERATOR	NOTES
`+` (positive)	Can be applied to any arithmetic expression. Does not change the sign of the value.
`-` (negation, negative)	Can be applied to any arithmetic expression. Changes the sign of the value.

The behavior at overflow or underflow for integral types is to wrap around. This causes an incorrect result. Integer overflow is a potentially serious problem that can contribute to security issues when software is not written to account for it. If this is a concern for your application, consider using the checked operators in `Microsoft.FSharp.Core.Operators.Checked`.

Summary of Binary Comparison Operators

The following table shows the binary comparison operators that are available for integral and floating-point types. These operators return values of type `bool`.

Floating-point numbers should never be directly compared for equality, because the IEEE floating-point representation does not support an exact equality operation. Two numbers that you can easily verify to be equal by inspecting the code might actually have different bit representations.

OPERATOR	NOTES
`=` (equality, equals)	This is not an assignment operator. It is used only for comparison. This is a generic operator.
`>` (greater than)	This is a generic operator.
`<` (less than)	This is a generic operator.
`>=` (greater than or equals)	This is a generic operator.
`<=` (less than or equals)	This is a generic operator.
`<>` (not equal)	This is a generic operator.

Overloaded and Generic Operators

All of the operators discussed in this topic are defined in the **Microsoft.FSharp.Core.Operators** namespace. Some of the operators are defined by using statically resolved type parameters. This means that there are individual definitions for each specific type that works with that operator. All of the unary and binary arithmetic and bitwise operators are in this category. The comparison operators are generic and therefore work with any type, not just primitive arithmetic types. Discriminated union and record types have their own custom implementations that are generated by the F# compiler. Class types use the method Equals.

The generic operators are customizable. To customize the comparison functions, override Equals to provide your own custom equality comparison, and then implement IComparable. The System.IComparable interface has a single method, the CompareTo method.

Operators and Type Inference

The use of an operator in an expression constrains type inference on that operator. Also, the use of operators prevents automatic generalization, because the use of operators implies an arithmetic type. In the absence of any other information, the F# compiler infers `int` as the type of arithmetic expressions. You can override this behavior by specifying another type. Thus the argument types and return type of `function1` in the following code are inferred to be `int`, but the types for `function2` are inferred to be `float`.

```
// x, y and return value inferred to be int
// function1: int -> int -> int
let function1 x y = x + y

// x, y and return value inferred to be float
// function2: float -> float -> float
let function2 (x: float) y = x + y
```

See also

- Symbol and Operator Reference
- Operator Overloading
- Bitwise Operators
- Boolean Operators

Boolean Operators

5/15/2019 • 2 minutes to read • Edit Online

This topic describes the support for Boolean operators in the F# language.

Summary of Boolean Operators

The following table summarizes the Boolean operators that are available in the F# language. The only type supported by these operators is the `bool` type.

OPERATOR	DESCRIPTION
`not`	Boolean negation
`||`	Boolean OR
`&&`	Boolean AND

The Boolean AND and OR operators perform *short-circuit evaluation*, that is, they evaluate the expression on the right of the operator only when it is necessary to determine the overall result of the expression. The second expression of the `&&` operator is evaluated only if the first expression evaluates to `true`; the second expression of the `||` operator is evaluated only if the first expression evaluates to `false`.

See also

- Bitwise Operators
- Arithmetic Operators
- Symbol and Operator Reference

Bitwise Operators

5/15/2019 • 2 minutes to read • Edit Online

This topic describes bitwise operators that are available in the F# language.

Summary of Bitwise Operators

The following table describes the bitwise operators that are supported for unboxed integral types in the F# language.

OPERATOR	NOTES
`&&&`	Bitwise AND operator. Bits in the result have the value 1 if and only if the corresponding bits in both source operands are 1.
`|||`	Bitwise OR operator. Bits in the result have the value 1 if either of the corresponding bits in the source operands are 1.
`^^^`	Bitwise exclusive OR operator. Bits in the result have the value 1 if and only if bits in the source operands have unequal values.
`~~~`	Bitwise negation operator. This is a unary operator and produces a result in which all 0 bits in the source operand are converted to 1 bits and all 1 bits are converted to 0 bits.
`<<<`	Bitwise left-shift operator. The result is the first operand with bits shifted left by the number of bits in the second operand. Bits shifted off the most significant position are not rotated into the least significant position. The least significant bits are padded with zeros. The type of the second argument is `int32`.
`>>>`	Bitwise right-shift operator. The result is the first operand with bits shifted right by the number of bits in the second operand. Bits shifted off the least significant position are not rotated into the most significant position. For unsigned types, the most significant bits are padded with zeros. For signed types with negative values, the most significant bits are padded with ones. The type of the second argument is `int32`.

The following types can be used with bitwise operators: `byte`, `sbyte`, `int16`, `uint16`, `int32 (int)`, `uint32`, `int64`, `uint64`, `nativeint`, and `unativeint`.

See also

- Symbol and Operator Reference
- Arithmetic Operators
- Boolean Operators

Nullable Operators

5/15/2019 • 3 minutes to read • Edit Online

Nullable operators are binary arithmetic or comparison operators that work with nullable arithmetic types on one or both sides. Nullable types arise frequently when you work with data from sources such as databases that allow nulls in place of actual values. Nullable operators are used frequently in query expressions. In addition to nullable operators for arithmetic and comparison, conversion operators can be used to convert between nullable types. There are also nullable versions of certain query operators.

Table of Nullable Operators

The following table lists nullable operators supported in the F# language.

NULLABLE ON LEFT	NULLABLE ON RIGHT	BOTH SIDES NULLABLE
?>=	>=?	?>=?
?>	>?	?>?
?<=	<=?	?<=?
?<	<?	?<?
?=	=?	?=?
?<>	<>?	?<>?
?+	+?	?+?
?-	-?	?-?
?*	*?	?*?
?/	/?	?/?
?%	%?	?%?

Remarks

The nullable operators are included in the NullableOperators module in the namespace Microsoft.FSharp.Linq. The type for nullable data is `System.Nullable<'T>`.

In query expressions, nullable types arise when selecting data from a data source that allows nulls instead of values. In a SQL Server database, each data column in a table has an attribute that indicates whether nulls are allowed. If nulls are allowed, the data returned from the database can contain nulls that cannot be represented by a primitive data type such as `int`, `float`, and so on. Therefore, the data is returned as a `System.Nullable<int>` instead of `int`, and `System.Nullable<float>` instead of `float`. The actual value can be obtained from a `System.Nullable<'T>` object by using the `Value` property, and you can determine if a `System.Nullable<'T>` object has a value by calling the `HasValue` method. Another useful method is the

`System.Nullable<'T>.GetValueOrDefault` method, which allows you to get the value or a default value of the appropriate type. The default value is some form of "zero" value, such as 0, 0.0, or `false`.

Nullable types may be converted to non-nullable primitive types using the usual conversion operators such as `int` or `float`. It is also possible to convert from one nullable type to another nullable type by using the conversion operators for nullable types. The appropriate conversion operators have the same name as the standard ones, but they are in a separate module, the Nullable module in the Microsoft.FSharp.Linq namespace. Typically, you open this namespace when working with query expressions. In that case, you can use the nullable conversion operators by adding the prefix `Nullable.` to the appropriate conversion operator, as shown in the following code.

```
open Microsoft.FSharp.Linq

let nullableInt = new System.Nullable<int>(10)

// Use the Nullable.float conversion operator to convert from one nullable type to another nullable type.
let nullableFloat = Nullable.float nullableInt

// Use the regular non-nullable float operator to convert to a non-nullable float.
printfn "%f" (float nullableFloat)
```

The output is `10.000000`.

Query operators on nullable data fields, such as `sumByNullable`, also exist for use in query expressions. The query operators for non-nullable types are not type-compatible with nullable types, so you must use the nullable version of the appropriate query operator when you are working with nullable data values. For more information, see Query Expressions.

The following example shows the use of nullable operators in an F# query expression. The first query shows how you would write a query without a nullable operator; the second query shows an equivalent query that uses a nullable operator. For the full context, including how to set up the database to use this sample code, see Walkthrough: Accessing a SQL Database by Using Type Providers.

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

[<Generate>]
type dbSchema = SqlDataConnection<"Data Source=MYSERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = dbSchema.GetDataContext()

query {
    for row in db.Table2 do
    where (row.TestData1.HasValue && row.TestData1.Value > 2)
    select row
} |> Seq.iter (fun row -> printfn "%d %s" row.TestData1.Value row.Name)

query {
    for row in db.Table2 do
    // Use a nullable operator ?>
    where (row.TestData1 ?> 2)
    select row
} |> Seq.iter (fun row -> printfn "%d %s" (row.TestData1.GetValueOrDefault()) row.Name)
```

See also

- Type Providers
- Query Expressions

Functions

7/30/2019 • 9 minutes to read • Edit Online

Functions are the fundamental unit of program execution in any programming language. As in other languages, an F# function has a name, can have parameters and take arguments, and has a body. F# also supports functional programming constructs such as treating functions as values, using unnamed functions in expressions, composition of functions to form new functions, curried functions, and the implicit definition of functions by way of the partial application of function arguments.

You define functions by using the `let` keyword, or, if the function is recursive, the `let rec` keyword combination.

Syntax

```
// Non-recursive function definition.
let [inline] function-name parameter-list [ : return-type ] = function-body
// Recursive function definition.
let rec function-name parameter-list = recursive-function-body
```

Remarks

The *function-name* is an identifier that represents the function. The *parameter-list* consists of successive parameters that are separated by spaces. You can specify an explicit type for each parameter, as described in the Parameters section. If you do not specify a specific argument type, the compiler attempts to infer the type from the function body. The *function-body* consists of an expression. The expression that makes up the function body is typically a compound expression consisting of a number of expressions that culminate in a final expression that is the return value. The *return-type* is a colon followed by a type and is optional. If you do not specify the type of the return value explicitly, the compiler determines the return type from the final expression.

A simple function definition resembles the following:

```
let f x = x + 1
```

In the previous example, the function name is `f`, the argument is `x`, which has type `int`, the function body is `x + 1`, and the return value is of type `int`.

Functions can be marked `inline`. For information about `inline`, see Inline Functions.

Scope

At any level of scope other than module scope, it is not an error to reuse a value or function name. If you reuse a name, the name declared later shadows the name declared earlier. However, at the top level scope in a module, names must be unique. For example, the following code produces an error when it appears at module scope, but not when it appears inside a function:

```
let list1 = [ 1; 2; 3]
// Error: duplicate definition.
let list1 = []
let function1 =
   let list1 = [1; 2; 3]
   let list1 = []
   list1
```

But the following code is acceptable at any level of scope:

```
let list1 = [ 1; 2; 3]
let sumPlus x =
// OK: inner list1 hides the outer list1.
   let list1 = [1; 5; 10]
   x + List.sum list1
```

Parameters

Names of parameters are listed after the function name. You can specify a type for a parameter, as shown in the following example:

```
let f (x : int) = x + 1
```

If you specify a type, it follows the name of the parameter and is separated from the name by a colon. If you omit the type for the parameter, the parameter type is inferred by the compiler. For example, in the following function definition, the argument `x` is inferred to be of type `int` because 1 is of type `int`.

```
let f x = x + 1
```

However, the compiler will attempt to make the function as generic as possible. For example, note the following code:

```
let f x = (x, x)
```

The function creates a tuple from one argument of any type. Because the type is not specified, the function can be used with any argument type. For more information, see Automatic Generalization.

Function Bodies

A function body can contain definitions of local variables and functions. Such variables and functions are in scope in the body of the current function but not outside it. When you have the lightweight syntax option enabled, you must use indentation to indicate that a definition is in a function body, as shown in the following example:

```
let cylinderVolume radius length =
    // Define a local value pi.
    let pi = 3.14159
    length * pi * radius * radius
```

For more information, see Code Formatting Guidelines and Verbose Syntax.

Return Values

The compiler uses the final expression in a function body to determine the return value and type. The compiler might infer the type of the final expression from previous expressions. In the function `cylinderVolume`, shown in the previous section, the type of `pi` is determined from the type of the literal `3.14159` to be `float`. The compiler uses the type of `pi` to determine the type of the expression `h * pi * r * r` to be `float`. Therefore, the overall return type of the function is `float`.

To specify the return value explicitly, write the code as follows:

```
let cylinderVolume radius length : float =
   // Define a local value pi.
   let pi = 3.14159
   length * pi * radius * radius
```

As the code is written above, the compiler applies **float** to the entire function; if you mean to apply it to the parameter types as well, use the following code:

```
let cylinderVolume (radius : float) (length : float) : float
```

Calling a Function

You call functions by specifying the function name followed by a space and then any arguments separated by spaces. For example, to call the function **cylinderVolume** and assign the result to the value **vol**, you write the following code:

```
let vol = cylinderVolume 2.0 3.0
```

Partial Application of Arguments

If you supply fewer than the specified number of arguments, you create a new function that expects the remaining arguments. This method of handling arguments is referred to as *currying* and is a characteristic of functional programming languages like F#. For example, suppose you are working with two sizes of pipe: one has a radius of **2.0** and the other has a radius of **3.0**. You could create functions that determine the volume of pipe as follows:

```
let smallPipeRadius = 2.0
let bigPipeRadius = 3.0

// These define functions that take the length as a remaining
// argument:

let smallPipeVolume = cylinderVolume smallPipeRadius
let bigPipeVolume = cylinderVolume bigPipeRadius
```

You would then supply the additional argument as needed for various lengths of pipe of the two different sizes:

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

Recursive Functions

Recursive functions are functions that call themselves. They require that you specify the **rec** keyword following the **let** keyword. Invoke the recursive function from within the body of the function just as you would invoke any function call. The following recursive function computes the n^{th} Fibonacci number. The Fibonacci number sequence has been known since antiquity and is a sequence in which each successive number is the sum of the previous two numbers in the sequence.

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

Some recursive functions might overflow the program stack or perform inefficiently if you do not write them with care and with awareness of special techniques, such as the use of accumulators and continuations.

Function Values

In F#, all functions are considered values; in fact, they are known as *function values*. Because functions are values, they can be used as arguments to other functions or in other contexts where values are used. Following is an example of a function that takes a function value as an argument:

```
let apply1 (transform : int -> int ) y = transform y
```

You specify the type of a function value by using the `->` token. On the left side of this token is the type of the argument, and on the right side is the return value. In the previous example, `apply1` is a function that takes a function `transform` as an argument, where `transform` is a function that takes an integer and returns another integer. The following code shows how to use `apply1`:

```
let increment x = x + 1

let result1 = apply1 increment 100
```

The value of `result` will be 101 after the previous code runs.

Multiple arguments are separated by successive `->` tokens, as shown in the following example:

```
let apply2 ( f: int -> int -> int) x y = f x y

let mul x y = x * y

let result2 = apply2 mul 10 20
```

The result is 200.

Lambda Expressions

A *lambda expression* is an unnamed function. In the previous examples, instead of defining named functions **increment** and **mul**, you could use lambda expressions as follows:

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

You define lambda expressions by using the `fun` keyword. A lambda expression resembles a function

definition, except that instead of the `=` token, the `->` token is used to separate the argument list from the function body. As in a regular function definition, the argument types can be inferred or specified explicitly, and the return type of the lambda expression is inferred from the type of the last expression in the body. For more information, see Lambda Expressions: The `fun` Keyword.

Function Composition and Pipelining

Functions in F# can be composed from other functions. The composition of two functions **function1** and **function2** is another function that represents the application of **function1** followed the application of **function2**:

```
let function1 x = x + 1
let function2 x = x * 2
let h = function1 >> function2
let result5 = h 100
```

The result is 202.

Pipelining enables function calls to be chained together as successive operations. Pipelining works as follows:

```
let result = 100 |> function1 |> function2
```

The result is again 202.

The composition operators take two functions and return a function; by contrast, the pipeline operators take a function and an argument and return a value. The following code example shows the difference between the pipeline and composition operators by showing the differences in the function signatures and usage.

```
// Function composition and pipeline operators compared.

let addOne x = x + 1
let timesTwo x = 2 * x

// Composition operator
// ( >> ) : ('T1 -> 'T2) -> ('T2 -> 'T3) -> 'T1 -> 'T3
let Compose2 = addOne >> timesTwo

// Backward composition operator
// ( << ) : ('T2 -> 'T3) -> ('T1 -> 'T2) -> 'T1 -> 'T3
let Compose1 = addOne << timesTwo

// Result is 5
let result1 = Compose1 2

// Result is 6
let result2 = Compose2 2

// Pipelining
// Pipeline operator
// ( |> ) : 'T1 -> ('T1 -> 'U) -> 'U
let Pipeline2 x = addOne x |> timesTwo

// Backward pipeline operator
// ( <| ) : ('T -> 'U) -> 'T -> 'U
let Pipeline1 x = addOne <| timesTwo x

// Result is 5
let result3 = Pipeline1 2

// Result is 6
let result4 = Pipeline2 2
```

Overloading Functions

You can overload methods of a type but not functions. For more information, see Methods.

See also

- Values
- F# Language Reference

let Bindings

7/30/2019 • 5 minutes to read • Edit Online

A *binding* associates an identifier with a value or function. You use the `let` keyword to bind a name to a value or function.

Syntax

```
// Binding a value:
let identifier-or-pattern [: type] =expressionbody-expression
// Binding a function value:
let identifier parameter-list [: return-type ] =expressionbody-expression
```

Remarks

The `let` keyword is used in binding expressions to define values or function values for one or more names. The simplest form of the `let` expression binds a name to a simple value, as follows.

```
let i = 1
```

If you separate the expression from the identifier by using a new line, you must indent each line of the expression, as in the following code.

```
let someVeryLongIdentifier =
    // Note indentation below.
    3 * 4 + 5 * 6
```

Instead of just a name, a pattern that contains names can be specified, for example, a tuple, as shown in the following code.

```
let i, j, k = (1, 2, 3)
```

The *body-expression* is the expression in which the names are used. The body expression appears on its own line, indented to line up exactly with the first character in the `let` keyword:

```
let result =

    let i, j, k = (1, 2, 3)

    // Body expression:
    i + 2*j + 3*k
```

A `let` binding can appear at the module level, in the definition of a class type, or in local scopes, such as in a function definition. A `let` binding at the top level in a module or in a class type does not need to have a body expression, but at other scope levels, the body expression is required. The bound names are usable after the point of definition, but not at any point before the `let` binding appears, as is illustrated in the following code.

```
// Error:
printfn "%d" x
let x = 100
// OK:
printfn "%d" x
```

Function Bindings

Function bindings follow the rules for value bindings, except that function bindings include the function name and the parameters, as shown in the following code.

```
let function1 a =
    a + 1
```

In general, parameters are patterns, such as a tuple pattern:

```
let function2 (a, b) = a + b
```

A `let` binding expression evaluates to the value of the last expression. Therefore, in the following code example, the value of `result` is computed from `100 * function3 (1, 2)`, which evaluates to `300`.

```
let result =
    let function3 (a, b) = a + b
    100 * function3 (1, 2)
```

For more information, see Functions.

Type Annotations

You can specify types for parameters by including a colon (:) followed by a type name, all enclosed in parentheses. You can also specify the type of the return value by appending the colon and type after the last parameter. The full type annotations for `function1`, with integers as the parameter types, would be as follows.

```
let function1 (a: int) : int = a + 1
```

When there are no explicit type parameters, type inference is used to determine the types of parameters of functions. This can include automatically generalizing the type of a parameter to be generic.

For more information, see Automatic Generalization and Type Inference.

let Bindings in Classes

A `let` binding can appear in a class type but not in a structure or record type. To use a let binding in a class type, the class must have a primary constructor. Constructor parameters must appear after the type name in the class definition. A `let` binding in a class type defines private fields and members for that class type and, together with `do` bindings in the type, forms the code for the primary constructor for the type. The following code examples show a class `MyClass` with private fields `field1` and `field2`.

```
type MyClass(a) =
    let field1 = a
    let field2 = "text"
    do printfn "%d %s" field1 field2
    member this.F input =
        printfn "Field1 %d Field2 %s Input %A" field1 field2 input
```

The scopes of `field1` and `field2` are limited to the type in which they are declared. For more information, see `let` Bindings in Classes and Classes.

Type Parameters in let Bindings

A `let` binding at the module level, in a type, or in a computation expression can have explicit type parameters. A let binding in an expression, such as within a function definition, cannot have type parameters. For more information, see Generics.

Attributes on let Bindings

Attributes can be applied to top-level `let` bindings in a module, as shown in the following code.

```
[<Obsolete>]
let function1 x y = x + y
```

Scope and Accessibility of Let Bindings

The scope of an entity declared with a let binding is limited to the portion of the containing scope (such as a function, module, file or class) after the binding appears. Therefore, it can be said that a let binding introduces a name into a scope. In a module, a let-bound value or function is accessible to clients of a module as long as the module is accessible, since the let bindings in a module are compiled into public functions of the module. By contrast, let bindings in a class are private to the class.

Normally, functions in modules must be qualified by the name of the module when used by client code. For example, if a module `Module1` has a function `function1`, users would specify `Module1.function1` to refer to the function.

Users of a module may use an import declaration to make the functions within that module available for use without being qualified by the module name. In the example just mentioned, users of the module can in that case open the module by using the import declaration open `Module1` and thereafter refer to `function1` directly.

```
module Module1 =
    let function1 x = x + 1.0

module Module2 =
    let function2 x =
        Module1.function1 x

open Module1

let function3 x =
    function1 x
```

Some modules have the attribute RequireQualifiedAccess, which means that the functions that they expose must be qualified with the name of the module. For example, the F# List module has this attribute.

For more information on modules and access control, see Modules and Access Control.

See also

- Functions
- `let` Bindings in Classes

do Bindings

7/30/2019 • 2 minutes to read • Edit Online

A `do` binding is used to execute code without defining a function or value. Also, do bindings can be used in classes, see `do` Bindings in Classes.

Syntax

```
[ attributes ]
[ do ]expression
```

Remarks

Use a `do` binding when you want to execute code independently of a function or value definition. The expression in a `do` binding must return `unit`. Code in a top-level `do` binding is executed when the module is initialized. The keyword `do` is optional.

Attributes can be applied to a top-level `do` binding. For example, if your program uses COM interop, you might want to apply the `STAThread` attribute to your program. You can do this by using an attribute on a `do` binding, as shown in the following code.

```
open System
open System.Windows.Forms

let form1 = new Form()
form1.Text <- "XYZ"

[<STAThread>]
do
   Application.Run(form1)
```

See also

- F# Language Reference
- Functions

Lambda Expressions: The fun Keyword (F#)

7/30/2019 • 2 minutes to read • Edit Online

The `fun` keyword is used to define a lambda expression, that is, an anonymous function.

Syntax

```
fun parameter-list -> expression
```

Remarks

The *parameter-list* typically consists of names and, optionally, types of parameters. More generally, the *parameter-list* can be composed of any F# patterns. For a full list of possible patterns, see Pattern Matching. Lists of valid parameters include the following examples.

```
// Lambda expressions with parameter lists.
fun a b c -> ...
fun (a: int) b c -> ...
fun (a : int) (b : string) (c:float) -> ...

// A lambda expression with a tuple pattern.
fun (a, b) -> …

// A lambda expression with a list pattern.
fun head :: tail -> …
```

The *expression* is the body of the function, the last expression of which generates a return value. Examples of valid lambda expressions include the following:

```
fun x -> x + 1
fun a b c -> printfn "%A %A %A" a b c
fun (a: int) (b: int) (c: int) -> a + b * c
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

Using Lambda Expressions

Lambda expressions are especially useful when you want to perform operations on a list or other collection and want to avoid the extra work of defining a function. Many F# library functions take function values as arguments, and it can be especially convenient to use a lambda expression in those cases. The following code applies a lambda expression to elements of a list. In this case, the anonymous function adds 1 to every element of a list.

```
let list = List.map (fun i -> i + 1) [1;2;3]
printfn "%A" list
```

See also

- Functions

Recursive Functions: The rec Keyword

7/30/2019 • 2 minutes to read • Edit Online

The `rec` keyword is used together with the `let` keyword to define a recursive function.

Syntax

```
// Recursive function:
let rec function-nameparameter-list =
function-body

// Mutually recursive functions:
let rec function1-nameparameter-list =
function1-body
and function2-nameparameter-list =
function2-body
...
```

Remarks

Recursive functions, functions that call themselves, are identified explicitly in the F# language. This makes the identifer that is being defined available in the scope of the function.

The following code illustrates a recursive function that computes the n^{th} Fibonacci number.

```
let rec fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

> **NOTE**
> In practice, code like that above is wasteful of memory and processor time because it involves the recomputation of previously computed values.

Methods are implicitly recursive within the type; there is no need to add the `rec` keyword. Let bindings within classes are not implicitly recursive.

Mutually Recursive Functions

Sometimes functions are *mutually recursive*, meaning that calls form a circle, where one function calls another which in turn calls the first, with any number of calls in between. You must define such functions together in the one `let` binding, using the `and` keyword to link them together.

The following example shows two mutually recursive functions.

```
let rec Even x =
    if x = 0 then true
    else Odd (x-1)
and Odd x =
    if x = 0 then false
    else Even (x-1)
```

See also

- Functions

Entry Point

7/30/2019 • 2 minutes to read • Edit Online

This topic describes the method that you use to set the entry point to an F# program.

Syntax

```
[<EntryPoint>]
let-function-binding
```

Remarks

In the previous syntax, *let-function-binding* is the definition of a function in a `let` binding.

The entry point to a program that is compiled as an executable file is where execution formally starts. You specify the entry point to an F# application by applying the `EntryPoint` attribute to the program's `main` function. This function (created by using a `let` binding) must be the last function in the last compiled file. The last compiled file is the last file in the project or the last file that is passed to the command line.

The entry point function has type `string array -> int`. The arguments provided on the command line are passed to the `main` function in the array of strings. The first element of the array is the first argument; the name of the executable file is not included in the array, as it is in some other languages. The return value is used as the exit code for the process. Zero usually indicates success; nonzero values indicate an error. There is no convention for the specific meaning of nonzero return codes; the meanings of the return codes are application-specific.

The following example illustrates a simple `main` function.

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    // Return 0. This indicates success.
    0
```

When this code is executed with the command line `EntryPoint.exe 1 2 3`, the output is as follows.

```
Arguments passed to function : [|"1"; "2"; "3"|]
```

Implicit Entry Point

When a program has no **EntryPoint** attribute that explicitly indicates the entry point, the top level bindings in the last file to be compiled are used as the entry point.

See also

- Functions
- let Bindings

External Functions

8/22/2019 • 2 minutes to read • Edit Online

This topic describes F# language support for calling functions in native code.

Syntax

```
[<DllImport( arguments )>]
extern declaration
```

Remarks

In the previous syntax, *arguments* represents arguments that are supplied to the `System.Runtime.InteropServices.DllImportAttribute` attribute. The first argument is a string that represents the name of the DLL that contains this function, without the .dll extension. Additional arguments can be supplied for any of the public properties of the `System.Runtime.InteropServices.DllImportAttribute` class, such as the calling convention.

Assume you have a native C++ DLL that contains the following exported function.

```
#include <stdio.h>
extern "C" void __declspec(dllexport) HelloWorld()
{
    printf("Hello world, invoked by F#!\n");
}
```

You can call this function from F# by using the following code.

```
open System.Runtime.InteropServices

module InteropWithNative =
    [<DllImport(@"C:\bin\natived11", CallingConvention = CallingConvention.Cdecl)>]
    extern void HelloWorld()

InteropWithNative.HelloWorld()
```

Interoperability with native code is referred to as *platform invoke* and is a feature of the CLR. For more information, see Interoperating with Unmanaged Code. The information in that section is applicable to F#.

See also

- Functions

Inline Functions

7/30/2019 • 2 minutes to read • Edit Online

Inline functions are functions that are integrated directly into the calling code.

Using Inline Functions

When you use static type parameters, any functions that are parameterized by type parameters must be inline. This guarantees that the compiler can resolve these type parameters. When you use ordinary generic type parameters, there is no such restriction.

Other than enabling the use of member constraints, inline functions can be helpful in optimizing code. However, overuse of inline functions can cause your code to be less resistant to changes in compiler optimizations and the implementation of library functions. For this reason, you should avoid using inline functions for optimization unless you have tried all other optimization techniques. Making a function or method inline can sometimes improve performance, but that is not always the case. Therefore, you should also use performance measurements to verify that making any given function inline does in fact have a positive effect.

The `inline` modifier can be applied to functions at the top level, at the module level, or at the method level in a class.

The following code example illustrates an inline function at the top level, an inline instance method, and an inline static method.

```
let inline increment x = x + 1
type WrapInt32() =
    member inline this.incrementByOne(x) = x + 1
    static member inline Increment(x) = x + 1
```

Inline Functions and Type Inference

The presence of `inline` affects type inference. This is because inline functions can have statically resolved type parameters, whereas non-inline functions cannot. The following code example shows a case where `inline` is helpful because you are using a function that has a statically resolved type parameter, the `float` conversion operator.

```
let inline printAsFloatingPoint number =
    printfn "%f" (float number)
```

Without the `inline` modifier, type inference forces the function to take a specific type, in this case `int`. But with the `inline` modifier, the function is also inferred to have a statically resolved type parameter. With the `inline` modifier, the type is inferred to be the following:

```
^a -> unit when ^a : (static member op_Explicit : ^a -> float)
```

This means that the function accepts any type that supports a conversion to **float**.

See also

- Functions
- Constraints
- Statically Resolved Type Parameters

Values

7/30/2019 • 3 minutes to read • Edit Online

Values in F# are quantities that have a specific type; values can be integral or floating point numbers, characters or text, lists, sequences, arrays, tuples, discriminated unions, records, class types, or function values.

Binding a Value

The term *binding* means associating a name with a definition. The `let` keyword binds a value, as in the following examples:

```
let a = 1
let b = 100u
let str = "text"

// A function value binding.

let f x = x + 1
```

The type of a value is inferred from the definition. For a primitive type, such as an integral or floating point number, the type is determined from the type of the literal. Therefore, in the previous example, the compiler infers the type of `b` to be `unsigned int`, whereas the compiler infers the type of `a` to be `int`. The type of a function value is determined from the return value in the function body. For more information about function value types, see Functions. For more information about literal types, see Literals.

The compiler does not issue diagnostics about unused bindings by default. To receive these messages, enable warning 1182 in your project file or when invoking the compiler (see `--warnon` under Compiler Options).

Why Immutable?

Immutable values are values that cannot be changed throughout the course of a program's execution. If you are used to languages such as C++, Visual Basic, or C#, you might find it surprising that F# puts primacy over immutable values rather than variables that can be assigned new values during the execution of a program. Immutable data is an important element of functional programming. In a multithreaded environment, shared mutable variables that can be changed by many different threads are difficult to manage. Also, with mutable variables, it can sometimes be hard to tell if a variable might be changed when it is passed to another function.

In pure functional languages, there are no variables, and functions behave strictly as mathematical functions. Where code in a procedural language uses a variable assignment to alter a value, the equivalent code in a functional language has an immutable value that is the input, an immutable function, and different immutable values as the output. This mathematical strictness allows for tighter reasoning about the behavior of the program. This tighter reasoning is what enables compilers to check code more stringently and to optimize more effectively, and helps make it easier for developers to understand and write correct code. Functional code is therefore likely to be easier to debug than ordinary procedural code.

F# is not a pure functional language, yet it fully supports functional programming. Using immutable values is a good practice because doing this allows your code to benefit from an important aspect of functional programming.

Mutable Variables

You can use the keyword `mutable` to specify a variable that can be changed. Mutable variables in F# should generally have a limited scope, either as a field of a type or as a local value. Mutable variables with a limited scope are easier to control and are less likely to be modified in incorrect ways.

You can assign an initial value to a mutable variable by using the `let` keyword in the same way as you would define a value. However, the difference is that you can subsequently assign new values to mutable variables by using the `<-` operator, as in the following example.

```
let mutable x = 1
x <- x + 1
```

Values marked `mutable` may be automatically promoted to `'a ref` if captured by a closure, including forms that create closures, such as `seq` builders. If you wish to be notified when this occurs, enable warning 3180 in your project file or when invoking the compiler.

Related Topics

TITLE	DESCRIPTION
let Bindings	Provides information about using the `let` keyword to bind names to values and functions.
Functions	Provides an overview of functions in F#.

See also

- Null Values
- F# Language Reference

Null Values

7/30/2019 • 3 minutes to read • Edit Online

This topic describes how the null value is used in F#.

Null Value

The null value is not normally used in F# for values or variables. However, null appears as an abnormal value in certain situations. If a type is defined in F#, null is not permitted as a regular value unless the AllowNullLiteral attribute is applied to the type. If a type is defined in some other .NET language, null is a possible value, and when you are interoperating with such types, your F# code might encounter null values.

For a type defined in F# and used strictly from F#, the only way to create a null value using the F# library directly is to use Unchecked.defaultof or Array.zeroCreate. However, for an F# type that is used from other .NET languages, or if you are using that type with an API that is not written in F#, such as the .NET Framework, null values can occur.

You can use the `option` type in F# when you might use a reference variable with a possible null value in another .NET language. Instead of null, with an F# `option` type, you use the option value `None` if there is no object. You use the option value `Some(obj)` with an object `obj` when there is an object. For more information, see Options. Note that you can still pack a `null` value into an Option if, for `Some x`, `x` happens to be `null`. Because of this, it is important you use `None` when a value is `null`.

The `null` keyword is a valid keyword in the F# language, and you have to use it when you are working with .NET Framework APIs or other APIs that are written in another .NET language. The two situations in which you might need a null value are when you call a .NET API and pass a null value as an argument, and when you interpret the return value or an output parameter from a .NET method call.

To pass a null value to a .NET method, just use the `null` keyword in the calling code. The following code example illustrates this.

```
open System

// Pass a null value to a .NET method.
let ParseDateTime (str: string) =
    let (success, res) = DateTime.TryParse(str, null, System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None
```

To interpret a null value that is obtained from a .NET method, use pattern matching if you can. The following code example shows how to use pattern matching to interpret the null value that is returned from `ReadLine` when it tries to read past the end of an input stream.

```
// Open a file and create a stream reader.
let fileStream1 =
    try
        System.IO.File.OpenRead("TextFile1.txt")
    with
        | :? System.IO.FileNotFoundException -> printfn "Error: TextFile1.txt not found."; exit(1)

let streamReader = new System.IO.StreamReader(fileStream1)

// ProcessNextLine returns false when there is no more input;
// it returns true when there is more input.
let ProcessNextLine nextLine =
    match nextLine with
    | null -> false
    | inputString ->
        match ParseDateTime inputString with
        | Some(date) -> printfn "%s" (date.ToLocalTime().ToString())
        | None -> printfn "Failed to parse the input."
        true

// A null value returned from .NET method ReadLine when there is
// no more input.
while ProcessNextLine (streamReader.ReadLine()) do ()
```

Null values for F# types can also be generated in other ways, such as when you use `Array.zeroCreate`, which calls `Unchecked.defaultof`. You must be careful with such code to keep the null values encapsulated. In a library intended only for F#, you do not have to check for null values in every function. If you are writing a library for interoperation with other .NET languages, you might have to add checks for null input parameters and throw an `ArgumentNullException`, just as you do in C# or Visual Basic code.

You can use the following code to check if an arbitrary value is null.

```
match box value with
| null -> printf "The value is null."
| _ -> printf "The value is not null."
```

See also

- Values
- Match Expressions

Literals

7/9/2019 • 2 minutes to read • Edit Online

> **NOTE**
> The API reference links in this article will take you to MSDN (for now).

This topic provides a table that shows how to specify the type of a literal in F#.

Literal Types

The following table shows the literal types in F#. Characters that represent digits in hexadecimal notation are not case-sensitive; characters that identify the type are case-sensitive.

TYPE	DESCRIPTION	SUFFIX OR PREFIX	EXAMPLES
sbyte	signed 8-bit integer	y	`86y` `0b00000101y`
byte	unsigned 8-bit natural number	uy	`86uy` `0b00000101uy`
int16	signed 16-bit integer	s	`86s`
uint16	unsigned 16-bit natural number	us	`86us`
int int32	signed 32-bit integer	l or none	`86` `86l`
uint uint32	unsigned 32-bit natural number	u or ul	`86u` `86ul`
nativeint	native pointer to a signed natural number	n	`123n`
unativeint	native pointer as an unsigned natural number	un	`0x00002D3Fun`
int64	signed 64-bit integer	L	`86L`
uint64	unsigned 64-bit natural number	UL	`86UL`
single, float32	32-bit floating point number	F or f	`4.14F` or `4.14f`

TYPE	DESCRIPTION	SUFFIX OR PREFIX	EXAMPLES
		lf	`0x000000001f`
float; double	64-bit floating point number	none	`4.14` or `2.3E+32` or `2.3e+32`
		LF	`0x0000000000000000LF`
bigint	integer not limited to 64-bit representation	I	`99999999999999999999999999999I`
decimal	fractional number represented as a fixed point or rational number	M or m	`0.7833M` or `0.7833m`
Char	Unicode character	none	`'a'`

String	Unicode string	none	`"text\n"`
			or
			`@"c:\filename"`
			or
			`"""<book title="Paradise Lost">"""`
			or
			`"string1" + "string2"`
			See also Strings.

byte	ASCII character	B	`'a'B`
byte[]	ASCII string	B	`"text"B`
String or byte[]	verbatim string	@ prefix	`@"\\server\share"` (Unicode) `@"\\server\share"B` (ASCII)

Named literals

Values that are intended to be constants can be marked with the Literal attribute. This attribute has the effect of causing a value to be compiled as a constant.

In pattern matching expressions, identifiers that begin with lowercase characters are always treated as variables to be bound, rather than as literals, so you should generally use initial capitals when you define literals.

```
[<Literal>]
let SomeJson = """{"numbers":[1,2,3,4,5]}"""

[<Literal>]
let Literal1 = "a" + "b"

[<Literal>]
let FileLocation =   __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let Literal2 = 1 ||| 64

[<Literal>]
let Literal3 = System.IO.FileAccess.Read ||| System.IO.FileAccess.Write
```

Remarks

Unicode strings can contain explicit encodings that you can specify by using `\u` followed by a 16-bit hexadecimal code (0000 - FFFF), or UTF-32 encodings that you can specify by using `\U` followed by a 32-bit hexadecimal code that represents any Unicode code point (00000000 - 0010FFFF).

The use of other bitwise operators other than `|||` isn't allowed.

Integers in other bases

Signed 32-bit integers can also be specified in hexadecimal, octal, or binary using a `0x`, `0o` or `0b` prefix respectively.

```
let numbers = (0x9F, 0o77, 0b1010)
// Result: numbers : int * int * int = (159, 63, 10)
```

Underscores in numeric literals

You can separate digits with the underscore character (`_`).

```
let value = 0xDEAD_BEEF

let valueAsBits = 0b1101_1110_1010_1101_1011_1110_1110_1111

let exampleSSN = 123_456_7890
```

See also

- Core.LiteralAttribute Class

F# Types

8/2/2019 • 4 minutes to read • Edit Online

This topic describes the types that are used in F# and how F# types are named and described.

Summary of F# Types

Some types are considered *primitive types*, such as the Boolean type `bool` and integral and floating point types of various sizes, which include types for bytes and characters. These types are described in Primitive Types.

Other types that are built into the language include tuples, lists, arrays, sequences, records, and discriminated unions. If you have experience with other .NET languages and are learning F#, you should read the topics for each of these types. Links to more information about these types are included in the Related Topics section of this topic. These F#-specific types support styles of programming that are common to functional programming languages. Many of these types have associated modules in the F# library that support common operations on these types.

The type of a function includes information about the parameter types and return type.

The .NET Framework is the source of object types, interface types, delegate types, and others. You can define your own object types just as you can in any other .NET language.

Also, F# code can define aliases, which are named *type abbreviations*, that are alternative names for types. You might use type abbreviations when the type might change in the future and you want to avoid changing the code that depends on the type. Or, you might use a type abbreviation as a friendly name for a type that can make code easier to read and understand.

F# provides useful collection types that are designed with functional programming in mind. Using these collection types helps you write code that is more functional in style. For more information, see F# Collection Types.

Syntax for Types

In F# code, you often have to write out the names of types. Every type has a syntactic form, and you use these syntactic forms in type annotations, abstract method declarations, delegate declarations, signatures, and other constructs. Whenever you declare a new program construct in the interpreter, the interpreter prints the name of the construct and the syntax for its type. This syntax might be just an identifier for a user-defined type or a built-in identifier such as for `int` or `string`, but for more complex types, the syntax is more complex.

The following table shows aspects of the type syntax for F# types.

TYPE	TYPE SYNTAX	EXAMPLES

primitive type	*type-name*	`int`
		`float`
		`string`

aggregate type (class, structure, union, record, enum, and so on)	*type-name*	`System.DateTime` `Color`

TYPE	TYPE SYNTAX	EXAMPLES
type abbreviation	*type-abbreviation-name*	`bigint`
fully qualified type	*namespaces.type-name* or *modules.type-name* or *namespaces.modules.type-name*	`System.IO.StreamWriter`

array	*type-name*[] or
	
	type-name array
		`int[]`
		
		`array<int>`
		
		`int array`

two-dimensional array	*type-name*[,]	`int[,]`
		
		`float[,]`

three-dimensional array	*type-name*[,,]	`float[,,]`
tuple	*type-name1* * *type-name2* ...	For example, `(1,'b',3)` has type `int * char * int`

generic type	*type-parameter generic-type-name*
	
	or
	
	generic-type-name<*type-parameter-list*>
		`'a list`
		
		`list<'a>`
		
		`Dictionary<'key, 'value>`

constructed type (a generic type that has a specific type argument supplied)	*type-argument generic-type-name*
	
	or
	
	generic-type-name<*type-argument-list*>
		`int option`
		
		`string list`
		
		`int ref`
		
		`option<int>`
		
		`list<string>`
		
		`ref<int>`
		
		`Dictionary<int, string>`

function type that has a single parameter	*parameter-type1* -> *return-type*	A function that takes an `int` and returns a `string` has type `int -> string`
function type that has multiple parameters	*parameter-type1* -> *parameter-type2* -> ... -> *return-type*	A function that takes an `int` and a `float` and returns a `string` has type `int -> float -> string`

TYPE	TYPE SYNTAX	EXAMPLES
higher order function as a parameter	(*function-type*)	`List.map` has type `('a -> 'b) -> 'a list -> 'b list`
delegate	delegate of *function-type*	`delegate of unit -> int`
flexible type	*#type-name*	`#System.Windows.Forms.Control` `#seq<int>`

Related Topics

TOPIC	DESCRIPTION
Primitive Types	Describes built-in simple types such as integral types, the Boolean type, and character types.
Unit Type	Describes the `unit` type, a type that has one value and that is indicated by (); equivalent to `void` in C# and `Nothing` in Visual Basic.
Tuples	Describes the tuple type, a type that consists of associated values of any type grouped in pairs, triples, quadruples, and so on.
Options	Describes the option type, a type that may either have a value or be empty.
Lists	Describes lists, which are ordered, immutable series of elements all of the same type.
Arrays	Describes arrays, which are ordered sets of mutable elements of the same type that occupy a contiguous block of memory and are of fixed size.
Sequences	Describes the sequence type, which represents a logical series of values; individual values are computed only as necessary.
Records	Describes the record type, a small aggregate of named values.
Discriminated Unions	Describes the discriminated union type, a type whose values can be any one of a set of possible types.
Functions	Describes function values.
Classes	Describes the class type, an object type that corresponds to a .NET reference type. Class types can contain members, properties, implemented interfaces, and a base type.
Structures	Describes the `struct` type, an object type that corresponds to a .NET value type. The `struct` type usually represents a small aggregate of data.

TOPIC	DESCRIPTION
Interfaces	Describes interface types, which are types that represent a set of members that provide certain functionality but that contain no data. An interface type must be implemented by an object type to be useful.
Delegates	Describes the delegate type, which represents a function as an object.
Enumerations	Describes enumeration types, whose values belong to a set of named values.
Attributes	Describes attributes, which are used to specify metadata for another type.
Exception Types	Describes exceptions, which specify error information.

Type Inference

7/30/2019 • 2 minutes to read • Edit Online

This topic describes how the F# compiler infers the types of values, variables, parameters and return values.

Type Inference in General

The idea of type inference is that you do not have to specify the types of F# constructs except when the compiler cannot conclusively deduce the type. Omitting explicit type information does not mean that F# is a dynamically typed language or that values in F# are weakly typed. F# is a statically typed language, which means that the compiler deduces an exact type for each construct during compilation. If there is not enough information for the compiler to deduce the types of each construct, you must supply additional type information, typically by adding explicit type annotations somewhere in the code.

Inference of Parameter and Return Types

In a parameter list, you do not have to specify the type of each parameter. And yet, F# is a statically typed language, and therefore every value and expression has a definite type at compile time. For those types that you do not specify explicitly, the compiler infers the type based on the context. If the type is not otherwise specified, it is inferred to be generic. If the code uses a value inconsistently, in such a way that there is no single inferred type that satisfies all the uses of a value, the compiler reports an error.

The return type of a function is determined by the type of the last expression in the function.

For example, in the following code, the parameter types `a` and `b` and the return type are all inferred to be `int` because the literal `100` is of type `int`.

```
let f a b = a + b + 100
```

You can influence type inference by changing the literals. If you make the `100` a `uint32` by appending the suffix `u`, the types of `a`, `b`, and the return value are inferred to be `uint32`.

You can also influence type inference by using other constructs that imply restrictions on the type, such as functions and methods that work with only a particular type.

Also, you can apply explicit type annotations to function or method parameters or to variables in expressions, as shown in the following examples. Errors result if conflicts occur between different constraints.

```
// Type annotations on a parameter.
let addu1 (x : uint32) y =
    x + y

// Type annotations on an expression.
let addu2 x y =
    (x : uint32) + y
```

You can also explicitly specify the return value of a function by providing a type annotation after all the parameters.

```
let addu1 x y : uint32 =
   x + y
```

A common case where a type annotation is useful on a parameter is when the parameter is an object type and you want to use a member.

```
let replace(str: string) =
    str.Replace("A", "a")
```

Automatic Generalization

If the function code is not dependent on the type of a parameter, the compiler considers the parameter to be generic. This is called *automatic generalization*, and it can be a powerful aid to writing generic code without increasing complexity.

For example, the following function combines two parameters of any type into a tuple.

```
let makeTuple a b = (a, b)
```

The type is inferred to be

```
'a -> 'b -> 'a * 'b
```

Additional Information

Type inference is described in more detail in the F# Language Specification.

See also

- Automatic Generalization

Basic types

5/15/2019 • 2 minutes to read • Edit Online

This topic lists the basic types that are defined in the F# language. These types are the most fundamental in F#, forming the basis of nearly every F# program. They are a superset of .NET primitive types.

TYPE	.NET TYPE	DESCRIPTION
`bool`	Boolean	Possible values are `true` and `false`.
`byte`	Byte	Values from 0 to 255.
`sbyte`	SByte	Values from -128 to 127.
`int16`	Int16	Values from -32768 to 32767.
`uint16`	UInt16	Values from 0 to 65535.
`int`	Int32	Values from -2,147,483,648 to 2,147,483,647.
`uint32`	UInt32	Values from 0 to 4,294,967,295.
`int64`	Int64	Values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
`uint64`	UInt64	Values from 0 to 18,446,744,073,709,551,615.
`nativeint`	IntPtr	A native pointer as a signed integer.
`unativeint`	UIntPtr	A native pointer as an unsigned integer.
`char`	Char	Unicode character values.
`string`	String	Unicode text.
`decimal`	Decimal	A floating point data type that has at least 28 significant digits.
`unit`	not applicable	Indicates the absence of an actual value. The type has only one formal value, which is denoted `()`. The unit value, `()`, is often used as a placeholder where a value is needed but no real value is available or makes sense.
`void`	Void	Indicates no type or value.

TYPE	.NET TYPE	DESCRIPTION
`float32`, `single`	Single	A 32-bit floating point type.
`float`, `double`	Double	A 64-bit floating point type.

NOTE

You can perform computations with integers too big for the 64-bit integer type by using the bigint type. `bigint` is not considered a basic type; it is an abbreviation for `System.Numerics.BigInteger`.

See also

- F# Language Reference

Unit Type

7/30/2019 • 2 minutes to read • Edit Online

The `unit` type is a type that indicates the absence of a specific value; the `unit` type has only a single value, which acts as a placeholder when no other value exists or is needed.

Syntax

```
// The value of the unit type.
()
```

Remarks

Every F# expression must evaluate to a value. For expressions that do not generate a value that is of interest, the value of type `unit` is used. The `unit` type resembles the `void` type in languages such as C# and C++.

The `unit` type has a single value, and that value is indicated by the token `()`.

The value of the `unit` type is often used in F# programming to hold the place where a value is required by the language syntax, but when no value is needed or desired. An example might be the return value of a `printf` function. Because the important actions of the `printf` operation occur in the function, the function does not have to return an actual value. Therefore, the return value is of type `unit`.

Some constructs expect a `unit` value. For example, a `do` binding or any code at the top level of a module is expected to evaluate to a `unit` value. The compiler reports a warning when a `do` binding or code at the top level of a module produces a result other than the `unit` value that is not used, as shown in the following example.

```
let function1 x y = x + y
// The next line results in a compiler warning.
function1 10 20
// Changing the code to one of the following eliminates the warning.
// Use this when you do want the return value.
let result = function1 10 20
// Use this if you are only calling the function for its side effects,
// and do not want the return value.
function1 10 20 |> ignore
```

This warning is a characteristic of functional programming; it does not appear in other .NET programming languages. In a purely functional program, in which functions do not have any side effects, the final return value is the only result of a function call. Therefore, when the result is ignored, it is a possible programming error. Although F# is not a purely functional programming language, it is a good practice to follow functional programming style whenever possible.

See also

- Primitive
- F# Language Reference

Strings

7/30/2019 • 4 minutes to read • Edit Online

NOTE

The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

The `string` type represents immutable text as a sequence of Unicode characters. `string` is an alias for `System.String` in the .NET Framework.

Remarks

String literals are delimited by the quotation mark (") character. The backslash character (\ ) is used to encode certain special characters. The backslash and the next character together are known as an *escape sequence*. Escape sequences supported in F# string literals are shown in the following table.

CHARACTER	ESCAPE SEQUENCE
Alert	`\a`
Backspace	`\b`
Form feed	`\f`
Newline	`\n`
Carriage return	`\r`
Tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Quotation mark	`\"`
Apostrophe	`\'`
Unicode character	`\DDD` (where `D` indicates a decimal digit; range of 000 - 255; for example, `\231` = "ç")
Unicode character	`\xHH` (where `H` indicates a hexadecimal digit; range of 00 - FF; for example, `\xE7` = "ç")
Unicode character	`\uHHHH` (UTF-16) (where `H` indicates a hexadecimal digit; range of 0000 - FFFF; for example, `\u00E7` = "ç")

CHARACTER	ESCAPE SEQUENCE
Unicode character	`\U00HHHHHH` (UTF-32) (where `H` indicates a hexadecimal digit; range of 000000 - 10FFFF; for example, `\U0001F47D` = "🙽")

IMPORTANT

The `\DDD` escape sequence is decimal notation, not octal notation like in most other languages. Therefore, digits `8` and `9` are valid, and a sequence of `\032` represents a space (U+0020), whereas that same code point in octal notation would be `\040`.

NOTE

Being constrained to a range of 0 - 255 (0xFF), the `\DDD` and `\x` escape sequences are effectively the ISO-8859-1 character set, since that matches the first 256 Unicode code points.

If preceded by the @ symbol, the literal is a verbatim string. This means that any escape sequences are ignored, except that two quotation mark characters are interpreted as one quotation mark character.

Additionally, a string may be enclosed by triple quotes. In this case, all escape sequences are ignored, including double quotation mark characters. To specify a string that contains an embedded quoted string, you can either use a verbatim string or a triple-quoted string. If you use a verbatim string, you must specify two quotation mark characters to indicate a single quotation mark character. If you use a triple-quoted string, you can use the single quotation mark characters without them being parsed as the end of the string. This technique can be useful when you work with XML or other structures that include embedded quotation marks.

```
// Using a verbatim string
let xmlFragment1 = @"<book author=""Milton, John"" title=""Paradise Lost"">"

// Using a triple-quoted string
let xmlFragment2 = """<book author="Milton, John" title="Paradise Lost">"""
```

In code, strings that have line breaks are accepted and the line breaks are interpreted literally as newlines, unless a backslash character is the last character before the line break. Leading white space on the next line is ignored when the backslash character is used. The following code produces a string `str1` that has value `"abc\ndef"` and a string `str2` that has value `"abcdef"`.

```
let str1 = "abc
def"
let str2 = "abc\
def"
```

You can access individual characters in a string by using array-like syntax, as follows.

```
printfn "%c" str1.[1]
```

The output is `b`.

Or you can extract substrings by using array slice syntax, as shown in the following code.

```
printfn "%s" (str1.[0..2])
printfn "%s" (str2.[3..5])
```

The output is as follows.

```
abc
def
```

You can represent ASCII strings by arrays of unsigned bytes, type `byte[]`. You add the suffix `B` to a string literal to indicate that it is an ASCII string. ASCII string literals used with byte arrays support the same escape sequences as Unicode strings, except for the Unicode escape sequences.

```
// "abc" interpreted as a Unicode string.
let str1 : string = "abc"
// "abc" interpreted as an ASCII byte array.
let bytearray : byte[] = "abc"B
```

String Operators

There are two ways to concatenate strings: by using the `+` operator or by using the `^` operator. The `+` operator maintains compatibility with the .NET Framework string handling features.

The following example illustrates string concatenation.

```
let string1 = "Hello, " + "world"
```

String Class

Because the string type in F# is actually a .NET Framework `System.String` type, all the `System.String` members are available. This includes the `+` operator, which is used to concatenate strings, the `Length` property, and the `Chars` property, which returns the string as an array of Unicode characters. For more information about strings, see `System.String`.

By using the `Chars` property of `System.String`, you can access the individual characters in a string by specifying an index, as is shown in the following code.

```
let printChar (str : string) (index : int) =
    printfn "First character: %c" (str.Chars(index))
```

String Module

Additional functionality for string handling is included in the `String` module in the `FSharp.Core` namespace. For more information, see Core.String Module.

See also

- F# Language Reference

Tuples

7/30/2019 • 5 minutes to read • Edit Online

A *tuple* is a grouping of unnamed but ordered values, possibly of different types. Tuples can either be reference types or structs.

Syntax

```
(element, ... , element)
struct(element, ... ,element )
```

Remarks

Each *element* in the previous syntax can be any valid F# expression.

Examples

Examples of tuples include pairs, triples, and so on, of the same or different types. Some examples are illustrated in the following code.

```
(1, 2)

// Triple of strings.
("one", "two", "three")

// Tuple of generic types.
(a, b)

// Tuple that has mixed types.
("one", 1, 2.0)

// Tuple of integer expressions.
(a + 1, b + 1)

// Struct Tuple of floats
struct (1.025f, 1.5f)
```

Obtaining Individual Values

You can use pattern matching to access and assign names for tuple elements, as shown in the following code.

```
let print tuple1 =
   match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

You can also deconstruct a tuple via pattern matching outside of a `match` expression via `let` binding:

```
let (a, b) = (1, 2)

// Or as a struct
let struct (c, d) = struct (1, 2)
```

Or you can pattern match on tuples as inputs to functions:

```
let getDistance ((x1,y1): float*float) ((x2,y2): float*float) =
    // Note the ability to work on individual elements
    (x1*x2 - y1*y2)
    |> abs
    |> sqrt
```

If you need only one element of the tuple, the wildcard character (the underscore) can be used to avoid creating a new name for a value that you do not need.

```
let (a, _) = (1, 2)
```

Copying elements from a reference tuple into a struct tuple is also simple:

```
// Create a reference tuple
let (a, b) = (1, 2)

// Construct a struct tuple from it
let struct (c, d) = struct (a, b)
```

The functions `fst` and `snd` (reference tuples only) return the first and second elements of a tuple, respectively.

```
let c = fst (1, 2)
let d = snd (1, 2)
```

There is no built-in function that returns the third element of a triple, but you can easily write one as follows.

```
let third (_, _, c) = c
```

Generally, it is better to use pattern matching to access individual tuple elements.

Using Tuples

Tuples provide a convenient way to return multiple values from a function, as shown in the following example. This example performs integer division and returns the rounded result of the operation as a first member of a tuple pair and the remainder as a second member of the pair.

```
let divRem a b =
   let x = a / b
   let y = a % b
   (x, y)
```

Tuples can also be used as function arguments when you want to avoid the implicit currying of function arguments that is implied by the usual function syntax.

```
let sumNoCurry (a, b) = a + b
```

The usual syntax for defining the function `let sum a b = a + b` enables you to define a function that is the partial application of the first argument of the function, as shown in the following code.

```
let sum a b = a + b

let addTen = sum 10
let result = addTen 95
// Result is 105.
```

Using a tuple as the parameter disables currying. For more information, see "Partial Application of Arguments" in Functions.

Names of Tuple Types

When you write out the name of a type that is a tuple, you use the `*` symbol to separate elements. For a tuple that consists of an `int`, a `float`, and a `string`, such as `(10, 10.0, "ten")`, the type would be written as follows.

```
int * float * string
```

Interoperation with C# Tuples

C# 7.0 introduced tuples to the language. Tuples in C# are structs, and are equivalent to struct tuples in F#. If you need to interoperate with C#, you must use struct tuples.

This is easy to do. For example, imagine you have to pass a tuple to a C# class and then consume its result, which is also a tuple:

```
namespace CSharpTupleInterop
{
    public static class Example
    {
        public static (int, int) AddOneToXAndY((int x, int y) a) =>
            (a.x + 1, a.y + 1);
    }
}
```

In your F# code, you can then pass a struct tuple as the parameter and consume the result as a struct tuple.

```
open TupleInterop

let struct (newX, newY) = Example.AddOneToXAndY(struct (1, 2))
// newX is now 2, and newY is now 3
```

Converting between Reference Tuples and Struct Tuples

Because Reference Tuples and Struct Tuples have a completely different underlying representation, they are not implicitly convertible. That is, code such as the following won't compile:

```
// Will not compile!
let (a, b) = struct (1, 2)

// Will not compile!
let struct (c, d) = (1, 2)

// Won't compile!
let f(t: struct(int*int)): int*int = t
```

You must pattern match on one tuple and construct the other with the constituent parts. For example:

```
// Pattern match on the result.
let (a, b) = (1, 2)

// Construct a new tuple from the parts you pattern matched on.
let struct (c, d) = struct (a, b)
```

Compiled Form of Reference Tuples

This section explains the form of tuples when they're compiled. The information here isn't necessary to read unless you are targeting .NET Framework 3.5 or lower.

Tuples are compiled into objects of one of several generic types, all named `System.Tuple`, that are overloaded on the arity, or number of type parameters. Tuple types appear in this form when you view them from another language, such as C# or Visual Basic, or when you are using a tool that is not aware of F# constructs. The `Tuple` types were introduced in .NET Framework 4. If you are targeting an earlier version of the .NET Framework, the compiler uses versions of System.Tuple from the 2.0 version of the F# Core Library. The types in this library are used only for applications that target the 2.0, 3.0, and 3.5 versions of the .NET Framework. Type forwarding is used to ensure binary compatibility between .NET Framework 2.0 and .NET Framework 4 F# components.

Compiled Form of Struct Tuples

Struct tuples (for example, `struct (x, y)`), are fundamentally different from reference tuples. They are compiled into the ValueTuple type, overloaded by arity, or the number of type parameters. They are equivalent to C# 7.0 Tuples and Visual Basic 2017 Tuples, and interoperate bidirectionally.

See also

- F# Language Reference
- F# Types

F# Collection Types

6/13/2019 • 15 minutes to read • Edit Online

By reviewing this topic, you can determine which F# collection type best suits a particular need. These collection types differ from the collection types in the .NET Framework, such as those in the `System.Collections.Generic` namespace, in that the F# collection types are designed from a functional programming perspective rather than an object-oriented perspective. More specifically, only the array collection has mutable elements. Therefore, when you modify a collection, you create an instance of the modified collection instead of altering the original collection.

Collection types also differ in the type of data structure in which objects are stored. Data structures such as hash tables, linked lists, and arrays have different performance characteristics and a different set of available operations.

F# Collection Types

The following table shows F# collection types.

TYPE	DESCRIPTION	RELATED LINKS
List	An ordered, immutable series of elements of the same type. Implemented as a linked list.	Lists List Module
Array	A fixed-size, zero-based, mutable collection of consecutive data elements that are all of the same type.	Arrays Array Module Array2D Module Array3D Module
seq	A logical series of elements that are all of one type. Sequences are particularly useful when you have a large, ordered collection of data but don't necessarily expect to use all the elements. Individual sequence elements are computed only as required, so a sequence can perform better than a list if not all the elements are used. Sequences are represented by the `seq<'T>` type, which is an alias for `IEnumerable<T>`. Therefore, any .NET Framework type that implements `System.Collections.Generic.IEnumerable<'T>` can be used as a sequence.	Sequences Seq Module
Map	An immutable dictionary of elements. Elements are accessed by key.	Map Module
Set	An immutable set that's based on binary trees, where comparison is the F# structural comparison function, which potentially uses implementations of the `System.IComparable` interface on key values.	Set Module

Table of Functions

This section compares the functions that are available on F# collection types. The computational complexity of the function is given, where N is the size of the first collection, and M is the size of the second collection, if any. A dash (-) indicates that this function isn't available on the collection. Because sequences are lazily evaluated, a function such as Seq.distinct may be O(1) because it returns immediately, although it still affects the performance of the sequence when enumerated.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
append	O(M)	O(N)	O(N)	-	-	Returns a new collection that contains the elements of the first collection followed by elements of the second collection.
add	-	-	-	O(log N)	O(log N)	Returns a new collection with the element added.
average	O(N)	O(N)	O(N)	-	-	Returns the average of the elements in the collection.
averageBy	O(N)	O(N)	O(N)	-	-	Returns the average of the results of the provided function applied to each element.
blit	O(N)	-	-	-	-	Copies a section of an array.
cache	-	-	O(N)	-	-	Computes and stores elements of a sequence.
cast	-	-	O(N)	-	-	Converts the elements to the specified type.
choose	O(N)	O(N)	O(N)	-	-	Applies the given function `f` to each element `x` of the list. Returns the list that contains the results for each element where the function returns `Some(f(x))`.
collect	O(N)	O(N)	O(N)	-	-	Applies the given function to each element of the collection, concatenates all the results, and returns the combined list.
compareWith	-	-	O(N)	-	-	Compares two sequences by using the given comparison function, element by element.
concat	O(N)	O(N)	O(N)	-	-	Combines the given enumeration-of-enumerations as a single concatenated enumeration.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
contains	-	-	-	-	O(log N)	Returns true if the set contains the specified element.
containsKey	-	-	-	O(log N)	-	Tests whether an element is in the domain of a map.
count	-	-	-	-	O(N)	Returns the number of elements in the set.
countBy	-	-	O(N)	-	-	Applies a key-generating function to each element of a sequence, and returns a sequence that yields unique keys and their number of occurrences in the original sequence.
copy	O(N)	-	O(N)	-	-	Copies the collection.
create	O(N)	-	-	-	-	Creates an array of whole elements that are all initially the given value.
delay	-	-	O(1)	-	-	Returns a sequence that's built from the given delayed specification of a sequence.
difference	-	-	-	-	O(M * log N)	Returns a new set with the elements of the second set removed from the first set.
distinct			O(1)*			Returns a sequence that contains no duplicate entries according to generic hash and equality comparisons on the entries. If an element occurs multiple times in the sequence, later occurrences are discarded.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
distinctBy			O(1)*			Returns a sequence that contains no duplicate entries according to the generic hash and equality comparisons on the keys that the given key-generating function returns. If an element occurs multiple times in the sequence, later occurrences are discarded.
empty	O(1)	O(1)	O(1)	O(1)	O(1)	Creates an empty collection.
exists	O(N)	O(N)	O(N)	O(log N)	O(log N)	Tests whether any element of the sequence satisfies the given predicate.
exists2	O(min(N,M))	-	O(min(N,M))			Tests whether any pair of corresponding elements of the input sequences satisfies the given predicate.
fill	O(N)					Sets a range of elements of the array to the given value.
filter	O(N)	O(N)	O(N)	O(N)	O(N)	Returns a new collection that contains only the elements of the collection for which the given predicate returns `true`.
find	O(N)	O(N)	O(N)	O(log N)	-	Returns the first element for which the given function returns `true`. Returns `System.Collections.Gener` if no such element exists.
findIndex	O(N)	O(N)	O(N)	-	-	Returns the index of the first element in the array that satisfies the given predicate. Raises `System.Collections.Gener` if no element satisfies the predicate.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
findKey	-	-	-	O(log N)	-	Evaluates the function on each mapping in the collection, and returns the key for the first mapping where the function returns `true`. If no such element exists, this function raises `System.Collections.Gener`.
fold	O(N)	O(N)	O(N)	O(N)	O(N)	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f (... (f s i0)...) iN.
fold2	O(N)	O(N)	-	-	-	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, this function computes f (... (f s i0 j0)...) iN jN.
foldBack	O(N)	O(N)	-	O(N)	O(N)	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f i0 (...(f iN s)).

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
foldBack2	O(N)	O(N)	-	-	-	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, this function computes f i0 j0 (...(f iN jN s)).
forall	O(N)	O(N)	O(N)	O(N)	O(N)	Tests whether all elements of the collection satisfy the given predicate.
forall2	O(N)	O(N)	O(N)	-	-	Tests whether all corresponding elements of the collection satisfy the given predicate pairwise.
get / nth	O(1)	O(N)	O(N)	-	-	Returns an element from the collection given its index.
head	-	O(1)	O(1)	-	-	Returns the first element of the collection.
init	O(N)	O(N)	O(1)	-	-	Creates a collection given the dimension and a generator function to compute the elements.
initInfinite	-	-	O(1)	-	-	Generates a sequence that, when iterated, returns successive elements by calling the given function.
intersect	-	-	-	-	O(log N * log M)	Computes the intersection of two sets.
intersectMany	-	-	-	-	O(N1 * N2 ...)	Computes the intersection of a sequence of sets. The sequence must not be empty.
isEmpty	O(1)	O(1)	O(1)	O(1)	-	Returns `true` if the collection is empty.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
isProperSubset	-	-	-	-	O(M * log N)	Returns `true` if all elements of the first set are in the second set, and at least one element of the second set isn't in the first set.
isProperSuperset	-	-	-	-	O(M * log N)	Returns `true` if all elements of the second set are in the first set, and at least one element of the first set isn't in the second set.
isSubset	-	-	-	-	O(M * log N)	Returns `true` if all elements of the first set are in the second set.
isSuperset	-	-	-	-	O(M * log N)	Returns `true` if all elements of the second set are in the first set.
iter	O(N)	O(N)	O(N)	O(N)	O(N)	Applies the given function to each element of the collection.
iteri	O(N)	O(N)	O(N)	-	-	Applies the given function to each element of the collection. The integer that's passed to the function indicates the index of the element.
iteri2	O(N)	O(N)	-	-	-	Applies the given function to a pair of elements that are drawn from matching indices in two arrays. The integer that's passed to the function indicates the index of the elements. The two arrays must have the same length.
iter2	O(N)	O(N)	O(N)	-	-	Applies the given function to a pair of elements that are drawn from matching indices in two arrays. The two arrays must have the same length.
last	O(1)	O(N)	O(N)	-	-	Returns the last item in the applicable collection.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
length	O(1)	O(N)	O(N)	-	-	Returns the number of elements in the collection.
map	O(N)	O(N)	O(1)	-	-	Builds a collection whose elements are the results of applying the given function to each element of the array.
map2	O(N)	O(N)	O(1)	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise. The two input arrays must have the same length.
map3	-	O(N)	-	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the three collections simultaneously.
mapi	O(N)	O(N)	O(N)	-	-	Builds an array whose elements are the results of applying the given function to each element of the array. The integer index that's passed to the function indicates the index of the element that's being transformed.
mapi2	O(N)	O(N)	-	-	-	Builds a collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise, also passing the index of the elements. The two input arrays must have the same length.
max	O(N)	O(N)	O(N)	-	-	Returns the greatest element in the collection, compared by using the max operator.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
maxBy	O(N)	O(N)	O(N)	-	-	Returns the greatest element in the collection, compared by using max on the function result.
maxElement	-	-	-	-	O(log N)	Returns the greatest element in the set according to the ordering that's used for the set.
min	O(N)	O(N)	O(N)	-	-	Returns the least element in the collection, compared by using the min operator.
minBy	O(N)	O(N)	O(N)	-	-	Returns the least element in the collection, compared by using the min operator on the function result.
minElement	-	-	-	-	O(log N)	Returns the lowest element in the set according to the ordering that's used for the set.
ofArray	-	O(N)	O(1)	O(N)	O(N)	Creates a collection that contains the same elements as the given array.
ofList	O(N)	-	O(1)	O(N)	O(N)	Creates a collection that contains the same elements as the given list.
ofSeq	O(N)	O(N)	-	O(N)	O(N)	Creates a collection that contains the same elements as the given sequence.
pairwise	-	-	O(N)	-	-	Returns a sequence of each element in the input sequence and its predecessor except for the first element, which is returned only as the predecessor of the second element.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
partition	O(N)	O(N)	-	O(N)	O(N)	Splits the collection into two collections. The first collection contains the elements for which the given predicate returns `true`, and the second collection contains the elements for which the given predicate returns `false`.
permute	O(N)	O(N)	-	-	-	Returns an array with all elements permuted according to the specified permutation.
pick	O(N)	O(N)	O(N)	O(log N)	-	Applies the given function to successive elements, returning the first result where the function returns Some. If the function never returns Some, `System.Collections.Gener` is raised.
readonly	-	-	O(N)	-	-	Creates a sequence object that delegates to the given sequence object. This operation ensures that a type cast can't rediscover and mutate the original sequence. For example, if given an array, the returned sequence will return the elements of the array, but you can't cast the returned sequence object to an array.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
reduce	O(N)	O(N)	O(N)	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. This function starts by applying the function to the first two elements, passes this result into the function along with the third element, and so on. The function returns the final result.
reduceBack	O(N)	O(N)	-	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f i0 (...(f iN-1 iN)).
remove	-	-	-	O(log N)	O(log N)	Removes an element from the domain of the map. No exception is raised if the element isn't present.
replicate	-	O(N)	-	-	-	Creates a list of a specified length with every element set to the given value.
rev	O(N)	O(N)	-	-	-	Returns a new list with the elements in reverse order.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
scan	O(N)	O(N)	O(N)	-	-	Applies a function to each element of the collection, threading an accumulator argument through the computation. This operation applies the function to the second argument and the first element of the list. The operation then passes this result into the function along with the second element and so on. Finally, the operation returns the list of intermediate results and the final result.
scanBack	O(N)	O(N)	-	-	-	Resembles the foldBack operation but returns both the intermediate and final results.
singleton	-	-	O(1)	-	O(1)	Returns a sequence that yields only one item.
set	O(1)	-	-	-	-	Sets an element of an array to the specified value.
skip	-	-	O(N)	-	-	Returns a sequence that skips N elements of the underlying sequence and then yields the remaining elements of the sequence.
skipWhile	-	-	O(N)	-	-	Returns a sequence that, when iterated, skips elements of the underlying sequence while the given predicate returns `true` and then yields the remaining elements of the sequence.
sort	O(N log N) average O(N^2) worst case	O(N log N)	O(N log N)	-	-	Sorts the collection by element value. Elements are compared using compare.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
sortBy	O(N log N) average O(N^2) worst case	O(N log N)	O(N log N)	-	-	Sorts the given list by using keys that the given projection provides. Keys are compared using compare.
sortInPlace	O(N log N) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given comparison function. Elements are compared by using compare.
sortInPlaceBy	O(N log N) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given projection for the keys. Elements are compared by using compare.
sortInPlaceWith	O(N log N) average O(N^2) worst case	-	-	-	-	Sorts the elements of an array by mutating it in place and using the given comparison function as the order.
sortWith	O(N log N) average O(N^2) worst case	O(N log N)	-	-	-	Sorts the elements of a collection, using the given comparison function as the order and returning a new collection.
sub	O(N)	-	-	-	-	Builds an array that contains the given subrange that's specified by starting index and length.
sum	O(N)	O(N)	O(N)	-	-	Returns the sum of the elements in the collection.
sumBy	O(N)	O(N)	O(N)	-	-	Returns the sum of the results that are generated by applying the function to each element of the collection.
tail	-	O(1)	-	-	-	Returns the list without its first element.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
take	-	-	O(N)	-	-	Returns the elements of the sequence up to a specified count.
takeWhile	-	-	O(1)	-	-	Returns a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns `true` and then returns no more elements.
toArray	-	O(N)	O(N)	O(N)	O(N)	Creates an array from the given collection.
toList	O(N)	-	O(N)	O(N)	O(N)	Creates a list from the given collection.
toSeq	O(1)	O(1)	-	O(1)	O(1)	Creates a sequence from the given collection.
truncate	-	-	O(1)	-	-	Returns a sequence that, when enumerated, returns no more than N elements.
tryFind	O(N)	O(N)	O(N)	O(log N)	-	Searches for an element that satisfies a given predicate.
tryFindIndex	O(N)	O(N)	O(N)	-	-	Searches for the first element that satisfies a given predicate and returns the index of the matching element, or `None` if no such element exists.
tryFindKey	-	-	-	O(log N)	-	Returns the key of the first mapping in the collection that satisfies the given predicate, or returns `None` if no such element exists.
tryPick	O(N)	O(N)	O(N)	O(log N)	-	Applies the given function to successive elements, returning the first result where the function returns `Some` for some value. If no such element exists, the operation returns `None`.

FUNCTION	ARRAY	LIST	SEQUENCE	MAP	SET	DESCRIPTION
unfold	-	-	O(N)	-	-	Returns a sequence that contains the elements that the given computation generates.
union	-	-	-	-	O(M * log N)	Computes the union of the two sets.
unionMany	-	-	-	-	O(N1 * N2 ...)	Computes the union of a sequence of sets.
unzip	O(N)	O(N)	O(N)	-	-	Splits a list of pairs into two lists.
unzip3	O(N)	O(N)	O(N)	-	-	Splits a list of triples into three lists.
windowed	-	-	O(N)	-	-	Returns a sequence that yields sliding windows of containing elements that are drawn from the input sequence. Each window is returned as a fresh array.
zip	O(N)	O(N)	O(N)	-	-	Combines the two collections into a list of pairs. The two lists must have equal lengths.
zip3	O(N)	O(N)	O(N)	-	-	Combines the three collections into a list of triples. The lists must have equal lengths.

See also

- F# Types
- F# Language Reference

Lists

7/30/2019 • 23 minutes to read • Edit Online

> **NOTE**
> The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

A list in F# is an ordered, immutable series of elements of the same type. To perform basic operations on lists, use the functions in the List module.

Creating and Initializing Lists

You can define a list by explicitly listing out the elements, separated by semicolons and enclosed in square brackets, as shown in the following line of code.

```
let list123 = [ 1; 2; 3 ]
```

You can also put line breaks between elements, in which case the semicolons are optional. The latter syntax can result in more readable code when the element initialization expressions are longer, or when you want to include a comment for each element.

```
let list123 = [
    1
    2
    3 ]
```

Normally, all list elements must be the same type. An exception is that a list in which the elements are specified to be a base type can have elements that are derived types. Thus the following is acceptable, because both `Button` and `CheckBox` derive from `Control`.

```
let myControlList : Control list = [ new Button(); new CheckBox() ]
```

You can also define list elements by using a range indicated by integers separated by the range operator (`..`), as shown in the following code.

```
let list1 = [ 1 .. 10 ]
```

An empty list is specified by a pair of square brackets with nothing in between them.

```
// An empty list.
let listEmpty = []
```

You can also use a sequence expression to create a list. See Sequence Expressions for more information. For example, the following code creates a list of squares of integers from 1 to 10.

```
let listOfSquares = [ for i in 1 .. 10 -> i*i ]
```

Operators for Working with Lists

You can attach elements to a list by using the `::` (cons) operator. If `list1` is `[2; 3; 4]`, the following code creates `list2` as `[100; 2; 3; 4]`.

```
let list2 = 100 :: list1
```

You can concatenate lists that have compatible types by using the `@` operator, as in the following code. If `list1` is `[2; 3; 4]` and `list2` is `[100; 2; 3; 4]`, this code creates `list3` as `[2; 3; 4; 100; 2; 3; 4]`.

```
let list3 = list1 @ list2
```

Functions for performing operations on lists are available in the List module.

Because lists in F# are immutable, any modifying operations generate new lists instead of modifying existing lists.

Lists in F# are implemented as singly linked lists, which means that operations that access only the head of the list are O(1), and element access is O(*n*).

Properties

The list type supports the following properties:

PROPERTY	TYPE	DESCRIPTION
Head	`'T`	The first element.
Empty	`'T list`	A static property that returns an empty list of the appropriate type.
IsEmpty	`bool`	`true` if the list has no elements.
Item	`'T`	The element at the specified index (zero-based).
Length	`int`	The number of elements.
Tail	`'T list`	The list without the first element.

Following are some examples of using these properties.

```
let list1 = [ 1; 2; 3 ]

// Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

Using Lists

Programming with lists enables you to perform complex operations with a small amount of code. This section describes common operations on lists that are important to functional programming.

Recursion with Lists

Lists are uniquely suited to recursive programming techniques. Consider an operation that must be performed on every element of a list. You can do this recursively by operating on the head of the list and then passing the tail of the list, which is a smaller list that consists of the original list without the first element, back again to the next level of recursion.

To write such a recursive function, you use the cons operator (`::`) in pattern matching, which enables you to separate the head of a list from the tail.

The following code example shows how to use pattern matching to implement a recursive function that performs operations on a list.

```
let rec sum list =
   match list with
   | head :: tail -> head + sum tail
   | [] -> 0
```

The previous code works well for small lists, but for larger lists, it could overflow the stack. The following code improves on this code by using an accumulator argument, a standard technique for working with recursive functions. The use of the accumulator argument makes the function tail recursive, which saves stack space.

```
let sum list =
   let rec loop list acc =
       match list with
       | head :: tail -> loop tail (acc + head)
       | [] -> acc
   loop list 0
```

The function `RemoveAllMultiples` is a recursive function that takes two lists. The first list contains the numbers whose multiples will be removed, and the second list is the list from which to remove the numbers. The code in the following example uses this recursive function to eliminate all the non-prime numbers from a list, leaving a list of prime numbers as the result.

```
let IsPrimeMultipleTest n x =
   x = n || x % n <> 0

let rec RemoveAllMultiples listn listx =
   match listn with
   | head :: tail -> RemoveAllMultiples tail (List.filter (IsPrimeMultipleTest head) listx)
   | [] -> listx


let GetPrimesUpTo n =
    let max = int (sqrt (float n))
    RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

The output is as follows:

```
Primes Up To 100:
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]
```

Module Functions

The List module provides functions that access the elements of a list. The head element is the fastest and easiest to access. Use the property Head or the module function List.head. You can access the tail of a list by using the Tail property or the List.tail function. To find an element by index, use the List.nth function. `List.nth` traverses the list. Therefore, it is O(*n*). If your code uses `List.nth` frequently, you might want to consider using an array instead of a list. Element access in arrays is O(1).

Boolean Operations on Lists

The List.isEmpty function determines whether a list has any elements.

The List.exists function applies a Boolean test to elements of a list and returns `true` if any element satisfies the test. List.exists2 is similar but operates on successive pairs of elements in two lists.

The following code demonstrates the use of `List.exists`.

```
// Use List.exists to determine whether there is an element of a list satisfies a given Boolean expression.
// containsNumber returns true if any of the elements of the supplied list match
// the supplied number.
let containsNumber number list = List.exists (fun elem -> elem = number) list
let list0to3 = [0 .. 3]
printfn "For list %A, contains zero is %b" list0to3 (containsNumber 0 list0to3)
```

The output is as follows:

```
For list [0; 1; 2; 3], contains zero is true
```

The following example demonstrates the use of `List.exists2`.

```
// Use List.exists2 to compare elements in two lists.
// isEqualElement returns true if any elements at the same position in two supplied
// lists match.
let isEqualElement list1 list2 = List.exists2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
let list1to5 = [ 1 .. 5 ]
let list5to1 = [ 5 .. -1 .. 1 ]
if (isEqualElement list1to5 list5to1) then
    printfn "Lists %A and %A have at least one equal element at the same position." list1to5 list5to1
else
    printfn "Lists %A and %A do not have an equal element at the same position." list1to5 list5to1
```

The output is as follows:

```
Lists [1; 2; 3; 4; 5] and [5; 4; 3; 2; 1] have at least one equal element at the same position.
```

You can use List.forall if you want to test whether all the elements of a list meet a condition.

```
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list
printfn "%b" (isAllZeroes [0.0; 0.0])
printfn "%b" (isAllZeroes [0.0; 1.0])
```

The output is as follows:

```
true
false
```

Similarly, List.forall2 determines whether all elements in the corresponding positions in two lists satisfy a Boolean expression that involves each pair of elements.

```
let listEqual list1 list2 = List.forall2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
printfn "%b" (listEqual [0; 1; 2] [0; 1; 2])
printfn "%b" (listEqual [0; 0; 0] [0; 1; 0])
```

The output is as follows:

```
true
false
```

Sort Operations on Lists

The List.sort, List.sortBy, and List.sortWith functions sort lists. The sorting function determines which of these three functions to use. `List.sort` uses default generic comparison. Generic comparison uses global operators based on the generic compare function to compare values. It works efficiently with a wide variety of element types, such as simple numeric types, tuples, records, discriminated unions, lists, arrays, and any type that implements `System.IComparable`. For types that implement `System.IComparable`, generic comparison uses the `System.IComparable.CompareTo()` function. Generic comparison also works with strings, but uses a culture-independent sorting order. Generic comparison should not be used on unsupported types, such as function types. Also, the performance of the default generic comparison is best for small structured types; for larger structured types that need to be compared and sorted frequently, consider implementing `System.IComparable` and providing an efficient implementation of the `System.IComparable.CompareTo()` method.

`List.sortBy` takes a function that returns a value that is used as the sort criterion, and `List.sortWith` takes a comparison function as an argument. These latter two functions are useful when you are working with types that do not support comparison, or when the comparison requires more complex comparison semantics, as in the case of culture-aware strings.

The following example demonstrates the use of `List.sort`.

```
let sortedList1 = List.sort [1; 4; 8; -2; 5]
printfn "%A" sortedList1
```

The output is as follows:

```
[-2; 1; 4; 5; 8]
```

The following example demonstrates the use of `List.sortBy`.

```
let sortedList2 = List.sortBy (fun elem -> abs elem) [1; 4; 8; -2; 5]
printfn "%A" sortedList2
```

The output is as follows:

```
[1; -2; 4; 5; 8]
```

The next example demonstrates the use of `List.sortWith`. In this example, the custom comparison function `compareWidgets` is used to first compare one field of a custom type, and then another when the values of the first field are equal.

```
type Widget = { ID: int; Rev: int }

let compareWidgets widget1 widget2 =
   if widget1.ID < widget2.ID then -1 else
   if widget1.ID > widget2.ID then 1 else
   if widget1.Rev < widget2.Rev then -1 else
   if widget1.Rev > widget2.Rev then 1 else
   0

let listToCompare = [
    { ID = 92; Rev = 1 }
    { ID = 110; Rev = 1 }
    { ID = 100; Rev = 5 }
    { ID = 100; Rev = 2 }
    { ID = 92; Rev = 1 }
    ]

let sortedWidgetList = List.sortWith compareWidgets listToCompare
printfn "%A" sortedWidgetList
```

The output is as follows:

```
[{ID = 92;
Rev = 1;}; {ID = 92;
Rev = 1;}; {ID = 100;
Rev = 2;}; {ID = 100;
Rev = 5;}; {ID = 110;
Rev = 1;}]
```

Search Operations on Lists

Numerous search operations are supported for lists. The simplest, List.find, enables you to find the first element that matches a given condition.

The following code example demonstrates the use of `List.find` to find the first number that is divisible by 5 in a list.

```
let isDivisibleBy number elem = elem % number = 0
let result = List.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

The result is 5.

If the elements must be transformed first, call List.pick, which takes a function that returns an option, and looks for the first option value that is `Some(x)`. Instead of returning the element, `List.pick` returns the result `x`. If no matching element is found, `List.pick` throws `System.Collections.Generic.KeyNotFoundException`. The following code shows the use of `List.pick`.

```
let valuesList = [ ("a", 1); ("b", 2); ("c", 3) ]

let resultPick = List.pick (fun elem ->
                    match elem with
                    | (value, 2) -> Some value
                    | _ -> None) valuesList
printfn "%A" resultPick
```

The output is as follows:

```
"b"
```

Another group of search operations, List.tryFind and related functions, return an option value. The `List.tryFind` function returns the first element of a list that satisfies a condition if such an element exists, but the option value `None` if not. The variation List.tryFindIndex returns the index of the element, if one is found, rather than the element itself. These functions are illustrated in the following code.

```
let list1d = [1; 3; 7; 9; 11; 13; 15; 19; 22; 29; 36]
let isEven x = x % 2 = 0
match List.tryFind isEven list1d with
| Some value -> printfn "The first even value is %d." value
| None -> printfn "There is no even value in the list."

match List.tryFindIndex isEven list1d with
| Some value -> printfn "The first even value is at position %d." value
| None -> printfn "There is no even value in the list."
```

The output is as follows:

```
The first even value is 22.
The first even value is at position 8.
```

Arithmetic Operations on Lists

Common arithmetic operations such as sum and average are built into the List module. To work with List.sum, the list element type must support the `+` operator and have a zero value. All built-in arithmetic types satisfy these conditions. To work with List.average, the element type must support division without a remainder, which excludes integral types but allows for floating point types. The List.sumBy and List.averageBy functions take a function as a parameter, and this function's results are used to calculate the values for the sum or average.

The following code demonstrates the use of `List.sum`, `List.sumBy`, and `List.average`.

```
// Compute the sum of the first 10 integers by using List.sum.
let sum1 = List.sum [1 .. 10]

// Compute the sum of the squares of the elements of a list by using List.sumBy.
let sum2 = List.sumBy (fun elem -> elem*elem) [1 .. 10]

// Compute the average of the elements of a list by using List.average.
let avg1 = List.average [0.0; 1.0; 1.0; 2.0]

printfn "%f" avg1
```

The output is `1.000000`.

The following code shows the use of `List.averageBy`.

```
let avg2 = List.averageBy (fun elem -> float elem) [1 .. 10]
printfn "%f" avg2
```

The output is `5.5`.

Lists and Tuples

Lists that contain tuples can be manipulated by zip and unzip functions. These functions combine two lists of single values into one list of tuples or separate one list of tuples into two lists of single values. The simplest List.zip function takes two lists of single elements and produces a single list of tuple pairs. Another version,

List.zip3, takes three lists of single elements and produces a single list of tuples that have three elements. The following code example demonstrates the use of `List.zip`.

```
let list1 = [ 1; 2; 3 ]
let list2 = [ -1; -2; -3 ]
let listZip = List.zip list1 list2
printfn "%A" listZip
```

The output is as follows:

```
[(1, -1); (2, -2); (3; -3)]
```

The following code example demonstrates the use of `List.zip3`.

```
let list3 = [ 0; 0; 0]
let listZip3 = List.zip3 list1 list2 list3
printfn "%A" listZip3
```

The output is as follows:

```
[(1, -1, 0); (2, -2, 0); (3, -3, 0)]
```

The corresponding unzip versions, List.unzip and List.unzip3, take lists of tuples and return lists in a tuple, where the first list contains all the elements that were first in each tuple, and the second list contains the second element of each tuple, and so on.

The following code example demonstrates the use of List.unzip.

```
let lists = List.unzip [(1,2); (3,4)]
printfn "%A" lists
printfn "%A %A" (fst lists) (snd lists)
```

The output is as follows:

```
([1; 3], [2; 4])
[1; 3] [2; 4]
```

The following code example demonstrates the use of List.unzip3.

```
let listsUnzip3 = List.unzip3 [(1,2,3); (4,5,6)]
printfn "%A" listsUnzip3
```

The output is as follows:

```
([1; 4], [2; 5], [3; 6])
```

Operating on List Elements

F# supports a variety of operations on list elements. The simplest is List.iter, which enables you to call a function on every element of a list. Variations include List.iter2, which enables you to perform an operation on elements of two lists, List.iteri, which is like `List.iter` except that the index of each element is passed as an argument to the function that is called for each element, and List.iteri2, which is a combination of the

functionality of `List.iter2` and `List.iteri`. The following code example illustrates these functions.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
List.iter (fun x -> printfn "List.iter: element is %d" x) list1
List.iteri(fun i x -> printfn "List.iteri: element %d is %d" i x) list1
List.iter2 (fun x y -> printfn "List.iter2: elements are %d %d" x y) list1 list2
List.iteri2 (fun i x y ->
                printfn "List.iteri2: element %d of list1 is %d element %d of list2 is %d"
                  i x i y)
            list1 list2
```

The output is as follows:

```
List.iter: element is 1
List.iter: element is 2
List.iter: element is 3
List.iteri: element 0 is 1
List.iteri: element 1 is 2
List.iteri: element 2 is 3
List.iter2: elements are 1 4
List.iter2: elements are 2 5
List.iter2: elements are 3 6
List.iteri2: element 0 of list1 is 1; element 0 of list2 is 4
List.iteri2: element 1 of list1 is 2; element 1 of list2 is 5
List.iteri2: element 2 of list1 is 3; element 2 of list2 is 6
```

Another frequently used function that transforms list elements is List.map, which enables you to apply a function to each element of a list and put all the results into a new list. List.map2 and List.map3 are variations that take multiple lists. You can also use List.mapi and List.mapi2, if, in addition to the element, the function needs to be passed the index of each element. The only difference between `List.mapi2` and `List.mapi` is that `List.mapi2` works with two lists. The following example illustrates List.map.

```
let list1 = [1; 2; 3]
let newList = List.map (fun x -> x + 1) list1
printfn "%A" newList
```

The output is as follows:

```
[2; 3; 4]
```

The following example shows the use of `List.map2`.

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let sumList = List.map2 (fun x y -> x + y) list1 list2
printfn "%A" sumList
```

The output is as follows:

```
[5; 7; 9]
```

The following example shows the use of `List.map3`.

```
let newList2 = List.map3 (fun x y z -> x + y + z) list1 list2 [2; 3; 4]
printfn "%A" newList2
```

The output is as follows:

```
[7; 10; 13]
```

The following example shows the use of `List.mapi`.

```
let newListAddIndex = List.mapi (fun i x -> x + i) list1
printfn "%A" newListAddIndex
```

The output is as follows:

```
[1; 3; 5]
```

The following example shows the use of `List.mapi2`.

```
let listAddTimesIndex = List.mapi2 (fun i x y -> (x + y) * i) list1 list2
printfn "%A" listAddTimesIndex
```

The output is as follows:

```
[0; 7; 18]
```

List.collect is like `List.map`, except that each element produces a list and all these lists are concatenated into a final list. In the following code, each element of the list generates three numbers. These are all collected into one list.

```
let collectList = List.collect (fun x -> [for i in 1..3 -> x * i]) list1
printfn "%A" collectList
```

The output is as follows:

```
[1; 2; 3; 2; 4; 6; 3; 6; 9]
```

You can also use List.filter, which takes a Boolean condition and produces a new list that consists only of elements that satisfy the given condition.

```
let evenOnlyList = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6]
```

The resulting list is `[2; 4; 6]`.

A combination of map and filter, List.choose enables you to transform and select elements at the same time. `List.choose` applies a function that returns an option to each element of a list, and returns a new list of the results for elements when the function returns the option value `Some`.

The following code demonstrates the use of `List.choose` to select capitalized words out of a list of words.

```
let listWords = [ "and"; "Rome"; "Bob"; "apple"; "zebra" ]
let isCapitalized (string1:string) = System.Char.IsUpper string1.[0]
let results = List.choose (fun elem ->
    match elem with
    | elem when isCapitalized elem -> Some(elem + "'s")
    | _ -> None) listWords
printfn "%A" results
```

The output is as follows:

```
["Rome's"; "Bob's"]
```

Operating on Multiple Lists

Lists can be joined together. To join two lists into one, use List.append. To join more than two lists, use List.concat.

```
let list1to10 = List.append [1; 2; 3] [4; 5; 6; 7; 8; 9; 10]
let listResult = List.concat [ [1; 2; 3]; [4; 5; 6]; [7; 8; 9] ]
List.iter (fun elem -> printf "%d " elem) list1to10
printfn ""
List.iter (fun elem -> printf "%d " elem) listResult
```

Fold and Scan Operations

Some list operations involve interdependencies between all of the list elements. The fold and scan operations are like `List.iter` and `List.map` in that you invoke a function on each element, but these operations provide an additional parameter called the *accumulator* that carries information through the computation.

Use `List.fold` to perform a calculation on a list.

The following code example demonstrates the use of List.fold to perform various operations.

The list is traversed; the accumulator `acc` is a value that is passed along as the calculation proceeds. The first argument takes the accumulator and the list element, and returns the interim result of the calculation for that list element. The second argument is the initial value of the accumulator.

```
let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 3 ] (sumList [ 1 .. 3 ])

// The following example computes the average of a list.
let averageList list = (List.fold (fun acc elem -> acc + float elem) 0.0 list / float list.Length)

// The following example computes the standard deviation of a list.
// The standard deviation is computed by taking the square root of the
// sum of the variances, which are the differences between each value
// and the average.
let stdDevList list =
    let avg = averageList list
    sqrt (List.fold (fun acc elem -> acc + (float elem - avg) ** 2.0 ) 0.0 list / float list.Length)

let testList listTest =
    printfn "List %A average: %f stddev: %f" listTest (averageList listTest) (stdDevList listTest)

testList [1; 1; 1]
testList [1; 2; 1]
testList [1; 2; 3]

// List.fold is the same as to List.iter when the accumulator is not used.
let printList list = List.fold (fun acc elem -> printfn "%A" elem) () list
printList [0.0; 1.0; 2.5; 5.1 ]

// The following example uses List.fold to reverse a list.
// The accumulator starts out as the empty list, and the function uses the cons operator
// to add each successive element to the head of the accumulator list, resulting in a
// reversed form of the list.
let reverseList list = List.fold (fun acc elem -> elem::acc) [] list
printfn "%A" (reverseList [1 .. 10])
```

The versions of these functions that have a digit in the function name operate on more than one list. For example, List.fold2 performs computations on two lists.

The following example demonstrates the use of `List.fold2`.

```
// Use List.fold2 to perform computations over two lists (of equal size) at the same time.
// Example: Sum the greater element at each list position.
let sumGreatest list1 list2 = List.fold2 (fun acc elem1 elem2 ->
                                            acc + max elem1 elem2) 0 list1 list2

let sum = sumGreatest [1; 2; 3] [3; 2; 1]
printfn "The sum of the greater of each pair of elements in the two lists is %d." sum
```

`List.fold` and List.scan differ in that `List.fold` returns the final value of the extra parameter, but `List.scan` returns the list of the intermediate values (along with the final value) of the extra parameter.

Each of these functions includes a reverse variation, for example, List.foldBack, which differs in the order in which the list is traversed and the order of the arguments. Also, `List.fold` and `List.foldBack` have variations, List.fold2 and List.foldBack2, that take two lists of equal length. The function that executes on each element can use corresponding elements of both lists to perform some action. The element types of the two lists can be different, as in the following example, in which one list contains transaction amounts for a bank account, and the other list contains the type of transaction: deposit or withdrawal.

```
// Discriminated union type that encodes the transaction type.
type Transaction =
    | Deposit
    | Withdrawal

let transactionTypes = [Deposit; Deposit; Withdrawal]
let transactionAmounts = [100.00; 1000.00; 95.00 ]
let initialBalance = 200.00

// Use fold2 to perform a calculation on the list to update the account balance.
let endingBalance = List.fold2 (fun acc elem1 elem2 ->
                                match elem1 with
                                | Deposit -> acc + elem2
                                | Withdrawal -> acc - elem2)
                                initialBalance
                                transactionTypes
                                transactionAmounts
printfn "%f" endingBalance
```

For a calculation like summation, `List.fold` and `List.foldBack` have the same effect because the result does not depend on the order of traversal. In the following example, `List.foldBack` is used to add the elements in a list.

```
let sumListBack list = List.foldBack (fun acc elem -> acc + elem) list 0
printfn "%d" (sumListBack [1; 2; 3])

// For a calculation in which the order of traversal is important, fold and foldBack have different
// results. For example, replacing fold with foldBack in the listReverse function
// produces a function that copies the list, rather than reversing it.
let copyList list = List.foldBack (fun elem acc -> elem::acc) list []
printfn "%A" (copyList [1 .. 10])
```

The following example returns to the bank account example. This time a new transaction type is added: an interest calculation. The ending balance now depends on the order of transactions.

```
type Transaction2 =
    | Deposit
    | Withdrawal
    | Interest

let transactionTypes2 = [Deposit; Deposit; Withdrawal; Interest]
let transactionAmounts2 = [100.00; 1000.00; 95.00; 0.05 / 12.0 ]
let initialBalance2 = 200.00

// Because fold2 processes the lists by starting at the head element,
// the interest is calculated last, on the balance of 1205.00.
let endingBalance2 = List.fold2 (fun acc elem1 elem2 ->
                                 match elem1 with
                                 | Deposit -> acc + elem2
                                 | Withdrawal -> acc - elem2
                                 | Interest -> acc * (1.0 + elem2))
                                 initialBalance2
                                 transactionTypes2
                                 transactionAmounts2
printfn "%f" endingBalance2


// Because foldBack2 processes the lists by starting at end of the list,
// the interest is calculated first, on the balance of only 200.00.
let endingBalance3 = List.foldBack2 (fun elem1 elem2 acc ->
                                 match elem1 with
                                 | Deposit -> acc + elem2
                                 | Withdrawal -> acc - elem2
                                 | Interest -> acc * (1.0 + elem2))
                                 transactionTypes2
                                 transactionAmounts2
                                 initialBalance2
printfn "%f" endingBalance3
```

The function List.reduce is somewhat like `List.fold` and `List.scan`, except that instead of passing around a separate accumulator, `List.reduce` takes a function that takes two arguments of the element type instead of just one, and one of those arguments acts as the accumulator, meaning that it stores the intermediate result of the computation. `List.reduce` starts by operating on the first two list elements, and then uses the result of the operation along with the next element. Because there is not a separate accumulator that has its own type, `List.reduce` can be used in place of `List.fold` only when the accumulator and the element type have the same type. The following code demonstrates the use of `List.reduce`. `List.reduce` throws an exception if the list provided has no elements.

In the following code, the first call to the lambda expression is given the arguments 2 and 4, and returns 6, and the next call is given the arguments 6 and 10, so the result is 16.

```
let sumAList list =
    try
        List.reduce (fun acc elem -> acc + elem) list
    with
       | :? System.ArgumentException as exc -> 0

let resultSum = sumAList [2; 4; 10]
printfn "%d " resultSum
```

Converting Between Lists and Other Collection Types

The `List` module provides functions for converting to and from both sequences and arrays. To convert to or from a sequence, use List.toSeq or List.ofSeq. To convert to or from an array, use List.toArray or List.ofArray.

Additional Operations

For information about additional operations on lists, see the library reference topic Collections.List Module.

See also

- F# Language Reference
- F# Types
- Sequences
- Arrays
- Options

Arrays

7/30/2019 • 17 minutes to read • Edit Online

> **NOTE**
> The API reference link will take you to MSDN. The docs.microsoft.com API reference is not complete.

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.

Creating Arrays

You can create arrays in several ways. You can create a small array by listing consecutive values between `[|` and `|]` and separated by semicolons, as shown in the following examples.

```
let array1 = [| 1; 2; 3 |]
```

You can also put each element on a separate line, in which case the semicolon separator is optional.

```
let array1 =
    [|
        1
        2
        3
     |]
```

The type of the array elements is inferred from the literals used and must be consistent. The following code causes an error because 1.0 is a float and 2 and 3 are integers.

```
// Causes an error.
// let array2 = [| 1.0; 2; 3 |]
```

You can also use sequence expressions to create arrays. Following is an example that creates an array of squares of integers from 1 to 10.

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

To create an array in which all the elements are initialized to zero, use `Array.zeroCreate`.

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

Accessing Elements

You can access array elements by using a dot operator (`.`) and brackets (`[` and `]`).

```
array1.[0]
```

Array indexes start at 0.

You can also access array elements by using slice notation, which enables you to specify a subrange of the array. Examples of slice notation follow.

```
// Accesses elements from 0 to 2.

array1.[0..2]

// Accesses elements from the beginning of the array to 2.

array1.[..2]

// Accesses elements from 2 to the end of the array.

array1.[2..]
```

When slice notation is used, a new copy of the array is created.

Array Types and Modules

The type of all F# arrays is the .NET Framework type System.Array. Therefore, F# arrays support all the functionality available in System.Array.

The library module `Microsoft.FSharp.Collections.Array` supports operations on one-dimensional arrays. The modules `Array2D`, `Array3D`, and `Array4D` contain functions that support operations on arrays of two, three, and four dimensions, respectively. You can create arrays of rank greater than four by using System.Array.

Simple Functions

`Array.get` gets an element. `Array.length` gives the length of an array. `Array.set` sets an element to a specified value. The following code example illustrates the use of these functions.

```
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
    Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
    printf "%s " (Array.get array1 i)
```

The output is as follows.

```
0 1 2 3 4 5 6 7 8 9
```

Functions That Create Arrays

Several functions create arrays without requiring an existing array. `Array.empty` creates a new array that does not contain any elements. `Array.create` creates an array of a specified size and sets all the elements to provided values. `Array.init` creates an array, given a dimension and a function to generate the elements. `Array.zeroCreate` creates an array in which all the elements are initialized to the zero value for the array's type. The following code demonstrates these functions.

```
let myEmptyArray = Array.empty
printfn "Length of empty array: %d" myEmptyArray.Length



printfn "Array of floats set to 5.0: %A" (Array.create 10 5.0)


printfn "Array of squares: %A" (Array.init 10 (fun index -> index * index))

let (myZeroArray : float array) = Array.zeroCreate 10
```

The output is as follows.

```
Length of empty array: 0
Area of floats set to 5.0: [|5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.copy` creates a new array that contains elements that are copied from an existing array. Note that the copy is a shallow copy, which means that if the element type is a reference type, only the reference is copied, not the underlying object. The following code example illustrates this.

```
open System.Text

let firstArray : StringBuilder array = Array.init 3 (fun index -> new StringBuilder(""))
let secondArray = Array.copy firstArray
// Reset an element of the first array to a new value.
firstArray.[0] <- new StringBuilder("Test1")
// Change an element of the first array.
firstArray.[1].Insert(0, "Test2") |> ignore
printfn "%A" firstArray
printfn "%A" secondArray
```

The output of the preceding code is as follows:

```
[|Test1; Test2; |]
[|; Test2; |]
```

The string `Test1` appears only in the first array because the operation of creating a new element overwrites the reference in `firstArray` but does not affect the original reference to an empty string that is still present in `secondArray`. The string `Test2` appears in both arrays because the `Insert` operation on the System.Text.StringBuilder type affects the underlying System.Text.StringBuilder object, which is referenced in both arrays.

`Array.sub` generates a new array from a subrange of an array. You specify the subrange by providing the starting index and the length. The following code demonstrates the use of `Array.sub`.

```
let a1 = [| 0 .. 99 |]
let a2 = Array.sub a1 5 10
printfn "%A" a2
```

The output shows that the subarray starts at element 5 and contains 10 elements.

```
[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14|]
```

`Array.append` creates a new array by combining two existing arrays.

The following code demonstrates **Array.append**.

```
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

The output of the preceding code is as follows.

```
[|1; 2; 3; 4; 5; 6|]
```

`Array.choose` selects elements of an array to include in a new array. The following code demonstrates `Array.choose`. Note that the element type of the array does not have to match the type of the value returned in the option type. In this example, the element type is `int` and the option is the result of a polynomial function, `elem*elem - 1`, as a floating point number.

```
printfn "%A" (Array.choose (fun elem -> if elem % 2 = 0 then
                                            Some(float (elem*elem - 1))
                                        else
                                            None) [| 1 .. 10 |])
```

The output of the preceding code is as follows.

```
[|3.0; 15.0; 35.0; 63.0; 99.0|]
```

`Array.collect` runs a specified function on each array element of an existing array and then collects the elements generated by the function and combines them into a new array. The following code demonstrates `Array.collect`.

```
printfn "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

The output of the preceding code is as follows.

```
[|0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

`Array.concat` takes a sequence of arrays and combines them into a single array. The following code demonstrates `Array.concat`.

```
Array.concat [ [|0..3|] ; [|4|] ]
//output [|0; 1; 2; 3; 4|]

Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]
```

The output of the preceding code is as follows.

```
[|(1, 1, 1); (1, 2, 2); (1, 3, 3); (2, 1, 2); (2, 2, 4); (2, 3, 6); (3, 1, 3);
(3, 2, 6); (3, 3, 9)|]
```

`Array.filter` takes a Boolean condition function and generates a new array that contains only those elements from the input array for which the condition is true. The following code demonstrates `Array.filter`.

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

The output of the preceding code is as follows.

```
[|2; 4; 6; 8; 10|]
```

`Array.rev` generates a new array by reversing the order of an existing array. The following code demonstrates `Array.rev`.

```
let stringReverse (s: string) =
    System.String(Array.rev (s.ToCharArray()))

printfn "%A" (stringReverse("!dlrow olleH"))
```

The output of the preceding code is as follows.

```
"Hello world!"
```

You can easily combine functions in the array module that transform arrays by using the pipeline operator (`|>`), as shown in the following example.

```
[| 1 .. 10 |]
|> Array.filter (fun elem -> elem % 2 = 0)
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)
|> Array.rev
|> printfn "%A"
```

The output is

```
[|100; 36; 16; 4|]
```

Multidimensional Arrays

A multidimensional array can be created, but there is no syntax for writing a multidimensional array literal. Use the operator `array2D` to create an array from a sequence of sequences of array elements. The sequences can be array or list literals. For example, the following code creates a two-dimensional array.

```
let my2DArray = array2D [ [ 1; 0]; [0; 1] ]
```

You can also use the function `Array2D.init` to initialize arrays of two dimensions, and similar functions are available for arrays of three and four dimensions. These functions take a function that is used to create the elements. To create a two-dimensional array that contains elements set to an initial value instead of specifying a function, use the `Array2D.create` function, which is also available for arrays up to four dimensions. The following code example first shows how to create an array of arrays that contain the desired elements, and then uses `Array2D.init` to generate the desired two-dimensional array.

```
let arrayOfArrays = [| [| 1.0; 0.0 |]; [|0.0; 1.0 |] |]
let twoDimensionalArray = Array2D.init 2 2 (fun i j -> arrayOfArrays.[i].[j])
```

Array indexing and slicing syntax is supported for arrays up to rank 4. When you specify an index in multiple dimensions, you use commas to separate the indexes, as illustrated in the following code example.

```
twoDimensionalArray.[0, 1] <- 1.0
```

The type of a two-dimensional array is written out as `<type>[,]` (for example, `int[,]`, `double[,]`), and the type of a three-dimensional array is written as `<type>[,,]`, and so on for arrays of higher dimensions.

Only a subset of the functions available for one-dimensional arrays is also available for multidimensional arrays. For more information, see `Collections.Array Module`, `Collections.Array2D Module`, `Collections.Array3D Module`, and `Collections.Array4D Module`.

Array Slicing and Multidimensional Arrays

In a two-dimensional array (a matrix), you can extract a sub-matrix by specifying ranges and using a wildcard (`*`) character to specify whole rows or columns.

```
// Get rows 1 to N from an NxM matrix (returns a matrix):
matrix.[1.., *]

// Get rows 1 to 3 from a matrix (returns a matrix):
matrix.[1..3, *]

// Get columns 1 to 3 from a matrix (returns a matrix):
matrix.[*, 1..3]

// Get a 3x3 submatrix:
matrix.[1..3, 1..3]
```

As of F# 3.1, you can decompose a multidimensional array into subarrays of the same or lower dimension. For example, you can obtain a vector from a matrix by specifying a single row or column.

```
// Get row 3 from a matrix as a vector:
matrix.[3, *]

// Get column 3 from a matrix as a vector:
matrix.[*, 3]
```

You can use this slicing syntax for types that implement the element access operators and overloaded `GetSlice` methods. For example, the following code creates a Matrix type that wraps the F# 2D array, implements an Item property to provide support for array indexing, and implements three versions of `GetSlice`. If you can use this code as a template for your matrix types, you can use all the slicing operations that this section describes.

```
type Matrix<'T>(N: int, M: int) =
    let internalArray = Array2D.zeroCreate<'T> N M

    member this.Item
        with get(a: int, b: int) = internalArray.[a, b]
        and set(a: int, b: int) (value:'T) = internalArray.[a, b] <- value

    member this.GetSlice(rowStart: int option, rowFinish : int option, colStart: int option, colFinish : int
option) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[rowStart..rowFinish, colStart..colFinish]

    member this.GetSlice(row: int, colStart: int option, colFinish: int option) =
        let colStart =
            match colStart with
            | Some(v) -> v
            | None -> 0
        let colFinish =
            match colFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(1) - 1
        internalArray.[row, colStart..colFinish]

    member this.GetSlice(rowStart: int option, rowFinish: int option, col: int) =
        let rowStart =
            match rowStart with
            | Some(v) -> v
            | None -> 0
        let rowFinish =
            match rowFinish with
            | Some(v) -> v
            | None -> internalArray.GetLength(0) - 1
        internalArray.[rowStart..rowFinish, col]

module test =
    let generateTestMatrix x y =
        let matrix = new Matrix<float>(3, 3)
        for i in 0..2 do
            for j in 0..2 do
                matrix.[i, j] <- float(i) * x - float(j) * y
        matrix

    let test1 = generateTestMatrix 2.3 1.1
    let submatrix = test1.[0..1, 0..1]
    printfn "%A" submatrix

    let firstRow = test1.[0,*]
    let secondRow = test1.[1,*]
    let firstCol = test1.[*,0]
    printfn "%A" firstCol
```

Boolean Functions on Arrays

The functions `Array.exists` and `Array.exists2` test elements in either one or two arrays, respectively. These functions take a test function and return `true` if there is an element (or element pair for `Array.exists2`) that satisfies the condition.

The following code demonstrates the use of `Array.exists` and `Array.exists2`. In these examples, new functions are created by applying only one of the arguments, in these cases, the function argument.

```
let allNegative = Array.exists (fun elem -> abs (elem) = elem) >> not
printfn "%A" (allNegative [| -1; -2; -3 |])
printfn "%A" (allNegative [| -10; -1; 5 |])
printfn "%A" (allNegative [| 0 |])


let haveEqualElement = Array.exists2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (haveEqualElement [| 1; 2; 3 |] [| 3; 2; 1|])
```

The output of the preceding code is as follows.

```
true
false
false
true
```

Similarly, the function `Array.forall` tests an array to determine whether every element satisfies a Boolean condition. The variation `Array.forall2` does the same thing by using a Boolean function that involves elements of two arrays of equal length. The following code illustrates the use of these functions.

```
let allPositive = Array.forall (fun elem -> elem > 0)
printfn "%A" (allPositive [| 0; 1; 2; 3 |])
printfn "%A" (allPositive [| 1; 2; 3 |])


let allEqual = Array.forall2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (allEqual [| 1; 2 |] [| 1; 2 |])
printfn "%A" (allEqual [| 1; 2 |] [| 2; 1 |])
```

The output for these examples is as follows.

```
false
true
true
false
```

Searching Arrays

`Array.find` takes a Boolean function and returns the first element for which the function returns `true`, or raises a System.Collections.Generic.KeyNotFoundException if no element that satisfies the condition is found. `Array.findIndex` is like `Array.find`, except that it returns the index of the element instead of the element itself.

The following code uses `Array.find` and `Array.findIndex` to locate a number that is both a perfect square and perfect cube.

```
let arrayA = [| 2 .. 100 |]
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
printfn "The first element that is both a square and a cube is %d and its index is %d." element index
```

The output is as follows.

```
The first element that is both a square and a cube is 64 and its index is 62.
```

`Array.tryFind` is like `Array.find`, except that its result is an option type, and it returns `None` if no element is found. `Array.tryFind` should be used instead of `Array.find` when you do not know whether a matching element is in the array. Similarly, `Array.tryFindIndex` is like `Array.findIndex` except that the option type is the return value. If no element is found, the option is `None`.

The following code demonstrates the use of `Array.tryFind`. This code depends on the previous code.

```
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let lookForCubeAndSquare array1 =
    let result = Array.tryFind (fun elem -> isPerfectSquare elem && isPerfectCube elem) array1
    match result with
    | Some x -> printfn "Found an element: %d" x
    | None -> printfn "Failed to find a matching element."

lookForCubeAndSquare [| 1 .. 10 |]
lookForCubeAndSquare [| 100 .. 1000 |]
lookForCubeAndSquare [| 2 .. 50 |]
```

The output is as follows.

```
Found an element: 1
Found an element: 729
```

Use `Array.tryPick` when you need to transform an element in addition to finding it. The result is the first element for which the function returns the transformed element as an option value, or `None` if no such element is found.

The following code shows the use of `Array.tryPick`. In this case, instead of a lambda expression, several local helper functions are defined to simplify the code.

```
let findPerfectSquareAndCube array1 =
    let delta = 1.0e-10
    let isPerfectSquare (x:int) =
        let y = sqrt (float x)
        abs(y - round y) < delta
    let isPerfectCube (x:int) =
        let y = System.Math.Pow(float x, 1.0/3.0)
        abs(y - round y) < delta
    // intFunction : (float -> float) -> int -> int
    // Allows the use of a floating point function with integers.
    let intFunction function1 number = int (round (function1 (float number)))
    let cubeRoot x = System.Math.Pow(x, 1.0/3.0)
    // testElement: int -> (int * int * int) option
    // Test an element to see whether it is a perfect square and a perfect
    // cube, and, if so, return the element, square root, and cube root
    // as an option value. Otherwise, return None.
    let testElement elem =
        if isPerfectSquare elem && isPerfectCube elem then
            Some(elem, intFunction sqrt elem, intFunction cubeRoot elem)
        else None
    match Array.tryPick testElement array1 with
    | Some (n, sqrt, cuberoot) -> printfn "Found an element %d with square root %d and cube root %d." n sqrt
cuberoot
    | None -> printfn "Did not find an element that is both a perfect square and a perfect cube."

findPerfectSquareAndCube [| 1 .. 10 |]
findPerfectSquareAndCube [| 2 .. 100 |]
findPerfectSquareAndCube [| 100 .. 1000 |]
findPerfectSquareAndCube [| 1000 .. 10000 |]
findPerfectSquareAndCube [| 2 .. 50 |]
```

The output is as follows.

```
Found an element 1 with square root 1 and cube root 1.
Found an element 64 with square root 8 and cube root 4.
Found an element 729 with square root 27 and cube root 9.
Found an element 4096 with square root 64 and cube root 16.
```

Performing Computations on Arrays

The `Array.average` function returns the average of each element in an array. It is limited to element types that support exact division by an integer, which includes floating point types but not integral types. The `Array.averageBy` function returns the average of the results of calling a function on each element. For an array of integral type, you can use `Array.averageBy` and have the function convert each element to a floating point type for the computation.

Use `Array.max` or `Array.min` to get the maximum or minimum element, if the element type supports it. Similarly, `Array.maxBy` and `Array.minBy` allow a function to be executed first, perhaps to transform to a type that supports comparison.

`Array.sum` adds the elements of an array, and `Array.sumBy` calls a function on each element and adds the results together.

To execute a function on each element in an array without storing the return values, use `Array.iter`. For a function involving two arrays of equal length, use `Array.iter2`. If you also need to keep an array of the results of the function, use `Array.map` or `Array.map2`, which operates on two arrays at a time.

The variations `Array.iteri` and `Array.iteri2` allow the index of the element to be involved in the computation; the same is true for `Array.mapi` and `Array.mapi2`.

The functions `Array.fold`, `Array.foldBack`, `Array.reduce`, `Array.reduceBack`, `Array.scan`, and `Array.scanBack`

execute algorithms that involve all the elements of an array. Similarly, the variations `Array.fold2` and `Array.foldBack2` perform computations on two arrays.

These functions for performing computations correspond to the functions of the same name in the List module. For usage examples, see Lists.

Modifying Arrays

`Array.set` sets an element to a specified value. `Array.fill` sets a range of elements in an array to a specified value. The following code provides an example of `Array.fill`.

```
let arrayFill1 = [| 1 .. 25 |]
Array.fill arrayFill1 2 20 0
printfn "%A" arrayFill1
```

The output is as follows.

```
[|1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 23; 24; 25|]
```

You can use `Array.blit` to copy a subsection of one array to another array.

Converting to and from Other Types

`Array.ofList` creates an array from a list. `Array.ofSeq` creates an array from a sequence. `Array.toList` and `Array.toSeq` convert to these other collection types from the array type.

Sorting Arrays

Use `Array.sort` to sort an array by using the generic comparison function. Use `Array.sortBy` to specify a function that generates a value, referred to as a *key*, to sort by using the generic comparison function on the key. Use `Array.sortWith` if you want to provide a custom comparison function. `Array.sort`, `Array.sortBy`, and `Array.sortWith` all return the sorted array as a new array. The variations `Array.sortInPlace`, `Array.sortInPlaceBy`, and `Array.sortInPlaceWith` modify the existing array instead of returning a new one.

Arrays and Tuples

The functions `Array.zip` and `Array.unzip` convert arrays of tuple pairs to tuples of arrays and vice versa. `Array.zip3` and `Array.unzip3` are similar except that they work with tuples of three elements or tuples of three arrays.

Parallel Computations on Arrays

The module `Array.Parallel` contains functions for performing parallel computations on arrays. This module is not available in applications that target versions of the .NET Framework prior to version 4.

See also

- F# Language Reference
- F# Types

Sequences

7/30/2019 • 18 minutes to read • Edit Online

NOTE

The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

A *sequence* is a logical series of elements all of one type. Sequences are particularly useful when you have a large, ordered collection of data but do not necessarily expect to use all of the elements. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used. Sequences are represented by the `seq<'T>` type, which is an alias for `System.Collections.Generic.IEnumerable`. Therefore, any .NET Framework type that implements `System.IEnumerable` can be used as a sequence. The Seq module provides support for manipulations involving sequences.

Sequence Expressions

A *sequence expression* is an expression that evaluates to a sequence. Sequence expressions can take a number of forms. The simplest form specifies a range. For example, `seq { 1 .. 5 }` creates a sequence that contains five elements, including the endpoints 1 and 5. You can also specify an increment (or decrement) between two double periods. For example, the following code creates the sequence of multiples of 10.

```
// Sequence that has an increment.
seq { 0 .. 10 .. 100 }
```

Sequence expressions are made up of F# expressions that produce values of the sequence. They can use the `yield` keyword to produce values that become part of the sequence.

Following is an example.

```
seq { for i in 1 .. 10 do yield i * i }
```

You can use the `->` operator instead of `yield`, in which case you can omit the `do` keyword, as shown in the following example.

```
seq { for i in 1 .. 10 -> i * i }
```

The following code generates a list of coordinate pairs along with an index into an array that represents the grid.

```
let (height, width) = (10, 10)
seq { for row in 0 .. width - 1 do
         for col in 0 .. height - 1 do
           yield (row, col, row*width + col)
    }
```

An `if` expression used in a sequence is a filter. For example, to generate a sequence of only prime numbers, assuming that you have a function `isprime` of type `int -> bool`, construct the sequence as follows.

```
seq { for n in 1 .. 100 do if isprime n then yield n }
```

When you use `yield` or `->` in an iteration, each iteration is expected to generate a single element of the sequence. If each iteration produces a sequence of elements, use `yield!`. In that case, the elements generated on each iteration are concatenated to produce the final sequence.

You can combine multiple expressions together in a sequence expression. The elements generated by each expression are concatenated together. For an example, see the "Examples" section of this topic.

Examples

The first example uses a sequence expression that contains an iteration, a filter, and a yield to generate an array. This code prints a sequence of prime numbers between 1 and 100 to the console.

```
// Recursive isprime function.
let isprime n =
    let rec check i =
        i > n/2 || (n % i <> 0 && check (i + 1))
    check 2

let aSequence = seq { for n in 1..100 do if isprime n then yield n }
for x in aSequence do
    printfn "%d" x
```

The following code uses `yield` to create a multiplication table that consists of tuples of three elements, each consisting of two factors and the product.

```
let multiplicationTable =
  seq { for i in 1..9 do
            for j in 1..9 do
               yield (i, j, i*j) }
```

The following example demonstrates the use of `yield!` to combine individual sequences into a single final sequence. In this case, the sequences for each subtree in a binary tree are concatenated in a recursive function to produce the final sequence.

```
// Yield the values of a binary tree in a sequence.
type Tree<'a> =
   | Tree of 'a * Tree<'a> * Tree<'a>
   | Leaf of 'a

// inorder : Tree<'a> -> seq<'a>
let rec inorder tree =
    seq {
      match tree with
          | Tree(x, left, right) ->
              yield! inorder left
              yield x
              yield! inorder right
          | Leaf x -> yield x
    }

let mytree = Tree(6, Tree(2, Leaf(1), Leaf(3)), Leaf(9))
let seq1 = inorder mytree
printfn "%A" seq1
```

Using Sequences

Sequences support many of the same functions as lists. Sequences also support operations such as grouping and counting by using key-generating functions. Sequences also support more diverse functions for extracting subsequences.

Many data types, such as lists, arrays, sets, and maps are implicitly sequences because they are enumerable collections. A function that takes a sequence as an argument works with any of the common F# data types, in addition to any .NET Framework data type that implements `System.Collections.Generic.IEnumerable<'T>`. Contrast this to a function that takes a list as an argument, which can only take lists. The type `seq<'T>` is a type abbreviation for `IEnumerable<'T>`. This means that any type that implements the generic `System.Collections.Generic.IEnumerable<'T>`, which includes arrays, lists, sets, and maps in F#, and also most .NET Framework collection types, is compatible with the `seq` type and can be used wherever a sequence is expected.

Module Functions

The Seq module in the Microsoft.FSharp.Collections namespace contains functions for working with sequences. These functions work with lists, arrays, maps, and sets as well, because all of those types are enumerable, and therefore can be treated as sequences.

Creating Sequences

You can create sequences by using sequence expressions, as described previously, or by using certain functions.

You can create an empty sequence by using Seq.empty, or you can create a sequence of just one specified element by using Seq.singleton.

```
let seqEmpty = Seq.empty
let seqOne = Seq.singleton 10
```

You can use Seq.init to create a sequence for which the elements are created by using a function that you provide. You also provide a size for the sequence. This function is just like List.init, except that the elements are not created until you iterate through the sequence. The following code illustrates the use of `Seq.init`.

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

The output is

```
0 10 20 30 40
```

By using Seq.ofArray and Seq.ofList<'T> Function, you can create sequences from arrays and lists. However, you can also convert arrays and lists to sequences by using a cast operator. Both techniques are shown in the following code.

```
// Convert an array to a sequence by using a cast.
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>
// Convert an array to a sequence by using Seq.ofArray.
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

By using Seq.cast, you can create a sequence from a weakly typed collection, such as those defined in `System.Collections`. Such weakly typed collections have the element type `System.Object` and are enumerated

by using the non-generic `System.Collections.Generic.IEnumerable`1` type. The following code illustrates the use of `Seq.cast` to convert an `System.Collections.ArrayList` into a sequence.

```
open System
let mutable arrayList1 = new System.Collections.ArrayList(10)
for i in 1 .. 10 do arrayList1.Add(10) |> ignore
let seqCast : seq<int> = Seq.cast arrayList1
```

You can define infinite sequences by using the Seq.initInfinite function. For such a sequence, you provide a function that generates each element from the index of the element. Infinite sequences are possible because of lazy evaluation; elements are created as needed by calling the function that you specify. The following code example produces an infinite sequence of floating point numbers, in this case the alternating series of reciprocals of squares of successive integers.

```
let seqInfinite = Seq.initInfinite (fun index ->
    let n = float( index + 1 )
    1.0 / (n * n * (if ((index + 1) % 2 = 0) then 1.0 else -1.0)))
printfn "%A" seqInfinite
```

Seq.unfold generates a sequence from a computation function that takes a state and transforms it to produce each subsequent element in the sequence. The state is just a value that is used to compute each element, and can change as each element is computed. The second argument to `Seq.unfold` is the initial value that is used to start the sequence. `Seq.unfold` uses an option type for the state, which enables you to terminate the sequence by returning the `None` value. The following code shows two examples of sequences, `seq1` and `fib`, that are generated by an `unfold` operation. The first, `seq1`, is just a simple sequence with numbers up to 20. The second, `fib`, uses `unfold` to compute the Fibonacci sequence. Because each element in the Fibonacci sequence is the sum of the previous two Fibonacci numbers, the state value is a tuple that consists of the previous two numbers in the sequence. The initial value is `(1,1)`, the first two numbers in the sequence.

```
let seq1 = Seq.unfold (fun state -> if (state > 20) then None else Some(state, state + 1)) 0
printfn "The sequence seq1 contains numbers from 0 to 20."
for x in seq1 do printf "%d " x
let fib = Seq.unfold (fun state ->
    if (snd state > 1000) then None
    else Some(fst state + snd state, (snd state, fst state + snd state))) (1,1)
printfn "\nThe sequence fib contains Fibonacci numbers."
for x in fib do printf "%d " x
```

The output is as follows:

```
The sequence seq1 contains numbers from 0 to 20.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The sequence fib contains Fibonacci numbers.

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The following code is an example that uses many of the sequence module functions described here to generate and compute the values of infinite sequences. The code might take a few minutes to run.

```
// infiniteSequences.fs
// generateInfiniteSequence generates sequences of floating point
// numbers. The sequences generated are computed from the fDenominator
// function, which has the type (int -> float) and computes the
// denominator of each term in the sequence from the index of that
// term. The isAlternating parameter is true if the sequence has
// alternating signs.
let generateInfiniteSequence fDenominator isAlternating =
    if (isAlternating) then
        Seq.initInfinite (fun index -> 1.0 /(fDenominator index) * (if (index % 2 = 0) then -1.0 else 1.0))
    else
        Seq.initInfinite (fun index -> 1.0 /(fDenominator index))

// The harmonic series is the series of reciprocals of whole numbers.
let harmonicSeries = generateInfiniteSequence (fun index -> float index) false
// The harmonic alternating series is like the harmonic series
// except that it has alternating signs.
let harmonicAlternatingSeries = generateInfiniteSequence (fun index -> float index) true
// This is the series of reciprocals of the odd numbers.
let oddNumberSeries = generateInfiniteSequence (fun index -> float (2 * index - 1)) true
// This is the series of recipocals of the squares.
let squaresSeries = generateInfiniteSequence (fun index -> float (index * index)) false

// This function sums a sequence, up to the specified number of terms.
let sumSeq length sequence =
    Seq.unfold (fun state ->
        let subtotal = snd state + Seq.nth (fst state + 1) sequence
        if (fst state >= length) then None
        else Some(subtotal,(fst state + 1, subtotal))) (0, 0.0)

// This function sums an infinite sequence up to a given value
// for the difference (epsilon) between subsequent terms,
// up to a maximum number of terms, whichever is reached first.
let infiniteSum infiniteSeq epsilon maxIteration =
    infiniteSeq
    |> sumSeq maxIteration
    |> Seq.pairwise
    |> Seq.takeWhile (fun elem -> abs (snd elem - fst elem) > epsilon)
    |> List.ofSeq
    |> List.rev
    |> List.head
    |> snd

// Compute the sums for three sequences that converge, and compare
// the sums to the expected theoretical values.
let result1 = infiniteSum harmonicAlternatingSeries 0.00001 100000
printfn "Result: %f  ln2: %f" result1 (log 2.0)

let pi = Math.PI
let result2 = infiniteSum oddNumberSeries 0.00001 10000
printfn "Result: %f pi/4: %f" result2 (pi/4.0)

// Because this is not an alternating series, a much smaller epsilon
// value and more terms are needed to obtain an accurate result.
let result3 = infiniteSum squaresSeries 0.0000001 1000000
printfn "Result: %f pi*pi/6: %f" result3 (pi*pi/6.0)
```

Searching and Finding Elements

Sequences support functionality available with lists: Seq.exists, Seq.exists2, Seq.find, Seq.findIndex, Seq.pick, Seq.tryFind, and Seq.tryFindIndex. The versions of these functions that are available for sequences evaluate the sequence only up to the element that is being searched for. For examples, see Lists.

Obtaining Subsequences

Seq.filter and Seq.choose are like the corresponding functions that are available for lists, except that the filtering and choosing does not occur until the sequence elements are evaluated.

Seq.truncate creates a sequence from another sequence, but limits the sequence to a specified number of elements. Seq.take creates a new sequence that contains only a specified number of elements from the start of a sequence. If there are fewer elements in the sequence than you specify to take, `Seq.take` throws a `System.InvalidOperationException`. The difference between `Seq.take` and `Seq.truncate` is that `Seq.truncate` does not produce an error if the number of elements is fewer than the number you specify.

The following code shows the behavior of and differences between `Seq.truncate` and `Seq.take`.

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let truncatedSeq2 = Seq.truncate 20 mySeq
let takenSeq2 = Seq.take 20 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""

// Up to this point, the sequences are not evaluated.
// The following code causes the sequences to be evaluated.
truncatedSeq |> printSeq
truncatedSeq2 |> printSeq
takenSeq |> printSeq
// The following line produces a run-time error (in printSeq):
takenSeq2 |> printSeq
```

The output, before the error occurs, is as follows.

```
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
```

By using Seq.takeWhile, you can specify a predicate function (a Boolean function) and create a sequence from another sequence made up of those elements of the original sequence for which the predicate is `true`, but stop before the first element for which the predicate returns `false`. Seq.skip returns a sequence that skips a specified number of the first elements of another sequence and returns the remaining elements. Seq.skipWhile returns a sequence that skips the first elements of another sequence as long as the predicate returns `true`, and then returns the remaining elements, starting with the first element for which the predicate returns `false`.

The following code example illustrates the behavior of and differences between `Seq.takeWhile`, `Seq.skip`, and `Seq.skipWhile`.

```
// takeWhile
let mySeqLessThan10 = Seq.takeWhile (fun elem -> elem < 10) mySeq
mySeqLessThan10 |> printSeq

// skip
let mySeqSkipFirst5 = Seq.skip 5 mySeq
mySeqSkipFirst5 |> printSeq

// skipWhile
let mySeqSkipWhileLessThan10 = Seq.skipWhile (fun elem -> elem < 10) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

The output is as follows.

```
1 4 9
36 49 64 81 100
16 25 36 49 64 81 100
```

Transforming Sequences

Seq.pairwise creates a new sequence in which successive elements of the input sequence are grouped into tuples.

```
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqPairwise = Seq.pairwise (seq { for i in 1 .. 10 -> i*i })
printSeq seqPairwise

printfn ""
let seqDelta = Seq.map (fun elem -> snd elem - fst elem) seqPairwise
printSeq seqDelta
```

Seq.windowed is like `Seq.pairwise`, except that instead of producing a sequence of tuples, it produces a sequence of arrays that contain copies of adjacent elements (a *window*) from the sequence. You specify the number of adjacent elements you want in each array.

The following code example demonstrates the use of `Seq.windowed`. In this case the number of elements in the window is 3. The example uses `printSeq`, which is defined in the previous code example.

```
let seqNumbers = [ 1.0; 1.5; 2.0; 1.5; 1.0; 1.5 ] :> seq<float>
let seqWindows = Seq.windowed 3 seqNumbers
let seqMovingAverage = Seq.map Array.average seqWindows
printfn "Initial sequence: "
printSeq seqNumbers
printfn "\nWindows of length 3: "
printSeq seqWindows
printfn "\nMoving average: "
printSeq seqMovingAverage
```

The output is as follows.

Initial sequence:

```
1.0 1.5 2.0 1.5 1.0 1.5

Windows of length 3:
[|1.0; 1.5; 2.0|] [|1.5; 2.0; 1.5|] [|2.0; 1.5; 1.0|] [|1.5; 1.0; 1.5|]

Moving average:
1.5 1.666666667 1.5 1.333333333
```

Operations with Multiple Sequences

Seq.zip and Seq.zip3 take two or three sequences and produce a sequence of tuples. These functions are like the corresponding functions available for lists. There is no corresponding functionality to separate one sequence into two or more sequences. If you need this functionality for a sequence, convert the sequence to a list and use List.unzip.

Sorting, Comparing, and Grouping

The sorting functions supported for lists also work with sequences. This includes Seq.sort and Seq.sortBy. These functions iterate through the whole sequence.

You compare two sequences by using the Seq.compareWith function. The function compares successive elements in turn, and stops when it encounters the first unequal pair. Any additional elements do not contribute to the comparison.

The following code shows the use of `Seq.compareWith`.

```
let sequence1 = seq { 1 .. 10 }
let sequence2 = seq { 10 .. -1 .. 1 }

// Compare two sequences element by element.
let compareSequences = Seq.compareWith (fun elem1 elem2 ->
    if elem1 > elem2 then 1
    elif elem1 < elem2 then -1
    else 0)

let compareResult1 = compareSequences sequence1 sequence2
match compareResult1 with
| 1 -> printfn "Sequence1 is greater than sequence2."
| -1 -> printfn "Sequence1 is less than sequence2."
| 0 -> printfn "Sequence1 is equal to sequence2."
| _ -> failwith("Invalid comparison result.")
```

In the previous code, only the first element is computed and examined, and the result is -1.

Seq.countBy takes a function that generates a value called a *key* for each element. A key is generated for each element by calling this function on each element. `Seq.countBy` then returns a sequence that contains the key values, and a count of the number of elements that generated each value of the key.

```
let mySeq1 = seq { 1.. 100 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqResult = Seq.countBy (fun elem -> if elem % 3 = 0 then 0
                                         elif elem % 3 = 1 then 1
                                         else 2) mySeq1

printSeq seqResult
```

The output is as follows.

```
(1, 34) (2, 33) (0, 33)
```

The previous output shows that there were 34 elements of the original sequence that produced the key 1, 33 values that produced the key 2, and 33 values that produced the key 0.

You can group elements of a sequence by calling Seq.groupBy. `Seq.groupBy` takes a sequence and a function that generates a key from an element. The function is executed on each element of the sequence. `Seq.groupBy` returns a sequence of tuples, where the first element of each tuple is the key and the second is a sequence of elements that produce that key.

The following code example shows the use of `Seq.groupBy` to partition the sequence of numbers from 1 to 100 into three groups that have the distinct key values 0, 1, and 2.

```
let sequence = seq { 1 .. 100 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let sequences3 = Seq.groupBy (fun index ->
                                if (index % 3 = 0) then 0
                                  elif (index % 3 = 1) then 1
                                  else 2) sequence
sequences3 |> printSeq
```

The output is as follows.

```
(1, seq [1; 4; 7; 10; ...]) (2, seq [2; 5; 8; 11; ...]) (0, seq [3; 6; 9; 12; ...])
```

You can create a sequence that eliminates duplicate elements by calling Seq.distinct. Or you can use Seq.distinctBy, which takes a key-generating function to be called on each element. The resulting sequence contains elements of the original sequence that have unique keys; later elements that produce a duplicate key to an earlier element are discarded.

The following code example illustrates the use of `Seq.distinct`. `Seq.distinct` is demonstrated by generating sequences that represent binary numbers, and then showing that the only distinct elements are 0 and 1.

```
let binary n =
    let rec generateBinary n =
        if (n / 2 = 0) then [n]
        else (n % 2) :: generateBinary (n / 2)
    generateBinary n |> List.rev |> Seq.ofList

printfn "%A" (binary 1024)

let resultSequence = Seq.distinct (binary 1024)
printfn "%A" resultSequence
```

The following code demonstrates `Seq.distinctBy` by starting with a sequence that contains negative and positive numbers and using the absolute value function as the key-generating function. The resulting sequence is missing all the positive numbers that correspond to the negative numbers in the sequence, because the negative numbers appear earlier in the sequence and therefore are selected instead of the positive numbers that have the same absolute value, or key.

```
let inputSequence = { -5 .. 10 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
printfn "Original sequence: "
printSeq inputSequence
printfn "\nSequence with distinct absolute values: "
let seqDistinctAbsoluteValue = Seq.distinctBy (fun elem -> abs elem) inputSequence
seqDistinctAbsoluteValue |> printSeq
```

Readonly and Cached Sequences

Seq.readonly creates a read-only copy of a sequence. `Seq.readonly` is useful when you have a read-write collection, such as an array, and you do not want to modify the original collection. This function can be used to preserve data encapsulation. In the following code example, a type that contains an array is created. A property exposes the array, but instead of returning an array, it returns a sequence that is created from the array by using `Seq.readonly`.

```
type ArrayContainer(start, finish) =
    let internalArray = [| start .. finish |]
    member this.RangeSeq = Seq.readonly internalArray
    member this.RangeArray = internalArray

let newArray = new ArrayContainer(1, 10)
let rangeSeq = newArray.RangeSeq
let rangeArray = newArray.RangeArray
// These lines produce an error:
//let myArray = rangeSeq :> int array
//myArray.[0] <- 0
// The following line does not produce an error.
// It does not preserve encapsulation.
rangeArray.[0] <- 0
```

Seq.cache creates a stored version of a sequence. Use `Seq.cache` to avoid reevaluation of a sequence, or when you have multiple threads that use a sequence, but you must make sure that each element is acted upon only one time. When you have a sequence that is being used by multiple threads, you can have one thread that enumerates and computes the values for the original sequence, and remaining threads can use the cached sequence.

Performing Computations on Sequences

Simple arithmetic operations are like those of lists, such as Seq.average, Seq.sum, Seq.averageBy, Seq.sumBy, and so on.

Seq.fold, Seq.reduce, and Seq.scan are like the corresponding functions that are available for lists. Sequences support a subset of the full variations of these functions that lists support. For more information and examples, see Lists.

See also

- F# Language Reference
- F# Types

Slices

7/30/2019 • 3 minutes to read • Edit Online

In F#, a slice is a subset of a data type. To be able to take a slice from a data type, the data type must either define a `GetSlice` method or in a type extension that is in scope. This article explains how to take slices from existing F# types and how to define your own.

Slices are similar to indexers, but instead of yielding a single value from the underlying data structure, they yield multiple ones.

F# currently has intrinsic support for slicing strings, lists, arrays, and 2D arrays.

Basic slicing with F# lists and arrays

The most common data types that are sliced are F# lists and arrays. The following example demonstrates how to do this with lists:

```
// Generate a list of 100 integers
let fullList = [ 1 .. 100 ]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullList.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullList.[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullList.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

Slicing arrays is just like slicing lists:

```
// Generate an array of 100 integers
let fullArray = [| 1 .. 100 |]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullArray.[1..5]
printfn "Small slice: %A" smallSlice

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullArray.[..5]
printfn "Unbounded beginning slice: %A" unboundedBeginning

// Create a slice from an index to the end of the list
let unboundedEnd = fullArray.[94..]
printfn "Unbounded end slice: %A" unboundedEnd
```

Slicing multidimensional arrays

F# supports multidimensional arrays in the F# core library. As with one-dimensional arrays, slices of multidimensional arrays can also be useful. However, the introduction of additional dimensions mandates a slightly different syntax so that you can take slices of specific rows and columns.

The following examples demonstrate how to slice a 2D array:

```
// Generate a 3x3 2D matrix
let A = array2D [[1;2;3];[4;5;6];[7;8;9]]
printfn "Full matrix:\n %A" A

// Take the first row
let row0 = A.[0,*]
printfn "Row 0: %A" row0

// Take the first column
let col0 = A.[*,0]
printfn "Column 0: %A" col0

// Take all rows but only two columns
let subA = A.[*,0..1]
printfn "%A" subA

// Take two rows and all columns
let subA' = A.[0..1,*]
printfn "%A" subA'

// Slice a 2x2 matrix out of the full 3x3 matrix
let twoByTwo = A.[0..1,0..1]
printfn "%A" twoByTwo
```

The F# core library does not define `GetSlice` for 3D arrays. If you wish to slice those or other arrays of more dimensions, you must define the `GetSlice` member yourself.

Defining slices for other data structures

The F# core library defines slices for a limited set of types. If you wish to define slices for more data types, you can do so either in the type definition itself or in a type extension.

For example, here's how you might define slices for the ArraySegment<T> class to allow for convenient data manipulation:

```
open System

type ArraySegment<'TItem> with
    member segment.GetSlice(?start, ?finish) =
        let start = defaultArg start 0
        let finish = defaultArg finish segment.Count
        ArraySegment(segment.Array, segment.Offset + start, finish - start)

let arr = ArraySegment [| 1 .. 10 |]
let slice = arr.[2..5] //[ 3; 4; 5]
```

Use inlining to avoid boxing if it is necessary

If you are defining slices for a type that is actually a struct, we recommend that you `inline` the `GetSlice` member. The F# compiler optimizes away the optional arguments, avoiding any heap allocations as a result of slicing. This is critically important for slicing constructs such as Span<T> that cannot be allocated on the heap.

```
open System

type Span<'T> with
    // Note the 'inline' in the member definition
    member inline sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e)

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn "%A" arr

let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
printSpan sp.[0..] // [|1; 2; 3; 4; 5|]
printSpan sp.[..5] // [|1; 2; 3; 4; 5|]
printSpan sp.[0..3] // [|1; 2; 3|]
printSpan sp.[1..2] // |2; 3|]
```

See also

- Indexed properties

Options

7/30/2019 • 4 minutes to read • Edit Online

The option type in F# is used when an actual value might not exist for a named value or variable. An option has an underlying type and can hold a value of that type, or it might not have a value.

Remarks

The following code illustrates a function which generates an option type.

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

As you can see, if the input `a` is greater than 0, `Some(a)` is generated. Otherwise, `None` is generated.

The value `None` is used when an option does not have an actual value. Otherwise, the expression `Some( ... )` gives the option a value. The values `Some` and `None` are useful in pattern matching, as in the following function `exists`, which returns `true` if the option has a value and `false` if it does not.

```
let exists (x : int option) =
    match x with
    | Some(x) -> true
    | None -> false
```

Using Options

Options are commonly used when a search does not return a matching result, as shown in the following code.

```
let rec tryFindMatch pred list =
    match list with
    | head :: tail -> if pred(head)
                      then Some(head)
                      else tryFindMatch pred tail
    | [] -> None

// result1 is Some 100 and its type is int option.
let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ]

// result2 is None and its type is int option.
let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]
```

In the previous code, a list is searched recursively. The function `tryFindMatch` takes a predicate function `pred` that returns a Boolean value, and a list to search. If an element that satisfies the predicate is found, the recursion ends and the function returns the value as an option in the expression `Some(head)`. The recursion ends when the empty list is matched. At that point the value `head` has not been found, and `None` is returned.

Many F# library functions that search a collection for a value that may or may not exist return the `option` type. By convention, these functions begin with the `try` prefix, for example, `Seq.tryFindIndex`.

Options can also be useful when a value might not exist, for example if it is possible that an exception will be thrown when you try to construct a value. The following code example illustrates this.

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
                None
```

The `openFile` function in the previous example has type `string -> File option` because it returns a `File` object if the file opens successfully and `None` if an exception occurs. Depending on the situation, it may not be an appropriate design choice to catch an exception rather than allowing it to propagate.

Additionally, it is still possible to pass `null` or a value that is null to the `Some` case of an option. This is generally to be avoided, and typically is in routine F# programming, but is possible due to the nature of reference types in .NET.

Option Properties and Methods

The option type supports the following properties and methods.

PROPERTY OR METHOD	TYPE	DESCRIPTION
None	`'T option`	A static property that enables you to create an option value that has the `None` value.
IsNone	`bool`	Returns `true` if the option has the `None` value.
IsSome	`bool`	Returns `true` if the option has a value that is not `None`.
Some	`'T option`	A static member that creates an option that has a value that is not `None`.
Value	`'T`	Returns the underlying value, or throws a `System.NullReferenceException` if the value is `None`.

Option Module

There is a module, Option, that contains useful functions that perform operations on options. Some functions repeat the functionality of the properties but are useful in contexts where a function is needed. Option.isSome and Option.isNone are both module functions that test whether an option holds a value. Option.get obtains the value, if there is one. If there is no value, it throws `System.ArgumentException`.

The Option.bind function executes a function on the value, if there is a value. The function must take exactly one argument, and its parameter type must be the option type. The return value of the function is another option type.

The option module also includes functions that correspond to the functions that are available for lists, arrays, sequences, and other collection types. These functions include `Option.map`, `Option.iter`, `Option.forall`, `Option.exists`, `Option.foldBack`, `Option.fold`, and `Option.count`. These functions enable options to be used like a collection of zero or one elements. For more information and examples, see the discussion of collection

functions in Lists.

Converting to Other Types

Options can be converted to lists or arrays. When an option is converted into either of these data structures, the resulting data structure has zero or one element. To convert an option to an array, use `Option.toArray`. To convert an option to a list, use `Option.toList`.

See also

- F# Language Reference
- F# Types

Value Options

2/7/2019 • 2 minutes to read • Edit Online

The Value Option type in F# is used when the following two circumstances hold:

1. A scenario is appropriate for an F# Option.
2. Using a struct provides a performance benefit in your scenario.

Not all performance-sensitive scenarios are "solved" by using structs. You must consider the additional cost of copying when using them instead of reference types. However, large F# programs commonly instantiate many optional types that flow through hot paths, and in such cases, structs can often yield better overall performance over the lifetime of a program.

Definition

Value Option is defined as a struct discriminated union that is similar to the reference option type. Its definition can be thought of this way:

```
[<StructuralEquality; StructuralComparison>]
[<Struct>]
type ValueOption<'T> =
    | ValueNone
    | ValueSome of 'T
```

Value Option conforms to structural equality and comparison. The main difference is that the compiled name, type name, and case names all indicate that it is a value type.

Using Value Options

Value Options are used just like Options. `ValueSome` is used to indicate that a value is present, and `ValueNone` is used when a value is not present:

```
let tryParseDateTime (s: string) =
    match System.DateTime.TryParse(s) with
    | (true, dt) -> ValueSome dt
    | (false, _) -> ValueNone

let possibleDateString1 = "1990-12-25"
let possibleDateString2 = "This is not a date"

let result1 = tryParseDateTime possibleDateString1
let result2 = tryParseDateTime possibleDateString2

match (result1, result2) with
| ValueSome d1, ValueSome d2 -> printfn "Both are dates!"
| ValueSome d1, ValueNone -> printfn "Only the first is a date!"
| ValueNone, ValueSome d2 -> printfn "Only the second is a date!"
| ValueNone, ValueNone -> printfn "None of them are dates!"
```

As with Options, the naming convention for a function that returns `ValueOption` is to prefix it with `try`.

Value Option properties and methods

There is one property for Value Options at this time: `Value`. An InvalidOperationException is raised if no value is present when this property is invoked.

Value Option functions

There is currently one module-bound function for Value Options, `defaultValueArg`:

```
val defaultValueArg : arg:'T voption -> defaultValue:'T -> 'T
```

As with the `defaultArg` function, `defaultValueArg` returns the underlying value of the given Value Option if it exists; otherwise, it returns the specified default value.

At this time, there are no other module-bound functions for Value Options.

See also

- Options

Results

5/15/2019 • 2 minutes to read • Edit Online

Starting with F# 4.1, there is a `Result<'T,'TFailure>` type which you can use for writing error-tolerant code which can be composed.

Syntax

```
// The definition of Result in FSharp.Core
[<StructuralEquality; StructuralComparison>]
[<CompiledName("FSharpResult`2")>]
[<Struct>]
type Result<'T,'TError> =
    | Ok of ResultValue:'T
    | Error of ErrorValue:'TError
```

Remarks

Note that the result type is a struct discriminated union, which is another feature introduced in F# 4.1. Structural equality semantics apply here.

The `Result` type is typically used in monadic error-handling, which is often referred to as Railway-oriented Programming within the F# community. The following trivial example demonstrates this approach.

```
// Define a simple type which has fields that can be validated
type Request =
    { Name: string
      Email: string }

// Define some logic for what defines a valid name.
//
// Generates a Result which is an Ok if the name validates;
// otherwise, it generates a Result which is an Error.
let validateName req =
    match req.Name with
    | null -> Error "No name found."
    | "" -> Error "Name is empty."
    | "bananas" -> Error "Bananas is not a name."
    | _ -> Ok req

// Similarly, define some email validation logic.
let validateEmail req =
    match req.Email with
    | null -> Error "No email found."
    | "" -> Error "Email is empty."
    | s when s.EndsWith("bananas.com") -> Error "No email from bananas.com is allowed."
    | _ -> Ok req

let validateRequest reqResult =
    reqResult
    |> Result.bind validateName
    |> Result.bind validateEmail

let test() =
    // Now, create a Request and pattern match on the result.
    let req1 = { Name = "Phillip"; Email = "phillip@contoso.biz" }
    let res1 = validateRequest (Ok req1)
    match res1 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "My request was valid!  Name: Phillip Email: phillip@consoto.biz"

    let req2 = { Name = "Phillip"; Email = "phillip@bananas.com" }
    let res2 = validateRequest (Ok req2)
    match res2 with
    | Ok req -> printfn "My request was valid! Name: %s Email %s" req.Name req.Email
    | Error e -> printfn "Error: %s" e
    // Prints: "Error: No email from bananas.com is allowed."

test()
```

As you can see, it's quite easy to chain together various validation functions if you force them all to return a `Result`. This lets you break up functionality like this into small pieces which are as composable as you need them to be. This also has the added value of *enforcing* the use of pattern matching at the end of a round of validation, which in turns enforces a higher degree of program correctness.

See also

- Discriminated Unions
- Pattern Matching

Generics

8/20/2019 • 6 minutes to read • Edit Online

F# function values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be *generic*. Generic constructs contain at least one type parameter, which is usually supplied by the user of the generic construct. Generic functions and types enable you to write code that works with a variety of types without repeating the code for each type. Making your code generic can be simple in F#, because often your code is implicitly inferred to be generic by the compiler's type inference and automatic generalization mechanisms.

Syntax

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
function-body

// Explicitly generic method.
[ static ] member object-identifer.method-name<type-parameters> parameter-list [ return-type ] =
method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

Remarks

The declaration of an explicitly generic function or type is much like that of a non-generic function or type, except for the specification (and use) of the type parameters, in angle brackets after the function or type name.

Declarations are often implicitly generic. If you do not fully specify the type of every parameter that is used to compose a function or type, the compiler attempts to infer the type of each parameter, value, and variable from the code you write. For more information, see Type Inference. If the code for your type or function does not otherwise constrain the types of parameters, the function or type is implicitly generic. This process is named *automatic generalization*. There are some limits on automatic generalization. For example, if the F# compiler is unable to infer the types for a generic construct, the compiler reports an error that refers to a restriction called the *value restriction*. In that case, you may have to add some type annotations. For more information about automatic generalization and the value restriction, and how to change your code to address the problem, see Automatic Generalization.

In the previous syntax, *type-parameters* is a comma-separated list of parameters that represent unknown types, each of which starts with a single quotation mark, optionally with a constraint clause that further limits what types may be used for that type parameter. For the syntax for constraint clauses of various kinds and other information about constraints, see Constraints.

The *type-definition* in the syntax is the same as the type definition for a non-generic type. It includes the constructor parameters for a class type, an optional `as` clause, the equal symbol, the record fields, the `inherit` clause, the choices for a discriminated union, `let` and `do` bindings, member definitions, and anything else permitted in a non-generic type definition.

The other syntax elements are the same as those for non-generic functions and types. For example, *object-identifier* is an identifier that represents the containing object itself.

Properties, fields, and constructors cannot be more generic than the enclosing type. Also, values in a module

cannot be generic.

Implicitly Generic Constructs

When the F# compiler infers the types in your code, it automatically treats any function that can be generic as generic. If you specify a type explicitly, such as a parameter type, you prevent automatic generalization.

In the following code example, `makeList` is generic, even though neither it nor its parameters are explicitly declared as generic.

```
let makeList a b =
    [a; b]
```

The signature of the function is inferred to be `'a -> 'a -> 'a list`. Note that `a` and `b` in this example are inferred to have the same type. This is because they are included in a list together, and all elements of a list must be of the same type.

You can also make a function generic by using the single quotation mark syntax in a type annotation to indicate that a parameter type is a generic type parameter. In the following code, `function1` is generic because its parameters are declared in this manner, as type parameters.

```
let function1 (x: 'a) (y: 'a) =
    printfn "%A %A" x y
```

Explicitly Generic Constructs

You can also make a function generic by explicitly declaring its type parameters in angle brackets (`<type-parameter>`). The following code illustrates this.

```
let function2<'T> x y =
    printfn "%A, %A" x y
```

Using Generic Constructs

When you use generic functions or methods, you might not have to specify the type arguments. The compiler uses type inference to infer the appropriate type arguments. If there is still an ambiguity, you can supply type arguments in angle brackets, separating multiple type arguments with commas.

The following code shows the use of the functions that are defined in the previous sections.

```
// In this case, the type argument is inferred to be int.
function1 10 20
// In this case, the type argument is float.
function1 10.0 20.0
// Type arguments can be specified, but should only be specified
// if the type parameters are declared explicitly. If specified,
// they have an effect on type inference, so in this example,
// a and b are inferred to have type int.
let function3 a b =
    // The compiler reports a warning:
    function1<int> a b
    // No warning.
    function2<int> a b
```

NOTE

There are two ways to refer to a generic type by name. For example, `list<int>` and `int list` are two ways to refer to a generic type `list` that has a single type argument `int`. The latter form is conventionally used only with built-in F# types such as `list` and `option`. If there are multiple type arguments, you normally use the syntax `Dictionary<int, string>` but you can also use the syntax `(int, string) Dictionary`.

Wildcards as Type Arguments

To specify that a type argument should be inferred by the compiler, you can use the underscore, or wildcard symbol (`_`), instead of a named type argument. This is shown in the following code.

```
let printSequence (sequence1: Collections.seq<_>) =
   Seq.iter (fun elem -> printf "%s " (elem.ToString())) sequence1
```

Constraints in Generic Types and Functions

In a generic type or function definition, you can use only those constructs that are known to be available on the generic type parameter. This is required to enable the verification of function and method calls at compile time. If you declare your type parameters explicitly, you can apply an explicit constraint to a generic type parameter to notify the compiler that certain methods and functions are available. However, if you allow the F# compiler to infer your generic parameter types, it will determine the appropriate constraints for you. For more information, see Constraints.

Statically Resolved Type Parameters

There are two kinds of type parameters that can be used in F# programs. The first are generic type parameters of the kind described in the previous sections. This first kind of type parameter is equivalent to the generic type parameters that are used in languages such as Visual Basic and C#. Another kind of type parameter is specific to F# and is referred to as a *statically resolved type parameter*. For information about these constructs, see Statically Resolved Type Parameters.

Examples

```
// A generic function.
// In this example, the generic type parameter 'a makes function3 generic.
let function3 (x : 'a) (y : 'a) =
    printf "%A %A" x y

// A generic record, with the type parameter in angle brackets.
type GR<'a> =
    {
        Field1: 'a;
        Field2: 'a;
    }

// A generic class.
type C<'a>(a : 'a, b : 'a) =
    let z = a
    let y = b
    member this.GenericMethod(x : 'a) =
        printfn "%A %A %A" x y z

// A generic discriminated union.
type U<'a> =
    | Choice1 of 'a
    | Choice2 of 'a * 'a

type Test() =
    // A generic member
    member this.Function1<'a>(x, y) =
        printfn "%A, %A" x y

    // A generic abstract method.
    abstract abstractMethod<'a, 'b> : 'a * 'b -> unit
    override this.abstractMethod<'a, 'b>(x:'a, y:'b) =
         printfn "%A, %A" x y
```

See also

- Language Reference
- Types
- Statically Resolved Type Parameters
- Generics
- Automatic Generalization
- Constraints

Automatic Generalization

7/30/2019 • 3 minutes to read • Edit Online

F# uses type inference to evaluate the types of functions and expressions. This topic describes how F# automatically generalizes the arguments and types of functions so that they work with multiple types when this is possible.

Automatic Generalization

The F# compiler, when it performs type inference on a function, determines whether a given parameter can be generic. The compiler examines each parameter and determines whether the function has a dependency on the specific type of that parameter. If it does not, the type is inferred to be generic.

The following code example illustrates a function that the compiler infers to be generic.

```
let max a b = if a > b then a else b
```

The type is inferred to be `'a -> 'a -> 'a`.

The type indicates that this is a function that takes two arguments of the same unknown type and returns a value of that same type. One of the reasons that the previous function can be generic is that the greater-than operator (`>`) is itself generic. The greater-than operator has the signature `'a -> 'a -> bool`. Not all operators are generic, and if the code in a function uses a parameter type together with a non-generic function or operator, that parameter type cannot be generalized.

Because `max` is generic, it can be used with types such as `int`, `float`, and so on, as shown in the following examples.

```
let biggestFloat = max 2.0 3.0
let biggestInt = max 2 3
```

However, the two arguments must be of the same type. The signature is `'a -> 'a -> 'a`, not `'a -> 'b -> 'a`. Therefore, the following code produces an error because the types do not match.

```
// Error: type mismatch.
let biggestIntFloat = max 2.0 3
```

The `max` function also works with any type that supports the greater-than operator. Therefore, you could also use it on a string, as shown in the following code.

```
let testString = max "cab" "cat"
```

Value Restriction

The compiler performs automatic generalization only on complete function definitions that have explicit arguments, and on simple immutable values.

This means that the compiler issues an error if you try to compile code that is not sufficiently constrained to be a specific type, but is also not generalizable. The error message for this problem refers to this restriction on

automatic generalization for values as the *value restriction*.

Typically, the value restriction error occurs either when you want a construct to be generic but the compiler has insufficient information to generalize it, or when you unintentionally omit sufficient type information in a nongeneric construct. The solution to the value restriction error is to provide more explicit information to more fully constrain the type inference problem, in one of the following ways:

- Constrain a type to be nongeneric by adding an explicit type annotation to a value or parameter.
- If the problem is using a nongeneralizable construct to define a generic function, such as a function composition or incompletely applied curried function arguments, try to rewrite the function as an ordinary function definition.
- If the problem is an expression that is too complex to be generalized, make it into a function by adding an extra, unused parameter.
- Add explicit generic type parameters. This option is rarely used.
- The following code examples illustrate each of these scenarios.

Case 1: Too complex an expression. In this example, the list `counter` is intended to be `int option ref`, but it is not defined as a simple immutable value.

```
let counter = ref None
// Adding a type annotation fixes the problem:
let counter : int option ref = ref None
```

Case 2: Using a nongeneralizable construct to define a generic function. In this example, the construct is nongeneralizable because it involves partial application of function arguments.

```
let maxhash = max << hash
// The following is acceptable because the argument for maxhash is explicit:
let maxhash obj = (max << hash) obj
```

Case 3: Adding an extra, unused parameter. Because this expression is not simple enough for generalization, the compiler issues the value restriction error.

```
let emptyList10 = Array.create 10 []
// Adding an extra (unused) parameter makes it a function, which is generalizable.
let emptyList10 () = Array.create 10 []
```

Case 4: Adding type parameters.

```
let arrayOf10Lists = Array.create 10 []
// Adding a type parameter and type annotation lets you write a generic value.
let arrayOf10Lists<'T> = Array.create 10 ([]:'T list)
```

In the last case, the value becomes a type function, which may be used to create values of many different types, for example as follows:

```
let intLists = arrayOf10Lists<int>
let floatLists = arrayOf10Lists<float>
```

See also

- Type Inference
- Generics
- Statically Resolved Type Parameters
- Constraints

Constraints

5/30/2019 • 4 minutes to read • Edit Online

This topic describes constraints that you can apply to generic type parameters to specify the requirements for a type argument in a generic type or function.

Syntax

```
type-parameter-list when constraint1 [ and constraint2]
```

Remarks

There are several different constraints you can apply to limit the types that can be used in a generic type. The following table lists and describes these constraints.

CONSTRAINT	SYNTAX	DESCRIPTION
Type Constraint	*type-parameter* :> *type*	The provided type must be equal to or derived from the type specified, or, if the type is an interface, the provided type must implement the interface.
Null Constraint	*type-parameter* : null	The provided type must support the null literal. This includes all .NET object types but not F# list, tuple, function, class, record, or union types.
Explicit Member Constraint	[(]*type-parameter* [or ... or *type-parameter*)] : (*member-signature*)	At least one of the type arguments provided must have a member that has the specified signature; not intended for common use. Members must be either explicitly defined on the type or part of an implicit type extension to be valid targets for an Explicit Member Constraint.
Constructor Constraint	*type-parameter* : (new : unit -> 'a)	The provided type must have a default constructor.
Value Type Constraint	: struct	The provided type must be a .NET value type.
Reference Type Constraint	: not struct	The provided type must be a .NET reference type.
Enumeration Type Constraint	: enum<*underlying-type*>	The provided type must be an enumerated type that has the specified underlying type; not intended for common use.

CONSTRAINT	SYNTAX	DESCRIPTION
Delegate Constraint	: delegate<*tuple-parameter-type*, *return-type*>	The provided type must be a delegate type that has the specified arguments and return value; not intended for common use.
Comparison Constraint	: comparison	The provided type must support comparison.
Equality Constraint	: equality	The provided type must support equality.
Unmanaged Constraint	: unmanaged	The provided type must be an unmanaged type. Unmanaged types are either certain primitive types (`sbyte` , `byte` , `char` , `nativeint` , `unativeint` , `float32` , `float` , `int16` , `uint16` , `int32` , `uint32` , `int64` , `uint64` , or `decimal`), enumeration types, `nativeptr<_>` , or a non-generic structure whose fields are all unmanaged types.

You have to add a constraint when your code has to use a feature that is available on the constraint type but not on types in general. For example, if you use the type constraint to specify a class type, you can use any one of the methods of that class in the generic function or type.

Specifying constraints is sometimes required when writing type parameters explicitly, because without a constraint, the compiler has no way of verifying that the features that you are using will be available on any type that might be supplied at run time for the type parameter.

The most common constraints you use in F# code are type constraints that specify base classes or interfaces. The other constraints are either used by the F# library to implement certain functionality, such as the explicit member constraint, which is used to implement operator overloading for arithmetic operators, or are provided mainly because F# supports the complete set of constraints that is supported by the common language runtime.

During the type inference process, some constraints are inferred automatically by the compiler. For example, if you use the `+` operator in a function, the compiler infers an explicit member constraint on variable types that are used in the expression.

The following code illustrates some constraint declarations.

```
// Base Type Constraint
type Class1<'T when 'T :> System.Exception> =
class end

// Interface Type Constraint
type Class2<'T when 'T :> System.IComparable> =
class end

// Null constraint
type Class3<'T when 'T : null> =
class end

// Member constraint with instance member
type Class5<'T when 'T : (member Method1 : 'T -> int)> =
class end

// Member constraint with property
type Class6<'T when 'T : (member Property1 : int)> =
class end

// Constructor constraint
type Class7<'T when 'T : (new : unit -> 'T)>() =
member val Field = new 'T()

// Reference type constraint
type Class8<'T when 'T : not struct> =
class end

// Enumeration constraint with underlying value specified
type Class9<'T when 'T : enum<uint32>> =
class end

// 'T must implement IComparable, or be an array type with comparable
// elements, or be System.IntPtr or System.UIntPtr. Also, 'T must not have
// the NoComparison attribute.
type Class10<'T when 'T : comparison> =
class end

// 'T must support equality. This is true for any type that does not
// have the NoEquality attribute.
type Class11<'T when 'T : equality> =
class end

type Class12<'T when 'T : delegate<obj * System.EventArgs, unit>> =
class end

type Class13<'T when 'T : unmanaged> =
class end

// Member constraints with two type parameters
// Most often used with static type parameters in inline functions
let inline add(value1 : ^T when ^T : (static member (+) : ^T * ^T -> ^T), value2: ^T) =
value1 + value2

// ^T and ^U must support operator +
let inline heterogenousAdd(value1 : ^T when (^T or ^U) : (static member (+) : ^T * ^U -> ^T), value2 : ^U)
=
value1 + value2

// If there are multiple constraints, use the and keyword to separate them.
type Class14<'T,'U when 'T : equality and 'U : equality> =
class end
```

See also

- Generics
- Constraints

Statically Resolved Type Parameters

7/30/2019 • 3 minutes to read • Edit Online

A *statically resolved type parameter* is a type parameter that is replaced with an actual type at compile time instead of at run time. They are preceded by a caret (^) symbol.

Syntax

```
^type-parameter
```

Remarks

In the F# language, there are two distinct kinds of type parameters. The first kind is the standard generic type parameter. These are indicated by an apostrophe ('), as in `'T` and `'U`. They are equivalent to generic type parameters in other .NET Framework languages. The other kind is statically resolved and is indicated by a caret symbol, as in `^T` and `^U`.

Statically resolved type parameters are primarily useful in conjunction with member constraints, which are constraints that allow you to specify that a type argument must have a particular member or members in order to be used. There is no way to create this kind of constraint by using a regular generic type parameter.

The following table summarizes the similarities and differences between the two kinds of type parameters.

FEATURE	GENERIC	STATICALLY RESOLVED
Syntax	`'T`, `'U`	`^T`, `^U`
Resolution time	Run time	Compile time
Member constraints	Cannot be used with member constraints.	Can be used with member constraints.
Code generation	A type (or method) with standard generic type parameters results in the generation of a single generic type or method.	Multiple instantiations of types and methods are generated, one for each type that is needed.
Use with types	Can be used on types.	Cannot be used on types.
Use with inline functions	No. An inline function cannot be parameterized with a standard generic type parameter.	Yes. Statically resolved type parameters cannot be used on functions or methods that are not inline.

Many F# core library functions, especially operators, have statically resolved type parameters. These functions and operators are inline, and result in efficient code generation for numeric computations.

Inline methods and functions that use operators, or use other functions that have statically resolved type parameters, can also use statically resolved type parameters themselves. Often, type inference infers such inline functions to have statically resolved type parameters. The following example illustrates an operator definition that is inferred to have a statically resolved type parameter.

```
let inline (+@) x y = x + x * y
// Call that uses int.
printfn "%d" (1 +@ 1)
// Call that uses float.
printfn "%f" (1.0 +@ 0.5)
```

The resolved type of `(+@)` is based on the use of both `(+)` and `(*)`, both of which cause type inference to infer member constraints on the statically resolved type parameters. The resolved type, as shown in the F# interpreter, is as follows.

```
^a -> ^c -> ^d
when (^a or ^b) : (static member ( + ) : ^a * ^b -> ^d) and
(^a or ^c) : (static member ( * ) : ^a * ^c -> ^b)
```

The output is as follows.

```
2
1.500000
```

Starting with F# 4.1, you can also specify concrete type names in statically resolved type parameter signatures. In previous versions of the language, the type name could actually be inferred by the compiler, but could not actually be specified in the signature. As of F# 4.1, you may also specify concrete type names in statically resolved type parameter signatures. Here's an example:

```
let inline konst x _ = x

type CFunctor() =
    static member inline fmap (f: ^a -> ^b, a: ^a list) = List.map f a
    static member inline fmap (f: ^a -> ^b, a: ^a option) =
        match a with
        | None -> None
        | Some x -> Some (f x)

    // default implementation of replace
    static member inline replace< ^a, ^b, ^c, ^d, ^e when ^a :> CFunctor and (^a or ^d): (static member fmap:
(^b -> ^c) * ^d -> ^e) > (a, f) =
        ((^a or ^d) : (static member fmap : (^b -> ^c) * ^d -> ^e) (konst a, f))

    // call overridden replace if present
    static member inline replace< ^a, ^b, ^c when ^b: (static member replace: ^a * ^b -> ^c)>(a: ^a, f: ^b) =
        (^b : (static member replace: ^a * ^b -> ^c) (a, f))

let inline replace_instance< ^a, ^b, ^c, ^d when (^a or ^c): (static member replace: ^b * ^c -> ^d)> (a: ^b,
f: ^c) =
        ((^a or ^c): (static member replace: ^b * ^c -> ^d) (a, f))

// Note the concrete type 'CFunctor' specified in the signature
let inline replace (a: ^a) (f: ^b): ^a0 when (CFunctor or  ^b): (static member replace: ^a *  ^b ->  ^a0) =
    replace_instance<CFunctor, _, _, _> (a, f)
```

See also

- Generics
- Type Inference
- Automatic Generalization
- Constraints

- Inline Functions

Records

7/30/2019 • 6 minutes to read • Edit Online

Records represent simple aggregates of named values, optionally with members. They can either be structs or reference types. They are reference types by default.

Syntax

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
    [ member-list ]
```

Remarks

In the previous syntax, *typename* is the name of the record type, *label1* and *label2* are names of values, referred to as *labels*, and *type1* and *type2* are the types of these values. *member-list* is the optional list of members for the type. You can use the `[<Struct>]` attribute to create a struct record rather than a record which is a reference type.

Following are some examples.

```
// Labels are separated by semicolons when defined on the same line.
type Point = { X: float; Y: float; Z: float; }

// You can define labels on their own line with a semicolon.
type Customer =
    { First: string
      Last: string
      SSN: uint32
      AccountNumber: uint32; }

// A struct record.
[<Struct>]
type StructPoint =
    { X: float
      Y: float
      Z: float }
```

When each label is on a separate line, the semicolon is optional.

You can set values in expressions known as *record expressions*. The compiler infers the type from the labels used (if the labels are sufficiently distinct from those of other record types). Braces ({ }) enclose the record expression. The following code shows a record expression that initializes a record with three float elements with labels `x`, `y` and `z`.

```
let mypoint = { X = 1.0; Y = 1.0; Z = -1.0; }
```

Do not use the shortened form if there could be another type that also has the same labels.

```
type Point = { X: float; Y: float; Z: float; }
type Point3D = { X: float; Y: float; Z: float }
// Ambiguity: Point or Point3D?
let mypoint3D = { X = 1.0; Y = 1.0; Z = 0.0; }
```

The labels of the most recently declared type take precedence over those of the previously declared type, so in the preceding example, `mypoint3D` is inferred to be `Point3D`. You can explicitly specify the record type, as in the following code.

```
let myPoint1 = { Point.X = 1.0; Y = 1.0; Z = 0.0; }
```

Methods can be defined for record types just as for class types.

Creating Records by Using Record Expressions

You can initialize records by using the labels that are defined in the record. An expression that does this is referred to as a *record expression*. Use braces to enclose the record expression and use the semicolon as a delimiter.

The following example shows how to create a record.

```
type MyRecord =
    { X: int
      Y: int
      Z: int }

let myRecord1 = { X = 1; Y = 2; Z = 3; }
```

The semicolons after the last field in the record expression and in the type definition are optional, regardless of whether the fields are all in one line.

When you create a record, you must supply values for each field. You cannot refer to the values of other fields in the initialization expression for any field.

In the following code, the type of `myRecord2` is inferred from the names of the fields. Optionally, you can specify the type name explicitly.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

Another form of record construction can be useful when you have to copy an existing record, and possibly change some of the field values. The following line of code illustrates this.

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

This form of the record expression is called the *copy and update record expression*.

Records are immutable by default; however, you can easily create modified records by using a copy and update expression. You can also explicitly specify a mutable field.

```
type Car =
    { Make : string
      Model : string
      mutable Odometer : int }

let myCar = { Make = "Fabrikam"; Model = "Coupe"; Odometer = 108112 }
myCar.Odometer <- myCar.Odometer + 21
```

Don't use the DefaultValue attribute with record fields. A better approach is to define default instances of records with fields that are initialized to default values and then use a copy and update record expression to set any fields that differ from the default values.

```
// Rather than use [<DefaultValue>], define a default record.
type MyRecord =
    { Field1 : int
      Field2 : int }

let defaultRecord1 = { Field1 = 0; Field2 = 0 }
let defaultRecord2 = { Field1 = 1; Field2 = 25 }

// Use the with keyword to populate only a few chosen fields
// and leave the rest with default values.
let rr3 = { defaultRecord1 with Field2 = 42 }
```

Creating Mutually Recursive Records

Sometime when creating a record, you may want to have it depend on another type that you would like to define afterwards. This is a compile error unless you define the record types to be mutually recursive.

Defining mutually recursive records is done with the `and` keyword. This lets you link 2 or more record types together.

For example, the following code defines a `Person` and `Address` type as mutually recursive:

```
// Create a Person type and use the Address type that is not defined
type Person =
  { Name: string
    Age: int
    Address: Address }
// Define the Address type which is used in the Person record
and Address =
  { Line1: string
    Line2: string
    PostCode: string }
```

If you were to define the previous example without the `and` keyword, then it would not compile. The `and` keyword is required for mutually recursive definitions.

Pattern Matching with Records

Records can be used with pattern matching. You can specify some fields explicitly and provide variables for other fields that will be assigned when a match occurs. The following code example illustrates this.

```
type Point3D = { X: float; Y: float; Z: float }
let evaluatePoint (point: Point3D) =
    match point with
    | { X = 0.0; Y = 0.0; Z = 0.0 } -> printfn "Point is at the origin."
    | { X = xVal; Y = 0.0; Z = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
    | { X = 0.0; Y = yVal; Z = 0.0 } -> printfn "Point is on the y-axis. Value is %f." yVal
    | { X = 0.0; Y = 0.0; Z = zVal } -> printfn "Point is on the z-axis. Value is %f." zVal
    | { X = xVal; Y = yVal; Z = zVal } -> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }
```

The output of this code is as follows.

```
Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).
```

Differences Between Records and Classes

Record fields differ from classes in that they are automatically exposed as properties, and they are used in the creation and copying of records. Record construction also differs from class construction. In a record type, you cannot define a constructor. Instead, the construction syntax described in this topic applies. Classes have no direct relationship between constructor parameters, fields, and properties.

Like union and structure types, records have structural equality semantics. Classes have reference equality semantics. The following code example demonstrates this.

```
type RecordTest = { X: int; Y: int }

let record1 = { X = 1; Y = 2 }
let record2 = { X = 1; Y = 2 }

if (record1 = record2) then
    printfn "The records are equal."
else
    printfn "The records are unequal."
```

The output of this code is as follows:

```
The records are equal.
```

If you write the same code with classes, the two class objects would be unequal because the two values would represent two objects on the heap and only the addresses would be compared (unless the class type overrides the `System.Object.Equals` method).

If you need reference equality for records, add the attribute `[<ReferenceEquality>]` above the record.

See also

- F# Types
- Classes
- F# Language Reference
- Reference-Equality

- Pattern Matching

Anonymous Records

6/13/2019 • 8 minutes to read • Edit Online

Anonymous records are simple aggregates of named values that don't need to be declared before use. You can declare them as either structs or reference types. They're reference types by default.

Syntax

The following examples demonstrate the anonymous record syntax. Items delimited as `[item]` are optional.

```
// Construct an anonymous record
let value-name = [struct] {| Label1: Type1; Label2: Type2; ...|}

// Use an anonymous record as a type parameter
let value-name = Type-Name<[struct] {| Label1: Type1; Label2: Type2; ...|}>

// Define a parameter with an anonymous record as input
let function-name (arg-name: [struct] {| Label1: Type1; Label2: Type2; ...|}) ...
```

Basic usage

Anonymous records are best thought of as F# record types that don't need to be declared before instantiation.

For example, here how you can interact with a function that produces an anonymous record:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

The following example expands on the previous one with a `printCircleStats` function that takes an anonymous record as input:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let printCircleStats r (stats: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Calling `printCircleStats` with any anonymous record type that doesn't have the same "shape" as the input type will fail to compile:

```
printCircleStats r {| Diameter = 2.0; Area = 4.0; MyCircumference = 12.566371 |}
// Two anonymous record types have mismatched sets of field names
// '["Area"; "Circumference"; "Diameter"]' and '["Area"; "Diameter"; "MyCircumference"]'
```

Struct anonymous records

Anonymous records can also be defined as struct with the optional `struct` keyword. The following example augments the previous one by producing and consuming a struct anonymous record:

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    // Note that the keyword comes before the '{| |}' brace pair
    struct {| Area = a; Circumference = c; Diameter = d |}

// the 'struct' keyword also comes before the '{| |}' brace pair when declaring the parameter type
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats
```

Structness inference

Struct anonymous records also allow for "structness inference" where you do not need to specify the `struct` keyword at the call site. In this example, you elide the `struct` keyword when calling `printCircleStats`:

```

let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

printCircleStats r {| Area = 4.0; Circumference = 12.6; Diameter = 12.6 |}
```

The reverse pattern - specifying `struct` when the input type is not a struct anonymous record - will fail to compile.

Embedding anonymous records within other types

It's useful to declare discriminated unions whose cases are records. But if the data in the records is the same type as the discriminated union, you must define all types as mutually recursive. Using anonymous records avoids this restriction. What follows is an example type and function that pattern matches over it:

```
type FullName = { FirstName: string; LastName: string }

// Note that using a named for Manager and Executive would require mutually recursive definitions.
type Employee =
    | Engineer of FullName
    | Manager of {| Name: FullName; Reports: Employee list |}
    | Executive of {| Name: FullName; Reports: Employee list; Assistant: Employee |}

let getFirstName e =
    match e with
    | Engineer fullName -> fullName.FirstName
    | Manager m -> m.Name.FirstName
    | Executive ex -> ex.Name.FirstName
```

Copy and update expressions

Anonymous records support construction with copy and update expressions. For example, here's how you can construct a new instance of an anonymous record that copies an existing one's data:

```
let data = {| X = 1; Y = 2 |}
let data' = {| data with Y = 3 |}
```

However, unlike named records, anonymous records allow you to construct entirely different forms with copy and update expressions. The follow example takes the same anonymous record from the previous example and expands it into a new anonymous record:

```
let data = {| X = 1; Y = 2 |}
let expandedData = {| data with Z = 3 |} // Gives {| X=1; Y=2; Z=3 |}
```

It is also possible to construct anonymous records from instances of named records:

```
type R = { X: int }
let data = { X = 1 }
let data' = {| data with Y = 2 |} // Gives {| X=1; Y=2 |}
```

You can also copy data to and from reference and struct anonymous records:

```
// Copy data from a reference record into a struct anonymous record
type R1 = { X: int }
let r1 = { X = 1 }

let data1 = struct {| r1 with Y = 1 |}

// Copy data from a struct record into a reference anonymous record
[<Struct>]
type R2 = { X: int }
let r2 = { X = 1 }

let data2 = {| r1 with Y = 1 |}

// Copy the reference anonymous record data into a struct anonymous record
let data3 = struct {| data2 with Z = r2.X |}
```

Properties of anonymous records

Anonymous records have a number of characteristics that are essential to fully understanding how they can be used.

Anonymous records are nominal

Anonymous records are nominal types. They are best thought as named record types (which are also nominal) that do not require an up-front declaration.

Consider the following example with two anonymous record declarations:

```
let x = {| X = 1 |}
let y = {| Y = 1 |}
```

The `x` and `y` values have different types and are not compatible with one another. They are not equatable and they are not comparable. To illustrate this, consider a named record equivalent:

```
type X = { X: int }
type Y = { Y: int }

let x = { X = 1 }
let y = { Y = 1 }
```

There isn't anything inherently different about anonymous records when compared with their named record equivalents when concerning type equivalency or comparison.

Anonymous records use structural equality and comparison

Like record types, anonymous records are structurally equatable and comparable. This is only true if all constituent types support equality and comparison, like with record types. To support equality or comparison, two anonymous records must have the same "shape".

```
{| a = 1+1 |} = {| a = 2 |} // true
{| a = 1+1 |} > {| a = 1 |} // true

// error FS0001: Two anonymous record types have mismatched sets of field names '["a"]' and '["a"; "b"]'
{| a = 1 + 1 |} = {| a = 2;  b = 1|}
```

Anonymous records are serializable

You can serialize anonymous records just as you can with named records. Here is an example using Newtonsoft.Json:

```
open Newtonsoft.Json

let phillip = {| name="Phillip"; age=28 |}
JsonConvert.SerializeObject(phillip)

let phillip = JsonConvert.DeserializeObject<{|name: string; age: int|}>(str)
printfn "Name: %s Age: %d" phillip.name phillip.age
```

Anonymous records are useful for sending lightweight data over a network without the need to define a domain for your serialized/deserialized types up front.

Anonymous records interoperate with C# anonymous types

It is possible to use a .NET API that requires the use of C# anonymous types. C# anonymous types are trivial to interoperate with by using anonymous records. The following example shows how to use anonymous records to call a LINQ overload that requires an anonymous type:

```
open System.Linq

let names = [ "Ana"; "Felipe"; "Emilia"]
let nameGrouping = names.Select(fun n -> {| Name = n; FirstLetter = n.[0] |})
for ng in nameGrouping do
    printfn "%s has first letter %c" ng.Name ng.FirstLetter
```

There are a multitude of other APIs used throughout .NET that require the use of passing in an anonymous type. Anonymous records are your tool for working with them.

Limitations

Anonymous records have some restrictions in their usage. Some are inherent to their design, but others are amenable to change.

Limitations with pattern matching

Anonymous records do not support pattern matching, unlike named records. There are three reasons:

1. A pattern would have to account for every field of an anonymous record, unlike named record types. This is because anonymous records do not support structural subtyping – they are nominal types.
2. Because of (1), there is no ability to have additional patterns in a pattern match expression, as each distinct pattern would imply a different anonymous record type.
3. Because of (3), any anonymous record pattern would be more verbose than the use of "dot" notation.

There is an open language suggestion to allow pattern matching in limited contexts.

Limitations with mutability

It is not currently possible to define an anonymous record with `mutable` data. There is an open language suggestion to allow mutable data.

Limitations with struct anonymous records

It is not possible to declare struct anonymous records as `IsByRefLike` or `IsReadOnly`. There is an open language suggestion to for `IsByRefLike` and `IsReadOnly` anonymous records.

Copy and Update Record Expressions

7/30/2019 • 2 minutes to read • Edit Online

A *copy and update record expression* is an expression that copies an existing record, updates specified fields, and returns the updated record.

Syntax

```
{ record-name with
    updated-labels }

{| anonymous-record-name with
    updated-labels |}
```

Remarks

Records and anonymous records are immutable by default, so that there is no update to an existing record possible. To create an updated record all the fields of a record would have to be specified again. To simplify this task a *copy and update expression* can be used. This expression takes an existing record, creates a new one of the same type by using specified fields from the expression and the missing field specified by the expression.

This can be useful when you have to copy an existing record, and possibly change some of the field values.

Take for instance a newly created record.

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

If you were to update only on field of that record you could use the *copy and update record expression* like the following:

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

See also

- Records
- Anonymous Records
- F# Language Reference

Discriminated Unions

7/30/2019 • 9 minutes to read • Edit Online

Discriminated unions provide support for values that can be one of a number of named cases, possibly each with different values and types. Discriminated unions are useful for heterogeneous data; data that can have special cases, including valid and error cases; data that varies in type from one instance to another; and as an alternative for small object hierarchies. In addition, recursive discriminated unions are used to represent tree data structures.

Syntax

```
[ attributes ]
type [accessibility-modifier] type-name =
    | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]
    | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]

    [ member-list ]
```

Remarks

Discriminated unions are similar to union types in other languages, but there are differences. As with a union type in C++ or a variant type in Visual Basic, the data stored in the value is not fixed; it can be one of several distinct options. Unlike unions in these other languages, however, each of the possible options is given a *case identifier*. The case identifiers are names for the various possible types of values that objects of this type could be; the values are optional. If values are not present, the case is equivalent to an enumeration case. If values are present, each value can either be a single value of a specified type, or a tuple that aggregates multiple fields of the same or different types. You can give an individual field a name, but the name is optional, even if other fields in the same case are named.

Accessibility for discriminated unions defaults to `public`.

For example, consider the following declaration of a Shape type.

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Prism of width : float * float * height : float
```

The preceding code declares a discriminated union Shape, which can have values of any of three cases: Rectangle, Circle, and Prism. Each case has a different set of fields. The Rectangle case has two named fields, both of type `float`, that have the names width and length. The Circle case has just one named field, radius. The Prism case has three fields, two of which (width and height) are named fields. Unnamed fields are referred to as anonymous fields.

You construct objects by providing values for the named and anonymous fields according to the following examples.

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

This code shows that you can either use the named fields in the initialization, or you can rely on the ordering of the fields in the declaration and just provide the values for each field in turn. The constructor call for `rect` in the previous code uses the named fields, but the constructor call for `circ` uses the ordering. You can mix the ordered fields and named fields, as in the construction of `prism`.

The `option` type is a simple discriminated union in the F# core library. The `option` type is declared as follows.

```
// The option type is a discriminated union.
type Option<'a> =
    | Some of 'a
    | None
```

The previous code specifies that the type `Option` is a discriminated union that has two cases, `Some` and `None`. The `Some` case has an associated value that consists of one anonymous field whose type is represented by the type parameter `'a`. The `None` case has no associated value. Thus the `option` type specifies a generic type that either has a value of some type or no value. The type `Option` also has a lowercase type alias, `option`, that is more commonly used.

The case identifiers can be used as constructors for the discriminated union type. For example, the following code is used to create values of the `option` type.

```
let myOption1 = Some(10.0)
let myOption2 = Some("string")
let myOption3 = None
```

The case identifiers are also used in pattern matching expressions. In a pattern matching expression, identifiers are provided for the values associated with the individual cases. For example, in the following code, `x` is the identifier given the value that is associated with the `Some` case of the `option` type.

```
let printValue opt =
    match opt with
    | Some x -> printfn "%A" x
    | None -> printfn "No value."
```

In pattern matching expressions, you can use named fields to specify discriminated union matches. For the Shape type that was declared previously, you can use the named fields as the following code shows to extract the values of the fields.

```
let getShapeHeight shape =
    match shape with
    | Rectangle(height = h) -> h
    | Circle(radius = r) -> 2. * r
    | Prism(height = h) -> h
```

Normally, the case identifiers can be used without qualifying them with the name of the union. If you want the name to always be qualified with the name of the union, you can apply the RequireQualifiedAccess attribute to the union type definition.

Unwrapping Discriminated Unions

In F# Discriminated Unions are often used in domain-modeling for wrapping a single type. It's easy to extract the underlying value via pattern matching as well. You don't need to use a match expression for a single case:

```
let ([UnionCaseName] [values]) = [UnionValue]
```

The following example demonstrates this:

```
type ShaderProgram = | ShaderProgram of id:int

let someFunctionUsingShaderProgram shaderProgram =
    let (ShaderProgram id) = shaderProgram
    // Use the unwrapped value
    ...
```

Pattern matching is also allowed directly in function parameters, so you can unwrap a single case there:

```
let someFunctionUsingShaderProgram (ShaderProgram id) =
    // Use the unwrapped value
    ...
```

Struct Discriminated Unions

You can also represent Discriminated Unions as structs. This is done with the `[<Struct>]` attribute.

```
[<Struct>]
type SingleCase = Case of string

[<Struct>]
type Multicase =
    | Case1 of Case1 : string
    | Case2 of Case2 : int
    | Case3 of Case3 : double
```

Because these are value types and not reference types, there are extra considerations compared with reference discriminated unions:

1. They are copied as value types and have value type semantics.
2. You cannot use a recursive type definition with a multicase struct Discriminated Union.
3. You must provide unique case names for a multicase struct Discriminated Union.

Using Discriminated Unions Instead of Object Hierarchies

You can often use a discriminated union as a simpler alternative to a small object hierarchy. For example, the following discriminated union could be used instead of a `Shape` base class that has derived types for circle, square, and so on.

```
type Shape =
  // The value here is the radius.
| Circle of float
  // The value here is the side length.
| EquilateralTriangle of double
  // The value here is the side length.
| Square of double
  // The values here are the height and width.
| Rectangle of double * double
```

Instead of a virtual method to compute an area or perimeter, as you would use in an object-oriented implementation, you can use pattern matching to branch to appropriate formulas to compute these quantities. In the following example, different formulas are used to compute the area, depending on the shape.

```
let pi = 3.141592654

let area myShape =
    match myShape with
    | Circle radius -> pi * radius * radius
    | EquilateralTriangle s -> (sqrt 3.0) / 4.0 * s * s
    | Square s -> s * s
    | Rectangle (h, w) -> h * w

let radius = 15.0
let myCircle = Circle(radius)
printfn "Area of circle that has radius %f: %f" radius (area myCircle)

let squareSide = 10.0
let mySquare = Square(squareSide)
printfn "Area of square that has side %f: %f" squareSide (area mySquare)

let height, width = 5.0, 10.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle that has height %f and width %f is %f" height width (area myRectangle)
```

The output is as follows:

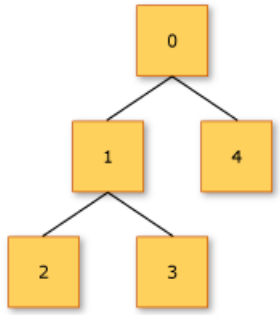
```
Area of circle that has radius 15.000000: 706.858347
Area of square that has side 10.000000: 100.000000
Area of rectangle that has height 5.000000 and width 10.000000 is 50.000000
```

Using Discriminated Unions for Tree Data Structures

Discriminated unions can be recursive, meaning that the union itself can be included in the type of one or more cases. Recursive discriminated unions can be used to create tree structures, which are used to model expressions in programming languages. In the following code, a recursive discriminated union is used to create a binary tree data structure. The union consists of two cases, `Node`, which is a node with an integer value and left and right subtrees, and `Tip`, which terminates the tree.

```
type Tree =
    | Tip
    | Node of int * Tree * Tree

let rec sumTree tree =
    match tree with
    | Tip -> 0
    | Node(value, left, right) ->
        value + sumTree(left) + sumTree(right)
let myTree = Node(0, Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip)), Node(4, Tip, Tip))
let resultSumTree = sumTree myTree
```

In the previous code, `resultSumTree` has the value 10. The following illustration shows the tree structure for `myTree`.



Discriminated unions work well if the nodes in the tree are heterogeneous. In the following code, the type `Expression` represents the abstract syntax tree of an expression in a simple programming language that supports addition and multiplication of numbers and variables. Some of the union cases are not recursive and represent either numbers (`Number`) or variables (`Variable`). Other cases are recursive, and represent operations (`Add` and `Multiply`), where the operands are also expressions. The `Evaluate` function uses a match expression to recursively process the syntax tree.

```
type Expression =
    | Number of int
    | Add of Expression * Expression
    | Multiply of Expression * Expression
    | Variable of string

let rec Evaluate (env:Map<string,int>) exp =
    match exp with
    | Number n -> n
    | Add (x, y) -> Evaluate env x + Evaluate env y
    | Multiply (x, y) -> Evaluate env x * Evaluate env y
    | Variable id    -> env.[id]

let environment = Map.ofList [ "a", 1 ;
                               "b", 2 ;
                               "c", 3 ]

// Create an expression tree that represents
// the expression: a + 2 * b.
let expressionTree1 = Add(Variable "a", Multiply(Number 2, Variable "b"))

// Evaluate the expression a + 2 * b, given the
// table of values for the variables.
let result = Evaluate environment expressionTree1
```

When this code is executed, the value of `result` is 5.

Members

It is possible to define members on discriminated unions. The following example shows how to define a property and implement an interface:

```
open System

type IPrintable =
    abstract Print: unit -> unit

type Shape =
    | Circle of float
    | EquilateralTriangle of float
    | Square of float
    | Rectangle of float * float

    member this.Area =
        match this with
        | Circle r -> 2.0 * Math.PI * r
        | EquilateralTriangle s -> s * s * sqrt 3.0 / 4.0
        | Square s -> s * s
        | Rectangle(l, w) -> l * w

    interface IPrintable with
        member this.Print () =
            match this with
            | Circle r -> printfn "Circle with radius %f" r
            | EquilateralTriangle s -> printfn "Equilateral Triangle of side %f" s
            | Square s -> printfn "Square with side %f" s
            | Rectangle(l, w) -> printfn "Rectangle with length %f and width %f" l w
```

Common attributes

The following attributes are commonly seen in discriminated unions:

- `[<RequireQualifiedAccess>]`
- `[<NoEquality>]`
- `[<NoComparison>]`
- `[<Struct>]`

See also

- F# Language Reference

Enumerations

7/30/2019 • 2 minutes to read • Edit Online

Enumerations, also known as *enums*, , are integral types where labels are assigned to a subset of the values. You can use them in place of literals to make code more readable and maintainable.

Syntax

```
type enum-name =
| value1 = integer-literal1
| value2 = integer-literal2
...
```

Remarks

An enumeration looks much like a discriminated union that has simple values, except that the values can be specified. The values are typically integers that start at 0 or 1, or integers that represent bit positions. If an enumeration is intended to represent bit positions, you should also use the Flags attribute.

The underlying type of the enumeration is determined from the literal that is used, so that, for example, you can use literals with a suffix, such as `1u` , `2u` , and so on, for an unsigned integer (`uint32`) type.

When you refer to the named values, you must use the name of the enumeration type itself as a qualifier, that is, `enum-name.value1` , not just `value1` . This behavior differs from that of discriminated unions. This is because enumerations always have the RequireQualifiedAccess attribute.

The following code shows the declaration and use of an enumeration.

```
// Declaration of an enumeration.
type Color =
   | Red = 0
   | Green = 1
   | Blue = 2
// Use of an enumeration.
let col1 : Color = Color.Red
```

You can easily convert enumerations to the underlying type by using the appropriate operator, as shown in the following code.

```
// Conversion to an integral type.
let n = int col1
```

Enumerated types can have one of the following underlying types: `sbyte` , `byte` , `int16` , `uint16` , `int32` , `uint32` , `int64` , `uint16` , `uint64` , and `char` . Enumeration types are represented in the .NET Framework as types that are inherited from `System.Enum` , which in turn is inherited from `System.ValueType` . Thus, they are value types that are located on the stack or inline in the containing object, and any value of the underlying type is a valid value of the enumeration. This is significant when pattern matching on enumeration values, because you have to provide a pattern that catches the unnamed values.

The `enum` function in the F# library can be used to generate an enumeration value, even a value other than one of

the predefined, named values. You use the `enum` function as follows.

```
let col2 = enum<Color>(3)
```

The default `enum` function works with type `int32`. Therefore, it cannot be used with enumeration types that have other underlying types. Instead, use the following.

```
type uColor =
   | Red = 0u
   | Green = 1u
   | Blue = 2u
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

Additionally, cases for enums are always emitted as `public`. This is so that they align with C# and the rest of the .NET platform.

See also

- F# Language Reference
- Casting and Conversions

Type Abbreviations

7/30/2019 • 2 minutes to read • Edit Online

A *type abbreviation* is an alias or alternate name for a type.

Syntax

```
type [accessibility-modifier] type-abbreviation = type-name
```

Remarks

You can use type abbreviations to give a type a more meaningful name, in order to make code easier to read. You can also use them to create an easy to use name for a type that is otherwise cumbersome to write out. Additionally, you can use type abbreviations to make it easier to change an underlying type without changing all the code that uses the type. The following is a simple type abbreviation.

Accessibility of type abbreviations defaults to `public`.

```
type SizeType = uint32
```

Type abbreviations can include generic parameters, as in the following code.

```
type Transform<'a> = 'a -> 'a
```

In the previous code, `Transform` is a type abbreviation that represents a function that takes a single argument of any type and that returns a single value of that same type.

Type abbreviations are not preserved in the .NET Framework MSIL code. Therefore, when you use an F# assembly from another .NET Framework language, you must use the underlying type name for a type abbreviation.

Type abbreviations can also be used on units of measure. For more information, see Units of Measure.

See also

- F# Language Reference

Classes

7/30/2019 • 8 minutes to read • Edit Online

Classes are types that represent objects that can have properties, methods, and events.

Syntax

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Remarks

Classes represent the fundamental description of .NET object types; the class is the primary type concept that supports object-oriented programming in F#.

In the preceding syntax, the `type-name` is any valid identifier. The `type-params` describes optional generic type parameters. It consists of type parameter names and constraints enclosed in angle brackets (`<` and `>`). For more information, see Generics and Constraints. The `parameter-list` describes constructor parameters. The first access modifier pertains to the type; the second pertains to the primary constructor. In both cases, the default is `public`.

You specify the base class for a class by using the `inherit` keyword. You must supply arguments, in parentheses, for the base class constructor.

You declare fields or function values that are local to the class by using `let` bindings, and you must follow the general rules for `let` bindings. The `do-bindings` section includes code to be executed upon object construction.

The `member-list` consists of additional constructors, instance and static method declarations, interface declarations, abstract bindings, and property and event declarations. These are described in Members.

The `identifier` that is used with the optional `as` keyword gives a name to the instance variable, or self identifier, which can be used in the type definition to refer to the instance of the type. For more information, see the section Self Identifiers later in this topic.

The keywords `class` and `end` that mark the start and end of the definition are optional.

Mutually recursive types, which are types that reference each other, are joined together with the `and` keyword just as mutually recursive functions are. For an example, see the section Mutually Recursive Types.

Constructors

The constructor is code that creates an instance of the class type. Constructors for classes work somewhat

differently in F# than they do in other .NET languages. In an F# class, there is always a primary constructor whose arguments are described in the `parameter-list` that follows the type name, and whose body consists of the `let` (and `let rec`) bindings at the start of the class declaration and the `do` bindings that follow. The arguments of the primary constructor are in scope throughout the class declaration.

You can add additional constructors by using the `new` keyword to add a member, as follows:

`new` (`argument-list`) = `constructor-body`

The body of the new constructor must invoke the primary constructor that is specified at the top of the class declaration.

The following example illustrates this concept. In the following code, `MyClass` has two constructors, a primary constructor that takes two arguments and another constructor that takes no arguments.

```
type MyClass1(x: int, y: int) =
   do printfn "%d %d" x y
   new() = MyClass1(0, 0)
```

let and do Bindings

The `let` and `do` bindings in a class definition form the body of the primary class constructor, and therefore they run whenever a class instance is created. If a `let` binding is a function, then it is compiled into a member. If the `let` binding is a value that is not used in any function or member, then it is compiled into a variable that is local to the constructor. Otherwise, it is compiled into a field of the class. The `do` expressions that follow are compiled into the primary constructor and execute initialization code for every instance. Because any additional constructors always call the primary constructor, the `let` bindings and `do` bindings always execute regardless of which constructor is called.

Fields that are created by `let` bindings can be accessed throughout the methods and properties of the class; however, they cannot be accessed from static methods, even if the static methods take an instance variable as a parameter. They cannot be accessed by using the self identifier, if one exists.

Self Identifiers

A *self identifier* is a name that represents the current instance. Self identifiers resemble the `this` keyword in C# or C++ or `Me` in Visual Basic. You can define a self identifier in two different ways, depending on whether you want the self identifier to be in scope for the whole class definition or just for an individual method.

To define a self identifier for the whole class, use the `as` keyword after the closing parentheses of the constructor parameter list, and specify the identifier name.

To define a self identifier for just one method, provide the self identifier in the member declaration, just before the method name and a period (.) as a separator.

The following code example illustrates the two ways to create a self identifier. In the first line, the `as` keyword is used to define the self identifier. In the fifth line, the identifier `this` is used to define a self identifier whose scope is restricted to the method `PrintMessage`.

```
type MyClass2(dataIn) as self =
    let data = dataIn
    do
        self.PrintMessage()
    member this.PrintMessage() =
        printf "Creating MyClass2 with Data %d" data
```

Unlike in other .NET languages, you can name the self identifier however you want; you are not restricted to names such as `self`, `Me`, or `this`.

The self identifier that is declared with the `as` keyword is not initialized until after the `let` bindings are executed. Therefore, it cannot be used in the `let` bindings. You can use the self identifier in the `do` bindings section.

Generic Type Parameters

Generic type parameters are specified in angle brackets (`<` and `>`), in the form of a single quotation mark followed by an identifier. Multiple generic type parameters are separated by commas. The generic type parameter is in scope throughout the declaration. The following code example shows how to specify generic type parameters.

```
type MyGenericClass<'a> (x: 'a) =
   do printfn "%A" x
```

Type arguments are inferred when the type is used. In the following code, the inferred type is a sequence of tuples.

```
let g1 = MyGenericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

Specifying Inheritance

The `inherit` clause identifies the direct base class, if there is one. In F#, only one direct base class is allowed. Interfaces that a class implements are not considered base classes. Interfaces are discussed in the Interfaces topic.

You can access the methods and properties of the base class from the derived class by using the language keyword `base` as an identifier, followed by a period (.) and the name of the member.

For more information, see Inheritance.

Members Section

You can define static or instance methods, properties, interface implementations, abstract members, event declarations, and additional constructors in this section. Let and do bindings cannot appear in this section. Because members can be added to a variety of F# types in addition to classes, they are discussed in a separate topic, Members.

Mutually Recursive Types

When you define types that reference each other in a circular way, you string together the type definitions by using the `and` keyword. The `and` keyword replaces the `type` keyword on all except the first definition, as follows.

```
open System.IO

type Folder(pathIn: string) =
  let path = pathIn
  let filenameArray : string array = Directory.GetFiles(path)
  member this.FileArray = Array.map (fun elem -> new File(elem, this)) filenameArray

and File(filename: string, containingFolder: Folder) =
   member this.Name = filename
   member this.ContainingFolder = containingFolder

let folder1 = new Folder(".")
for file in folder1.FileArray do
   printfn "%s" file.Name
```

The output is a list of all the files in the current directory.

When to Use Classes, Unions, Records, and Structures

Given the variety of types to choose from, you need to have a good understanding of what each type is designed for to select the appropriate type for a particular situation. Classes are designed for use in object-oriented programming contexts. Object-oriented programming is the dominant paradigm used in applications that are written for the .NET Framework. If your F# code has to work closely with the .NET Framework or another object-oriented library, and especially if you have to extend from an object-oriented type system such as a UI library, classes are probably appropriate.

If you are not interoperating closely with object-oriented code, or if you are writing code that is self-contained and therefore protected from frequent interaction with object-oriented code, you should consider using records and discriminated unions. A single, well thought–out discriminated union, together with appropriate pattern matching code, can often be used as a simpler alternative to an object hierarchy. For more information about discriminated unions, see Discriminated Unions.

Records have the advantage of being simpler than classes, but records are not appropriate when the demands of a type exceed what can be accomplished with their simplicity. Records are basically simple aggregates of values, without separate constructors that can perform custom actions, without hidden fields, and without inheritance or interface implementations. Although members such as properties and methods can be added to records to make their behavior more complex, the fields stored in a record are still a simple aggregate of values. For more information about records, see Records.

Structures are also useful for small aggregates of data, but they differ from classes and records in that they are .NET value types. Classes and records are .NET reference types. The semantics of value types and reference types are different in that value types are passed by value. This means that they are copied bit for bit when they are passed as a parameter or returned from a function. They are also stored on the stack or, if they are used as a field, embedded inside the parent object instead of stored in their own separate location on the heap. Therefore, structures are appropriate for frequently accessed data when the overhead of accessing the heap is a problem. For more information about structures, see Structures.

See also

- F# Language Reference
- Members
- Inheritance
- Interfaces

Structures

7/30/2019 • 4 minutes to read • Edit Online

A *structure* is a compact object type that can be more efficient than a class for types that have a small amount of data and simple behavior.

Syntax

```
[ attributes ]
type [accessibility-modifier] type-name =
    struct
        type-definition-elements-and-members
    end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
    type-definition-elements-and-members
```

Remarks

Structures are *value types*, which means that they are stored directly on the stack or, when they are used as fields or array elements, inline in the parent type. Unlike classes and records, structures have pass-by-value semantics. This means that they are useful primarily for small aggregates of data that are accessed and copied frequently.

In the previous syntax, two forms are shown. The first is not the lightweight syntax, but it is nevertheless frequently used because, when you use the `struct` and `end` keywords, you can omit the `StructAttribute` attribute, which appears in the second form. You can abbreviate `StructAttribute` to just `Struct`.

The *type-definition-elements-and-members* in the previous syntax represents member declarations and definitions. Structures can have constructors and mutable and immutable fields, and they can declare members and interface implementations. For more information, see Members.

Structures cannot participate in inheritance, cannot contain `let` or `do` bindings, and cannot recursively contain fields of their own type (although they can contain reference cells that reference their own type).

Because structures do not allow `let` bindings, you must declare fields in structures by using the `val` keyword. The `val` keyword defines a field and its type but does not allow initialization. Instead, `val` declarations are initialized to zero or null. For this reason, structures that have an implicit constructor (that is, parameters that are given immediately after the structure name in the declaration) require that `val` declarations be annotated with the `DefaultValue` attribute. Structures that have a defined constructor still support zero-initialization. Therefore, the `DefaultValue` attribute is a declaration that such a zero value is valid for the field. Implicit constructors for structures do not perform any actions because `let` and `do` bindings aren't allowed on the type, but the implicit constructor parameter values passed in are available as private fields.

Explicit constructors might involve initialization of field values. When you have a structure that has an explicit constructor, it still supports zero-initialization; however, you do not use the `DefaultValue` attribute on the `val` declarations because it conflicts with the explicit constructor. For more information about `val` declarations, see Explicit Fields: The `val` Keyword.

Attributes and accessibility modifiers are allowed on structures, and follow the same rules as those for other types. For more information, see Attributes and Access Control.

The following code examples illustrate structure definitions.

```
// In Point3D, three immutable values are defined.
// x, y, and z will be initialized to 0.0.
type Point3D =
    struct
        val x: float
        val y: float
        val z: float
    end

// In Point2D, two immutable values are defined.
// It also has a member which computes a distance between itself and another Point2D.
// Point2D has an explicit constructor.
// You can create zero-initialized instances of Point2D, or you can
// pass in arguments to initialize the values.
type Point2D =
    struct
        val X: float
        val Y: float
        new(x: float, y: float) = { X = x; Y = y }

        member this.GetDistanceFrom(p: Point2D) =
            let dX = (p.X - this.X) ** 2.0
            let dY = (p.Y - this.Y) ** 2.0

            dX + dY
            |> sqrt
    end
```

ByRefLike structs

You can define your own structs that can adhere to `byref`-like semantics: see Byrefs for more information. This is done with the IsByRefLikeAttribute attribute:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` does not imply `Struct`. Both must be present on the type.

A "`byref`-like" struct in F# is a stack-bound value type. It is never allocated on the managed heap. A `byref`-like struct is useful for high-performance programming, as it is enforced with set of strong checks about lifetime and non-capture. The rules are:

- They can be used as function parameters, method parameters, local variables, method returns.
- They cannot be static or instance members of a class or normal struct.
- They cannot be captured by any closure construct (`async` methods or lambda expressions).
- They cannot be used as a generic parameter.

Although these rules very strongly restrict usage, they do so to fulfill the promise of high-performance computing in a safe manner.

ReadOnly structs

You can annotate structs with the IsReadOnlyAttribute attribute. For example:

```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsReadOnly` does not imply `Struct`. You must add both to have an `IsReadOnly` struct.

Use of this attribute emits metadata letting F# and C# know to treat it as `inref<'T>` and `in ref`, respectively.

Defining a mutable value inside of a readonly struct produces an error.

Struct Records and Discriminated Unions

You can represent Records and Discriminated Unions as structs with the `[<Struct>]` attribute. See each article to learn more.

See also

- F# Language Reference
- Classes
- Records
- Members

Inheritance

7/30/2019 • 3 minutes to read • Edit Online

Inheritance is used to model the "is-a" relationship, or subtyping, in object-oriented programming.

Specifying Inheritance Relationships

You specify inheritance relationships by using the `inherit` keyword in a class declaration. The basic syntactical form is shown in the following example.

```
type MyDerived(...) =
    inherit MyBase(...)
```

A class can have at most one direct base class. If you do not specify a base class by using the `inherit` keyword, the class implicitly inherits from `System.Object`.

Inherited Members

If a class inherits from another class, the methods and members of the base class are available to users of the derived class as if they were direct members of the derived class.

Any let bindings and constructor parameters are private to a class and, therefore, cannot be accessed from derived classes.

The keyword `base` is available in derived classes and refers to the base class instance. It is used like the self-identifier.

Virtual Methods and Overrides

Virtual methods (and properties) work somewhat differently in F# as compared to other .NET languages. To declare a new virtual member, you use the `abstract` keyword. You do this regardless of whether you provide a default implementation for that method. Thus a complete definition of a virtual method in a base class follows this pattern:

```
abstract member [method-name] : [type]

default [self-identifier].[method-name] [argument-list] = [method-body]
```

And in a derived class, an override of this virtual method follows this pattern:

```
override [self-identifier].[method-name] [argument-list] = [method-body]
```

If you omit the default implementation in the base class, the base class becomes an abstract class.

The following code example illustrates the declaration of a new virtual method `function1` in a base class and how to override it in a derived class.

```
type MyClassBase1() =
   let mutable z = 0
   abstract member function1 : int -> int
   default u.function1(a : int) = z <- z + a; z

type MyClassDerived1() =
   inherit MyClassBase1()
   override u.function1(a: int) = a + 1
```

Constructors and Inheritance

The constructor for the base class must be called in the derived class. The arguments for the base class constructor appear in the argument list in the `inherit` clause. The values that are used must be determined from the arguments supplied to the derived class constructor.

The following code shows a base class and a derived class, where the derived class calls the base class constructor in the inherit clause:

```
type MyClassBase2(x: int) =
   let mutable z = x * x
   do for i in 1..z do printf "%d " i


type MyClassDerived2(y: int) =
   inherit MyClassBase2(y * 2)
   do for i in 1..y do printf "%d " i
```

In the case of multiple constructors, the following code can be used. The first line of the derived class constructors is the `inherit` clause, and the fields appear as explicit fields that are declared with the `val` keyword. For more information, see Explicit Fields: The `val` Keyword.

```
type BaseClass =
    val string1 : string
    new (str) = { string1 = str }
    new () = { string1 = "" }

type DerivedClass =
    inherit BaseClass

    val string2 : string
    new (str1, str2) = { inherit BaseClass(str1); string2 = str2 }
    new (str2) = { inherit BaseClass(); string2 = str2 }

let obj1 = DerivedClass("A", "B")
let obj2 = DerivedClass("A")
```

Alternatives to Inheritance

In cases where a minor modification of a type is required, consider using an object expression as an alternative to inheritance. The following example illustrates the use of an object expression as an alternative to creating a new derived type:

```
open System

let object1 = { new Object() with
      override this.ToString() = "This overrides object.ToString()"
      }

printfn "%s" (object1.ToString())
```

For more information about object expressions, see Object Expressions.

When you are creating object hierarchies, consider using a discriminated union instead of inheritance. Discriminated unions can also model varied behavior of different objects that share a common overall type. A single discriminated union can often eliminate the need for a number of derived classes that are minor variations of each other. For information about discriminated unions, see Discriminated Unions.

See also

- Object Expressions
- F# Language Reference

Interfaces

7/30/2019 • 3 minutes to read • Edit Online

Interfaces specify sets of related members that other classes implement.

Syntax

```
// Interface declaration:
[ attributes ]
type [accessibility-modifier] interface-name =
    [ interface ]     [ inherit base-interface-name ...]
    abstract member1 : [ argument-types1 -> ] return-type1
    abstract member2 : [ argument-types2 -> ] return-type2
    ...
[ end ]

// Implementing, inside a class type definition:
interface interface-name with
    member self-identifier.member1argument-list = method-body1
    member self-identifier.member2argument-list = method-body2

// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
    { new interface-name with
        member self-identifier.member1argument-list = method-body1
        member self-identifier.member2argument-list = method-body2
        [ base-interface-definitions ]
    }
    member-list
```

Remarks

Interface declarations resemble class declarations except that no members are implemented. Instead, all the members are abstract, as indicated by the keyword `abstract`. You do not provide a method body for abstract methods. However, you can provide a default implementation by also including a separate definition of the member as a method together with the `default` keyword. Doing so is equivalent to creating a virtual method in a base class in other .NET languages. Such a virtual method can be overridden in classes that implement the interface.

The default accessibility for interfaces is `public`.

You can optionally give each method parameter a name using normal F# syntax:

```
type ISprintable =
    abstract member Print : format:string -> unit
```

In the above `ISprintable` example, the `Print` method has a single parameter of the type `string` with the name `format`.

There are two ways to implement interfaces: by using object expressions, and by using class types. In either case, the class type or object expression provides method bodies for abstract methods of the interface. Implementations are specific to each type that implements the interface. Therefore, interface methods on different types might be different from each other.

The keywords `interface` and `end`, which mark the start and end of the definition, are optional when you use lightweight syntax. If you do not use these keywords, the compiler attempts to infer whether the type is a class or an interface by analyzing the constructs that you use. If you define a member or use other class syntax, the type is interpreted as a class.

The .NET coding style is to begin all interfaces with a capital `I`.

Implementing Interfaces by Using Class Types

You can implement one or more interfaces in a class type by using the `interface` keyword, the name of the interface, and the `with` keyword, followed by the interface member definitions, as shown in the following code.

```
type IPrintable =
   abstract member Print : unit -> unit

type SomeClass1(x: int, y: float) =
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y
```

Interface implementations are inherited, so any derived classes do not need to reimplement them.

Calling Interface Methods

Interface methods can be called only through the interface, not through any object of the type that implements the interface. Thus, you might have to upcast to the interface type by using the `:>` operator or the `upcast` operator in order to call these methods.

To call the interface method when you have an object of type `SomeClass`, you must upcast the object to the interface type, as shown in the following code.

```
let x1 = new SomeClass1(1, 2.0)
(x1 :> IPrintable).Print()
```

An alternative is to declare a method on the object that upcasts and calls the interface method, as in the following example.

```
type SomeClass2(x: int, y: float) =
   member this.Print() = (this :> IPrintable).Print()
   interface IPrintable with
      member this.Print() = printfn "%d %f" x y

let x2 = new SomeClass2(1, 2.0)
x2.Print()
```

Implementing Interfaces by Using Object Expressions

Object expressions provide a short way to implement an interface. They are useful when you do not have to create a named type, and you just want an object that supports the interface methods, without any additional methods. An object expression is illustrated in the following code.

```
let makePrintable(x: int, y: float) =
    { new IPrintable with
              member this.Print() = printfn "%d %f" x y }
let x3 = makePrintable(1, 2.0)
x3.Print()
```

Interface Inheritance

Interfaces can inherit from one or more base interfaces.

```
type Interface1 =
    abstract member Method1 : int -> int

type Interface2 =
    abstract member Method2 : int -> int

type Interface3 =
    inherit Interface1
    inherit Interface2
    abstract member Method3 : int -> int

type MyClass() =
    interface Interface3 with
        member this.Method1(n) = 2 * n
        member this.Method2(n) = n + 100
        member this.Method3(n) = n / 10
```

See also

- F# Language Reference
- Object Expressions
- Classes

Abstract Classes

7/30/2019 • 4 minutes to read • Edit Online

Abstract classes are classes that leave some or all members unimplemented, so that implementations can be provided by derived classes.

Syntax

```
// Abstract class syntax.
[<AbstractClass>]
type [ accessibility-modifier ] abstract-class-name =
[ inherit base-class-or-interface-name ]
[ abstract-member-declarations-and-member-definitions ]

// Abstract member syntax.
abstract member member-name : type-signature
```

Remarks

In object-oriented programming, an abstract class is used as a base class of a hierarchy, and represents common functionality of a diverse set of object types. As the name "abstract" implies, abstract classes often do not correspond directly onto concrete entities in the problem domain. However, they do represent what many different concrete entities have in common.

Abstract classes must have the `AbstractClass` attribute. They can have implemented and unimplemented members. The use of the term *abstract* when applied to a class is the same as in other .NET languages; however, the use of the term *abstract* when applied to methods (and properties) is a little different in F# from its use in other .NET languages. In F#, when a method is marked with the `abstract` keyword, this indicates that a member has an entry, known as a *virtual dispatch slot*, in the internal table of virtual functions for that type. In other words, the method is virtual, although the `virtual` keyword is not used in the F# language. The keyword `abstract` is used on virtual methods regardless of whether the method is implemented. The declaration of a virtual dispatch slot is separate from the definition of a method for that dispatch slot. Therefore, the F# equivalent of a virtual method declaration and definition in another .NET language is a combination of both an abstract method declaration and a separate definition, with either the `default` keyword or the `override` keyword. For more information and examples, see Methods.

A class is considered abstract only if there are abstract methods that are declared but not defined. Therefore, classes that have abstract methods are not necessarily abstract classes. Unless a class has undefined abstract methods, do not use the **AbstractClass** attribute.

In the previous syntax, *accessibility-modifier* can be `public`, `private` or `internal`. For more information, see Access Control.

As with other types, abstract classes can have a base class and one or more base interfaces. Each base class or interface appears on a separate line together with the `inherit` keyword.

The type definition of an abstract class can contain fully defined members, but it can also contain abstract members. The syntax for abstract members is shown separately in the previous syntax. In this syntax, the *type signature* of a member is a list that contains the parameter types in order and the return types, separated by `->` tokens and/or `*` tokens as appropriate for curried and tupled parameters. The syntax for abstract member type signatures is the same as that used in signature files and that shown by IntelliSense in the Visual Studio Code

Editor.

The following code illustrates an abstract class Shape, which has two non-abstract derived classes, Square and Circle. The example shows how to use abstract classes, methods, and properties. In the example, the abstract class Shape represents the common elements of the concrete entities circle and square. The common features of all shapes (in a two-dimensional coordinate system) are abstracted out into the Shape class: the position on the grid, an angle of rotation, and the area and perimeter properties. These can be overridden, except for position, the behavior of which individual shapes cannot change.

The rotation method can be overridden, as in the Circle class, which is rotation invariant because of its symmetry. So in the Circle class, the rotation method is replaced by a method that does nothing.

```
// An abstract class that has some methods and properties defined
// and some left abstract.
[<AbstractClass>]
type Shape2D(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
    let mutable rotAngle = 0.0

    // These properties are not declared abstract. They
    // cannot be overriden.
    member this.CenterX with get() = x and set xval = x <- xval
    member this.CenterY with get() = y and set yval = y <- yval

    // These properties are abstract, and no default implementation
    // is provided. Non-abstract derived classes must implement these.
    abstract Area : float with get
    abstract Perimeter : float  with get
    abstract Name : string with get

    // This method is not declared abstract. It cannot be
    // overridden.
    member this.Move dx dy =
       x <- x + dx
       y <- y + dy

    // An abstract method that is given a default implementation
    // is equivalent to a virtual method in other .NET languages.
    // Rotate changes the internal angle of rotation of the square.
    // Angle is assumed to be in degrees.
    abstract member Rotate: float -> unit
    default this.Rotate(angle) = rotAngle <- rotAngle + angle

type Square(x, y, sideLengthIn) =
    inherit Shape2D(x, y)
    member this.SideLength = sideLengthIn
    override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"

type Circle(x, y, radius) =
    inherit Shape2D(x, y)
    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
    // Rotating a circle does nothing, so use the wildcard
    // character to discard the unused argument and
    // evaluate to unit.
    override this.Rotate(_) = ()
    override this.Name = "Circle"

let square1 = new Square(0.0, 0.0, 10.0)
let circle1 = new Circle(0.0, 0.0, 5.0)
circle1.CenterX <- 1.0
circle1.CenterY <- -2.0
```

```
square1.Move -1.0 2.0
square1.Rotate 45.0
circle1.Rotate 45.0
printfn "Perimeter of square with side length %f is %f, %f"
        (square1.SideLength) (square1.Area) (square1.Perimeter)
printfn "Circumference of circle with radius %f is %f, %f"
        (circle1.Radius) (circle1.Area) (circle1.Perimeter)

let shapeList : list<Shape2D> = [ (square1 :> Shape2D);
                                  (circle1 :> Shape2D) ]
List.iter (fun (elem : Shape2D) ->
              printfn "Area of %s: %f" (elem.Name) (elem.Area))
          shapeList
```

Output:

```
Perimeter of square with side length 10.000000 is 40.000000
Circumference of circle with radius 5.000000 is 31.415927
Area of Square: 100.000000
Area of Circle: 78.539816
```

See also

- Classes
- Members
- Methods
- Properties

Members

5/15/2019 • 2 minutes to read • Edit Online

This section describes members of F# object types.

Remarks

Members are features that are part of a type definition and are declared with the `member` keyword. F# object types such as records, classes, discriminated unions, interfaces, and structures support members. For more information, see Records, Classes, Discriminated Unions, Interfaces, and Structures.

Members typically make up the public interface for a type, which is why they are public unless otherwise specified. Members can also be declared private or internal. For more information, see Access Control. Signatures for types can also be used to expose or not expose certain members of a type. For more information, see Signatures.

Private fields and `do` bindings, which are used only with classes, are not true members, because they are never part of the public interface of a type and are not declared with the `member` keyword, but they are described in this section also.

Related Topics

TOPIC	DESCRIPTION
`let` Bindings in Classes	Describes the definition of private fields and functions in classes.
`do` Bindings in Classes	Describes the specification of object initialization code.
Properties	Describes property members in classes and other types.
Indexed Properties	Describes array-like properties in classes and other types.
Methods	Describes functions that are members of a type.
Constructors	Describes special functions that initialize objects of a type.
Operator Overloading	Describes the definition of customized operators for types.
Events	Describes the definition of events and event handling support in F#.
Explicit Fields: The `val` Keyword	Describes the definition of uninitialized fields in a type.

let Bindings in Classes

7/30/2019 • 2 minutes to read • Edit Online

You can define private fields and private functions for F# classes by using `let` bindings in the class definition.

Syntax

```
// Field.
[static] let [ mutable ] binding1 [ and ... binding-n ]

// Function.
[static] let [ rec ] binding1 [ and ... binding-n ]
```

Remarks

The previous syntax appears after the class heading and inheritance declarations but before any member definitions. The syntax is like that of `let` bindings outside of classes, but the names defined in a class have a scope that is limited to the class. A `let` binding creates a private field or function; to expose data or functions publicly, declare a property or a member method.

A `let` binding that is not static is called an instance `let` binding. Instance `let` bindings execute when objects are created. Static `let` bindings are part of the static initializer for the class, which is guaranteed to execute before the type is first used.

The code within instance `let` bindings can use the primary constructor's parameters.

Attributes and accessibility modifiers are not permitted on `let` bindings in classes.

The following code examples illustrate several types of `let` bindings in classes.

```
type PointWithCounter(a: int, b: int) =
    // A variable i.
    let mutable i = 0

    // A let binding that uses a pattern.
    let (x, y) = (a, b)

    // A private function binding.
    let privateFunction x y = x * x + 2*y

    // A static let binding.
    static let mutable count = 0

    // A do binding.
    do
       count <- count + 1

    member this.Prop1 = x
    member this.Prop2 = y
    member this.CreatedCount = count
    member this.FunctionValue = privateFunction x y

let point1 = PointWithCounter(10, 52)

printfn "%d %d %d %d" (point1.Prop1) (point1.Prop2) (point1.CreatedCount) (point1.FunctionValue)
```

The output is as follows.

```
10 52 1 204
```

Alternative Ways to Create Fields

You can also use the `val` keyword to create a private field. When using the `val` keyword, the field is not given a value when the object is created, but instead is initialized with a default value. For more information, see Explicit Fields: The val Keyword.

You can also define private fields in a class by using a member definition and adding the keyword `private` to the definition. This can be useful if you expect to change the accessibility of a member without rewriting your code. For more information, see Access Control.

See also

- Members
- `do` Bindings in Classes
- `let` Bindings

do Bindings in Classes

7/30/2019 • 2 minutes to read • Edit Online

A `do` binding in a class definition performs actions when the object is constructed or, for a static `do` binding, when the type is first used.

Syntax

```
[static] do expression
```

Remarks

A `do` binding appears together with or after `let` bindings but before member definitions in a class definition. Although the `do` keyword is optional for `do` bindings at the module level, it is not optional for `do` bindings in a class definition.

For the construction of every object of any given type, non-static `do` bindings and non-static `let` bindings are executed in the order in which they appear in the class definition. Multiple `do` bindings can occur in one type. The non-static `let` bindings and the non-static `do` bindings become the body of the primary constructor. The code in the non-static `do` bindings section can reference the primary constructor parameters and any values or functions that are defined in the `let` bindings section.

Non-static `do` bindings can access members of the class as long as the class has a self identifier that is defined by an `as` keyword in the class heading, and as long as all uses of those members are qualified with the self identifier for the class.

Because `let` bindings initialize the private fields of a class, which is often necessary to guarantee that members behave as expected, `do` bindings are usually put after `let` bindings so that code in the `do` binding can execute with a fully initialized object. If your code attempts to use a member before the initialization is complete, an InvalidOperationException is raised.

Static `do` bindings can reference static members or fields of the enclosing class but not instance members or fields. Static `do` bindings become part of the static initializer for the class, which is guaranteed to execute before the class is first used.

Attributes are ignored for `do` bindings in types. If an attribute is required for code that executes in a `do` binding, it must be applied to the primary constructor.

In the following code, a class has a static `do` binding and a non-static `do` binding. The object has a constructor that has two parameters, `a` and `b`, and two private fields are defined in the `let` bindings for the class. Two properties are also defined. All of these are in scope in the non-static `do` bindings section, as is illustrated by the line that prints all those values.

```
open System

type MyType(a:int, b:int) as this =
    inherit Object()
    let x = 2*a
    let y = 2*b
    do printfn "Initializing object %d %d %d %d %d %d"
               a b x y (this.Prop1) (this.Prop2)
    static do printfn "Initializing MyType."
    member this.Prop1 = 4*x
    member this.Prop2 = 4*y
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)

let obj1 = new MyType(1, 2)
```

The output is as follows.

```
Initializing MyType.
Initializing object 1 2 2 4 8 16
```

See also

- Members
- Classes
- Constructors
- `let` Bindings in Classes
- `do` Bindings

Properties

7/30/2019 • 7 minutes to read • Edit Online

Properties are members that represent values associated with an object.

Syntax

```
// Property that has both get and set defined.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with [accessibility-modifier] get() =
    get-function-body
and [accessibility-modifier] set parameter =
    set-function-body

// Alternative syntax for a property that has get and set.
[ attributes-for-get ]
[ static ] member [accessibility-modifier-for-get] [self-identifier.]PropertyName =
    get-function-body
[ attributes-for-set ]
[ static ] member [accessibility-modifier-for-set] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName =
    get-function-body

// Alternative syntax for property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with get() =
    get-function-body

// Property that has set only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Automatically implemented properties.
[ attributes ]
[ static ] member val [accessibility-modifier] PropertyName = initialization-expression [ with get, set ]
```

Remarks

Properties represent the "has a" relationship in object-oriented programming, representing data that is associated with object instances or, for static properties, with the type.

You can declare properties in two ways, depending on whether you want to explicitly specify the underlying value (also called the backing store) for the property, or if you want to allow the compiler to automatically generate the backing store for you. Generally, you should use the more explicit way if the property has a non-trivial implementation and the automatic way when the property is just a simple wrapper for a value or variable. To declare a property explicitly, use the `member` keyword. This declarative syntax is followed by the syntax that specifies the `get` and `set` methods, also named *accessors*. The various forms of the explicit syntax shown in the

syntax section are used for read/write, read-only, and write-only properties. For read-only properties, you define only a `get` method; for write-only properties, define only a `set` method. Note that when a property has both `get` and `set` accessors, the alternative syntax enables you to specify attributes and accessibility modifiers that are different for each accessor, as is shown in the following code.

```
// A read-only property.
member this.MyReadOnlyProperty = myInternalValue
// A write-only property.
member this.MyWriteOnlyProperty with set (value) = myInternalValue <- value
// A read-write property.
member this.MyReadWriteProperty
    with get () = myInternalValue
    and set (value) = myInternalValue <- value
```

For read/write properties, which have both a `get` and `set` method, the order of `get` and `set` can be reversed. Alternatively, you can provide the syntax shown for `get` only and the syntax shown for `set` only instead of using the combined syntax. Doing this makes it easier to comment out the individual `get` or `set` method, if that is something you might need to do. This alternative to using the combined syntax is shown in the following code.

```
member this.MyReadWriteProperty with get () = myInternalValue
member this.MyReadWriteProperty with set (value) = myInternalValue <- value
```

Private values that hold the data for properties are called *backing stores*. To have the compiler create the backing store automatically, use the keywords `member val`, omit the self-identifier, then provide an expression to initialize the property. If the property is to be mutable, include `with get, set`. For example, the following class type includes two automatically implemented properties. `Property1` is read-only and is initialized to the argument provided to the primary constructor, and `Property2` is a settable property initialized to an empty string:

```
type MyClass(property1 : int) =
member val Property1 = property1
member val Property2 = "" with get, set
```

Automatically implemented properties are part of the initialization of a type, so they must be included before any other member definitions, just like `let` bindings and `do` bindings in a type definition. Note that the expression that initializes an automatically implemented property is only evaluated upon initialization, and not every time the property is accessed. This behavior is in contrast to the behavior of an explicitly implemented property. What this effectively means is that the code to initialize these properties is added to the constructor of a class. Consider the following code that shows this difference:

```
type MyClass() =
    let random  = new System.Random()
    member val AutoProperty = random.Next() with get, set
    member this.ExplicitProperty = random.Next()

let class1 = new MyClass()

printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.AutoProperty = %d" class1.AutoProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty
printfn "class1.ExplicitProperty = %d" class1.ExplicitProperty
```

Output

```
class1.AutoProperty = 1853799794
class1.AutoProperty = 1853799794
class1.ExplicitProperty = 978922705
class1.ExplicitProperty = 1131210765
```

The output of the preceding code shows that the value of AutoProperty is unchanged when called repeatedly, whereas the ExplicitProperty changes each time it is called. This demonstrates that the expression for an automatically implemented property is not evaluated each time, as is the getter method for the explicit property.

> **WARNING**
> There are some libraries, such as the Entity Framework (`System.Data.Entity`) that perform custom operations in base class constructors that don't work well with the initialization of automatically implemented properties. In those cases, try using explicit properties.

Properties can be members of classes, structures, discriminated unions, records, interfaces, and type extensions and can also be defined in object expressions.

Attributes can be applied to properties. To apply an attribute to a property, write the attribute on a separate line before the property. For more information, see Attributes.

By default, properties are public. Accessibility modifiers can also be applied to properties. To apply an accessibility modifier, add it immediately before the name of the property if it is meant to apply to both the `get` and `set` methods; add it before the `get` and `set` keywords if different accessibility is required for each accessor. The *accessibility-modifier* can be one of the following: `public` , `private` , `internal` . For more information, see Access Control.

Property implementations are executed each time a property is accessed.

Static and Instance Properties

Properties can be static or instance properties. Static properties can be invoked without an instance and are used for values associated with the type, not with individual objects. For static properties, omit the self-identifier. The self-identifier is required for instance properties.

The following static property definition is based on a scenario in which you have a static field `myStaticValue` that is the backing store for the property.

```
static member MyStaticProperty
    with get() = myStaticValue
    and set(value) = myStaticValue <- value
```

Properties can also be array-like, in which case they are called *indexed properties*. For more information, see Indexed Properties.

Type Annotation for Properties

In many cases, the compiler has enough information to infer the type of a property from the type of the backing store, but you can set the type explicitly by adding a type annotation.

```
// To apply a type annotation to a property that does not have an explicit
// get or set, apply the type annotation directly to the property.
member this.MyProperty1 : int = myInternalValue
// If there is a get or set, apply the type annotation to the get or set method.
member this.MyProperty2 with get() : int = myInternalValue
```

Using Property set Accessors

You can set properties that provide `set` accessors by using the `<-` operator.

```
// Assume that the constructor argument sets the initial value of the
// internal backing store.
let mutable myObject = new MyType(10)
myObject.MyProperty <- 20
printfn "%d" (myObject.MyProperty)
```

The output is **20**.

Abstract Properties

Properties can be abstract. As with methods, `abstract` just means that there is a virtual dispatch associated with the property. Abstract properties can be truly abstract, that is, without a definition in the same class. The class that contains such a property is therefore an abstract class. Alternatively, abstract can just mean that a property is virtual, and in that case, a definition must be present in the same class. Note that abstract properties must not be private, and if one accessor is abstract, the other must also be abstract. For more information about abstract classes, see Abstract Classes.

```
// Abstract property in abstract class.
// The property is an int type that has a get and
// set method
[<AbstractClass>]
type AbstractBase() =
   abstract Property1 : int with get, set

// Implementation of the abstract property
type Derived1() =
   inherit AbstractBase()
   let mutable value = 10
   override this.Property1 with get() = value and set(v : int) = value <- v

// A type with a "virtual" property.
 type Base1() =
   let mutable value = 10
   abstract Property1 : int with get, set
   default this.Property1 with get() = value and set(v : int) = value <- v

// A derived type that overrides the virtual property
type Derived2() =
   inherit Base1()
   let mutable value2 = 11
   override this.Property1 with get() = value2 and set(v) = value2 <- v
```

See also

- Members
- Methods

Indexed Properties

7/30/2019 • 3 minutes to read • Edit Online

When defining a class that abstracts over ordered data, it can sometimes be helpful to provide indexed access to that data without exposing the underlying implementation. This is done with the `Item` member.

Syntax

```
// Indexed property that can be read and written to
member self-identifier.Item
    with get(index-values) =
        get-member-body
    and set index-values values-to-set =
        set-member-body

// Indexed property can only be read
member self-identifier.Item
    with get(index-values) =
        get-member-body

// Indexed property that can only be set
member self-identifier.Item
    with set index-values values-to-set =
        set-member-body
```

Remarks

The forms of the previous syntax show how to define indexed properties that have both a `get` and a `set` method, have a `get` method only, or have a `set` method only. You can also combine both the syntax shown for get only and the syntax shown for set only, and produce a property that has both get and set. This latter form allows you to put different accessibility modifiers and attributes on the get and set methods.

By using the name `Item`, the compiler treats the property as a default indexed property. A *default indexed property* is a property that you can access by using array-like syntax on the object instance. For example, if `o` is an object of the type that defines this property, the syntax `o.[index]` is used to access the property.

The syntax for accessing a non-default indexed property is to provide the name of the property and the index in parentheses, just like a regular member. For example, if the property on `o` is called `Ordinal`, you write `o.Ordinal(index)` to access it.

Regardless of which form you use, you should always use the curried form for the set method on an indexed property. For information about curried functions, see Functions.

Example

The following code example illustrates the definition and use of default and non-default indexed properties that have get and set methods.

```
type NumberStrings() =
   let mutable ordinals = [| "one"; "two"; "three"; "four"; "five";
                             "six"; "seven"; "eight"; "nine"; "ten" |]
   let mutable cardinals = [| "first"; "second"; "third"; "fourth";
                              "fifth"; "sixth"; "seventh"; "eighth";
                              "ninth"; "tenth" |]
   member this.Item
      with get(index) = ordinals.[index]
      and set index value = ordinals.[index] <- value
   member this.Ordinal
      with get(index) = ordinals.[index]
      and set index value = ordinals.[index] <- value
   member this.Cardinal
      with get(index) = cardinals.[index]
      and set index value = cardinals.[index] <- value

let nstrs = new NumberStrings()
nstrs.[0] <- "ONE"
for i in 0 .. 9 do
  printf "%s " (nstrs.[i])
printfn ""

nstrs.Cardinal(5) <- "6th"

for i in 0 .. 9 do
  printf "%s " (nstrs.Ordinal(i))
  printf "%s " (nstrs.Cardinal(i))
printfn ""
```

Output

```
ONE two three four five six seven eight nine ten
ONE first two second three third four fourth five fifth six 6th
seven seventh eight eighth nine ninth ten tenth
```

Indexed Properties with multiple index values

Indexed properties can have more than one index value. In that case, the values are separated by commas when the property is used. The set method in such a property must have two curried arguments, the first of which is a tuple containing the keys, and the second of which is the value to set.

The following code demonstrates the use of an indexed property with multiple index values.

```
open System.Collections.Generic

/// Basic implementation of a sparse matrix based on a dictionary
type SparseMatrix() =
    let table = new Dictionary<(int * int), float>()
    member __.Item
        // Because the key is comprised of two values, 'get' has two index values
        with get(key1, key2) = table.[(key1, key2)]

        // 'set' has two index values and a new value to place in the key's position
        and set (key1, key2) value = table.[(key1, key2)] <- value

let sm = new SparseMatrix()
for i in 1..1000 do
    sm.[i, i] <- float i * float i
```

See also

- Members

Methods

7/30/2019 • 7 minutes to read • Edit Online

A *method* is a function that is associated with a type. In object-oriented programming, methods are used to expose and implement the functionality and behavior of objects and types.

Syntax

```
// Instance method definition.
[ attributes ]
member [inline] self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Static method definition.
[ attributes ]
static member [inline] method-name parameter-list [ : return-type ] =
    method-body

// Abstract method declaration or virtual dispatch slot.
[ attributes ]
abstract member method-name : type-signature

// Virtual method declaration and default implementation.
[ attributes ]
abstract member method-name : type-signature
[ attributes ]
default self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Override of inherited virtual method.
[ attributes ]
override self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Optional and DefaultParameterValue attributes on input parameters
[ attributes ]
[ modifier ] member [inline] self-identifier.method-name ([<Optional; DefaultParameterValue( default-value
)>] input) [ : return-type ]
```

Remarks

In the previous syntax, you can see the various forms of method declarations and definitions. In longer method bodies, a line break follows the equal sign (=), and the whole method body is indented.

Attributes can be applied to any method declaration. They precede the syntax for a method definition and are usually listed on a separate line. For more information, see Attributes.

Methods can be marked `inline`. For information about `inline`, see Inline Functions.

Non-inline methods can be used recursively within the type; there is no need to explicitly use the `rec` keyword.

Instance Methods

Instance methods are declared with the `member` keyword and a *self-identifier*, followed by a period (.) and the method name and parameters. As is the case for `let` bindings, the *parameter-list* can be a pattern. Typically, you enclose method parameters in parentheses in a tuple form, which is the way methods appear in F# when they are

created in other .NET Framework languages. However, the curried form (parameters separated by spaces) is also common, and other patterns are supported also.

The following example illustrates the definition and use of a non-abstract instance method.

```
type SomeType(factor0: int) =
   let factor = factor0
   member this.SomeMethod(a, b, c) =
      (a + b + c) * factor

   member this.SomeOtherMethod(a, b, c) =
      this.SomeMethod(a, b, c) * factor
```

Within instance methods, do not use the self identifier to access fields defined by using let bindings. Use the self identifier when accessing other members and properties.

Static Methods

The keyword `static` is used to specify that a method can be called without an instance and is not associated with an object instance. Otherwise, methods are instance methods.

The example in the next section shows fields declared with the `let` keyword, property members declared with the `member` keyword, and a static method declared with the `static` keyword.

The following example illustrates the definition and use of static methods. Assume that these method definitions are in the `SomeType` class in the previous section.

```
static member SomeStaticMethod(a, b, c) =
   (a + b + c)

static member SomeOtherStaticMethod(a, b, c) =
   SomeType.SomeStaticMethod(a, b, c) * 100
```

Abstract and Virtual Methods

The keyword `abstract` indicates that a method has a virtual dispatch slot and might not have a definition in the class. A *virtual dispatch slot* is an entry in an internally maintained table of functions that is used at run time to look up virtual function calls in an object-oriented type. The virtual dispatch mechanism is the mechanism that implements *polymorphism*, an important feature of object-oriented programming. A class that has at least one abstract method without a definition is an *abstract class*, which means that no instances can be created of that class. For more information about abstract classes, see Abstract Classes.

Abstract method declarations do not include a method body. Instead, the name of the method is followed by a colon (:) and a type signature for the method. The type signature of a method is the same as that shown by IntelliSense when you pause the mouse pointer over a method name in the Visual Studio Code Editor, except without parameter names. Type signatures are also displayed by the interpreter, fsi.exe, when you are working interactively. The type signature of a method is formed by listing out the types of the parameters, followed by the return type, with appropriate separator symbols. Curried parameters are separated by `->` and tuple parameters are separated by `*`. The return value is always separated from the arguments by a `->` symbol. Parentheses can be used to group complex parameters, such as when a function type is a parameter, or to indicate when a tuple is treated as a single parameter rather than as two parameters.

You can also give abstract methods default definitions by adding the definition to the class and using the `default` keyword, as shown in the syntax block in this topic. An abstract method that has a definition in the same class is equivalent to a virtual method in other .NET Framework languages. Whether or not a definition exists, the

`abstract` keyword creates a new dispatch slot in the virtual function table for the class.

Regardless of whether a base class implements its abstract methods, derived classes can provide implementations of abstract methods. To implement an abstract method in a derived class, define a method that has the same name and signature in the derived class, except use the `override` or `default` keyword, and provide the method body. The keywords `override` and `default` mean exactly the same thing. Use `override` if the new method overrides a base class implementation; use `default` when you create an implementation in the same class as the original abstract declaration. Do not use the `abstract` keyword on the method that implements the method that was declared abstract in the base class.

The following example illustrates an abstract method `Rotate` that has a default implementation, the equivalent of a .NET Framework virtual method.

```
type Ellipse(a0 : float, b0 : float, theta0 : float) =
    let mutable axis1 = a0
    let mutable axis2 = b0
    let mutable rotAngle = theta0
    abstract member Rotate: float -> unit
    default this.Rotate(delta : float) = rotAngle <- rotAngle + delta
```

The following example illustrates a derived class that overrides a base class method. In this case, the override changes the behavior so that the method does nothing.

```
type Circle(radius : float) =
    inherit Ellipse(radius, radius, 0.0)
     // Circles are invariant to rotation, so do nothing.
    override this.Rotate(_) = ()
```

Overloaded Methods

Overloaded methods are methods that have identical names in a given type but that have different arguments. In F#, optional arguments are usually used instead of overloaded methods. However, overloaded methods are permitted in the language, provided that the arguments are in tuple form, not curried form.

Optional Arguments

Starting with F# 4.1, you can also have optional arguments with a default parameter value in methods. This is to help facilitate interoperation with C# code. The following example demonstrates the syntax:

```
// A class with a method M, which takes in an optional integer argument.
type C() =
    __.M([<Optional; DefaultParameterValue(12)>] i) = i + 1
```

Note that the value passed in for `DefaultParameterValue` must match the input type. In the above sample, it is an `int`. Attempting to pass a non-integer value into `DefaultParameterValue` would result in a compile error.

Example: Properties and Methods

The following example contains a type that has examples of fields, private functions, properties, and a static method.

```
type RectangleXY(x1 : float, y1: float, x2: float, y2: float) =
    // Field definitions.
    let height = y2 - y1
    let width = x2 - x1
    let area = height * width
    // Private functions.
    static let maxFloat (x: float) (y: float) =
      if x >= y then x else y
    static let minFloat (x: float) (y: float) =
      if x <= y then x else y
    // Properties.
    // Here, "this" is used as the self identifier,
    // but it can be any identifier.
    member this.X1 = x1
    member this.Y1 = y1
    member this.X2 = x2
    member this.Y2 = y2
    // A static method.
    static member intersection(rect1 : RectangleXY, rect2 : RectangleXY) =
       let x1 = maxFloat rect1.X1 rect2.X1
       let y1 = maxFloat rect1.Y1 rect2.Y1
       let x2 = minFloat rect1.X2 rect2.X2
       let y2 = minFloat rect1.Y2 rect2.Y2
       let result : RectangleXY option =
         if ( x2 > x1 && y2 > y1) then
           Some (RectangleXY(x1, y1, x2, y2))
         else
           None
       result

// Test code.
let testIntersection =
    let r1 = RectangleXY(10.0, 10.0, 20.0, 20.0)
    let r2 = RectangleXY(15.0, 15.0, 25.0, 25.0)
    let r3 : RectangleXY option = RectangleXY.intersection(r1, r2)
    match r3 with
    | Some(r3) -> printfn "Intersection rectangle: %f %f %f %f" r3.X1 r3.Y1 r3.X2 r3.Y2
    | None -> printfn "No intersection found."

testIntersection
```

See also

- Members

Constructors

7/30/2019 • 6 minutes to read • Edit Online

This topic describes how to define and use constructors to create and initialize class and structure objects.

Construction of Class Objects

Objects of class types have constructors. There are two kinds of constructors. One is the primary constructor, whose parameters appear in parentheses just after the type name. You specify other, optional additional constructors by using the `new` keyword. Any such additional constructors must call the primary constructor.

The primary constructor contains `let` and `do` bindings that appear at the start of the class definition. A `let` binding declares private fields and methods of the class; a `do` binding executes code. For more information about `let` bindings in class constructors, see `let` Bindings in Classes. For more information about `do` bindings in constructors, see `do` Bindings in Classes.

Regardless of whether the constructor you want to call is a primary constructor or an additional constructor, you can create objects by using a `new` expression, with or without the optional `new` keyword. You initialize your objects together with constructor arguments, either by listing the arguments in order and separated by commas and enclosed in parentheses, or by using named arguments and values in parentheses. You can also set properties on an object during the construction of the object by using the property names and assigning values just as you use named constructor arguments.

The following code illustrates a class that has a constructor and various ways of creating objects.

```
// This class has a primary constructor that takes three arguments
// and an additional constructor that calls the primary constructor.
type MyClass(x0, y0, z0) =
    let mutable x = x0
    let mutable y = y0
    let mutable z = z0
    do
        printfn "Initialized object that has coordinates (%d, %d, %d)" x y z
    member this.X with get() = x and set(value) = x <- value
    member this.Y with get() = y and set(value) = y <- value
    member this.Z with get() = z and set(value) = z <- value
    new() = MyClass(0, 0, 0)

// Create by using the new keyword.
let myObject1 = new MyClass(1, 2, 3)
// Create without using the new keyword.
let myObject2 = MyClass(4, 5, 6)
// Create by using named arguments.
let myObject3 = MyClass(x0 = 7, y0 = 8, z0 = 9)
// Create by using the additional constructor.
let myObject4 = MyClass()
```

The output is as follows.

```
Initialized object that has coordinates (1, 2, 3)
Initialized object that has coordinates (4, 5, 6)
Initialized object that has coordinates (7, 8, 9)
Initialized object that has coordinates (0, 0, 0)
```

Construction of Structures

Structures follow all the rules of classes. Therefore, you can have a primary constructor, and you can provide additional constructors by using `new`. However, there is one important difference between structures and classes: structures can have a parameterless constructor (that is, one with no arguments) even if no primary constructor is defined. The parameterless constructor initializes all the fields to the default value for that type, usually zero or its equivalent. Any constructors that you define for structures must have at least one argument so that they do not conflict with the default constructor.

Also, structures often have fields that are created by using the `val` keyword; classes can also have these fields. Structures and classes that have fields defined by using the `val` keyword can also be initialized in additional constructors by using record expressions, as shown in the following code.

```
type MyStruct =
    struct
       val X : int
       val Y : int
       val Z : int
       new(x, y, z) = { X = x; Y = y; Z = z }
    end

let myStructure1 = new MyStruct(1, 2, 3)
```

For more information, see Explicit Fields: The `val` Keyword.

Executing Side Effects in Constructors

A primary constructor in a class can execute code in a `do` binding. However, what if you have to execute code in an additional constructor, without a `do` binding? To do this, you use the `then` keyword.

```
 // Executing side effects in the primary constructor and
// additional constructors.
type Person(nameIn : string, idIn : int) =
    let mutable name = nameIn
    let mutable id = idIn
    do printfn "Created a person object."
    member this.Name with get() = name and set(v) = name <- v
    member this.ID with get() = id and set(v) = id <- v
    new() =
        Person("Invalid Name", -1)
        then
            printfn "Created an invalid person object."

let person1 = new Person("Humberto Acevedo", 123458734)
let person2 = new Person()
```

The side effects of the primary constructor still execute. Therefore, the output is as follows.

```
Created a person object.
Created a person object.
Created an invalid person object.
```

Self Identifiers in Constructors

In other members, you provide a name for the current object in the definition of each member. You can also put the self identifier on the first line of the class definition by using the `as` keyword immediately following the constructor parameters. The following example illustrates this syntax.

```
type MyClass1(x) as this =
    // This use of the self identifier produces a warning - avoid.
    let x1 = this.X
    // This use of the self identifier is acceptable.
    do printfn "Initializing object with X =%d" this.X
    member this.X = x
```

In additional constructors, you can also define a self identifier by putting the `as` clause right after the constructor parameters. The following example illustrates this syntax.

```
type MyClass2(x : int) =
    member this.X = x
    new() as this = MyClass2(0) then printfn "Initializing with X = %d" this.X
```

Problems can occur when you try to use an object before it is fully defined. Therefore, uses of the self identifier can cause the compiler to emit a warning and insert additional checks to ensure the members of an object are not accessed before the object is initialized. You should only use the self identifier in the `do` bindings of the primary constructor, or after the `then` keyword in additional constructors.

The name of the self identifier does not have to be `this`. It can be any valid identifier.

Assigning Values to Properties at Initialization

You can assign values to the properties of a class object in the initialization code by appending a list of assignments of the form `property = value` to the argument list for a constructor. This is shown in the following code example.

```
 type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
       with get() = number
       and set(value) = number <- value
    member this.FirstName
       with get() = firstName
       and set(value) = firstName <- value
    member this.LastName
       with get() = lastName
       and set(value) = lastName <- value
    member this.Balance
       with get() = balance
       and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

The following version of the previous code illustrates the combination of ordinary arguments, optional arguments, and property settings in one constructor call.

```
type Account(accountNumber : int, ?first: string, ?last: string, ?bal : float) =
   let mutable balance = defaultArg bal 0.0
   let mutable number = accountNumber
   let mutable firstName = defaultArg first ""
   let mutable lastName = defaultArg last ""
   member this.AccountNumber
      with get() = number
      and set(value) = number <- value
   member this.FirstName
      with get() = firstName
      and set(value) = firstName <- value
   member this.LastName
      with get() = lastName
      and set(value) = lastName <- value
   member this.Balance
      with get() = balance
      and set(value) = balance <- value
   member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
   member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(8782108, bal = 543.33,
                           FirstName="Raman", LastName="Iyer")
```

Constructors in inherited class

When inheriting from a base class that has a constructor, you must specify its arguments in the inherit clause. For more information, see Constructors and inheritance.

Static Constructors or Type Constructors

In addition to specifying code for creating objects, static `let` and `do` bindings can be authored in class types that execute before the type is first used to perform initialization at the type level. For more information, see `let` Bindings in Classes and `do` Bindings in Classes.

See also

- Members

Events

8/22/2019 • 6 minutes to read • Edit Online

NOTE

The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

Events enable you to associate function calls with user actions and are important in GUI programming. Events can also be triggered by your applications or by the operating system.

Handling Events

When you use a GUI library like Windows Forms or Windows Presentation Foundation (WPF), much of the code in your application runs in response to events that are predefined by the library. These predefined events are members of GUI classes such as forms and controls. You can add custom behavior to a preexisting event, such as a button click, by referencing the specific named event of interest (for example, the `Click` event of the `Form` class) and invoking the `Add` method, as shown in the following code. If you run this from F# Interactive, omit the call to `System.Windows.Forms.Application.Run(System.Windows.Forms.Form)`.

```
open System.Windows.Forms

let form = new Form(Text="F# Windows Form",
                    Visible = true,
                    TopMost = true)

form.Click.Add(fun evArgs -> System.Console.Beep())
Application.Run(form)
```

The type of the `Add` method is `('a -> unit) -> unit`. Therefore, the event handler method takes one parameter, typically the event arguments, and returns `unit`. The previous example shows the event handler as a lambda expression. The event handler can also be a function value, as in the following code example. The following code example also shows the use of the event handler parameters, which provide information specific to the type of event. For a `MouseMove` event, the system passes a `System.Windows.Forms.MouseEventArgs` object, which contains the `X` and `Y` position of the pointer.

```
open System.Windows.Forms

let Beep evArgs =
    System.Console.Beep( )


let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)

let MouseMoveEventHandler (evArgs : System.Windows.Forms.MouseEventArgs) =
    form.Text <- System.String.Format("{0},{1}", evArgs.X, evArgs.Y)

form.Click.Add(Beep)
form.MouseMove.Add(MouseMoveEventHandler)
Application.Run(form)
```

Creating Custom Events

F# events are represented by the F# Event class, which implements the IEvent interface. `IEvent` is itself an interface that combines the functionality of two other interfaces, `System.IObservable<'T>` and IDelegateEvent. Therefore, `Event`s have the equivalent functionality of delegates in other languages, plus the additional functionality from `IObservable`, which means that F# events support event filtering and using F# first-class functions and lambda expressions as event handlers. This functionality is provided in the Event module.

To create an event on a class that acts just like any other .NET Framework event, add to the class a `let` binding that defines an `Event` as a field in a class. You can specify the desired event argument type as the type argument, or leave it blank and have the compiler infer the appropriate type. You also must define an event member that exposes the event as a CLI event. This member should have the CLIEvent attribute. It is declared like a property and its implementation is just a call to the Publish property of the event. Users of your class can use the `Add` method of the published event to add a handler. The argument for the `Add` method can be a lambda expression. You can use the `Trigger` property of the event to raise the event, passing the arguments to the handler function. The following code example illustrates this. In this example, the inferred type argument for the event is a tuple, which represents the arguments for the lambda expression.

```
open System.Collections.Generic

type MyClassWithCLIEvent() =

    let event1 = new Event<_>()

    [<CLIEvent>]
    member this.Event1 = event1.Publish

    member this.TestEvent(arg) =
        event1.Trigger(this, arg)

let classWithEvent = new MyClassWithCLIEvent()
classWithEvent.Event1.Add(fun (sender, arg) ->
        printfn "Event1 occurred! Object data: %s" arg)

classWithEvent.TestEvent("Hello World!")

System.Console.ReadLine() |> ignore
```

The output is as follows.

```
Event1 occurred! Object data: Hello World!
```

The additional functionality provided by the `Event` module is illustrated here. The following code example illustrates the basic use of `Event.create` to create an event and a trigger method, add two event handlers in the form of lambda expressions, and then trigger the event to execute both lambda expressions.

```
type MyType() =
    let myEvent = new Event<_>()

    member this.AddHandlers() =
       Event.add (fun string1 -> printfn "%s" string1) myEvent.Publish
       Event.add (fun string1 -> printfn "Given a value: %s" string1) myEvent.Publish

    member this.Trigger(message) =
       myEvent.Trigger(message)

let myMyType = MyType()
myMyType.AddHandlers()
myMyType.Trigger("Event occurred.")
```

The output of the previous code is as follows.

```
Event occurred.
Given a value: Event occurred.
```

Processing Event Streams

Instead of just adding an event handler for an event by using the Event.add function, you can use the functions in the `Event` module to process streams of events in highly customized ways. To do this, you use the forward pipe (`|>`) together with the event as the first value in a series of function calls, and the `Event` module functions as subsequent function calls.

The following code example shows how to set up an event for which the handler is only called under certain conditions.

```
let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)
form.MouseMove
    |> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)
    |> Event.add ( fun evArgs ->
        form.BackColor <- System.Drawing.Color.FromArgb(
            evArgs.X, evArgs.Y, evArgs.X ^^^ evArgs.Y) )
```

The Observable module contains similar functions that operate on observable objects. Observable objects are similar to events but only actively subscribe to events if they themselves are being subscribed to.

Implementing an Interface Event

As you develop UI components, you often start by creating a new form or a new control that inherits from an existing form or control. Events are frequently defined on an interface, and, in that case, you must implement the interface to implement the event. The `System.ComponentModel.INotifyPropertyChanged` interface defines a single `System.ComponentModel.INotifyPropertyChanged.PropertyChanged` event. The following code illustrates how to implement the event that this inherited interface defined:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

type AppForm() as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs -> this.Property1 <- "text2"
        this.Property2 <- "text3")

    // This property does not have the property-changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property-changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    // Expose the PropertyChanged event as a first class .NET event.
    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = propertyChanged.Publish.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = propertyChanged.Publish.RemoveHandler(handler)

    // This is the event-handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property %s changed its value to %s" args.PropertyName newValue

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
let inpc = appForm :> INotifyPropertyChanged
inpc.PropertyChanged.Add(appForm.OnPropertyChanged)
Application.Run(appForm)
```

If you want to hook up the event in the constructor, the code is a bit more complicated because the event hookup must be in a `then` block in an additional constructor, as in the following example:

```
module CustomForm

open System.Windows.Forms
open System.ComponentModel

// Create a private constructor with a dummy argument so that the public
// constructor can have no arguments.
type AppForm private (dummy) as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs -> this.Property1 <- "text2"
        this.Property2 <- "text3")

    // This property does not have the property changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = this.PropertyChanged.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = this.PropertyChanged.RemoveHandler(handler)

    // This is the event handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property %s changed its value to %s" args.PropertyName newValue

    new() as this =
        new AppForm(0)
        then
            let inpc = this :> INotifyPropertyChanged
            inpc.PropertyChanged.Add(this.OnPropertyChanged)

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
Application.Run(appForm)
```

See also

- Members
- Handling and Raising Events
- Lambda Expressions: The `fun` Keyword
- Control.Event Module
- Control.Event<'T> Class
- Control.Event<'Delegate,'Args> Class

Explicit Fields: The val Keyword

7/30/2019 • 4 minutes to read • Edit Online

The `val` keyword is used to declare a location to store a value in a class or structure type, without initializing it. Storage locations declared in this manner are called *explicit fields*. Another use of the `val` keyword is in conjunction with the `member` keyword to declare an auto-implemented property. For more information on auto-implemented properties, see Properties.

Syntax

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

Remarks

The usual way to define fields in a class or structure type is to use a `let` binding. However, `let` bindings must be initialized as part of the class constructor, which is not always possible, necessary, or desirable. You can use the `val` keyword when you want a field that is uninitialized.

Explicit fields can be static or non-static. The *access-modifier* can be `public`, `private`, or `internal`. By default, explicit fields are public. This differs from `let` bindings in classes, which are always private.

The DefaultValue attribute is required on explicit fields in class types that have a primary constructor. This attribute specifies that the field is initialized to zero. The type of the field must support zero-initialization. A type supports zero-initialization if it is one of the following:

- A primitive type that has a zero value.
- A type that supports a null value, either as a normal value, as an abnormal value, or as a representation of a value. This includes classes, tuples, records, functions, interfaces, .NET reference types, the `unit` type, and discriminated union types.
- A .NET value type.
- A structure whose fields all support a default zero value.

For example, an immutable field called `someField` has a backing field in the .NET compiled representation with the name `someField@`, and you access the stored value using a property named `someField`.

For a mutable field, the .NET compiled representation is a .NET field.

> **WARNING**
> The .NET Framework namespace `System.ComponentModel` contains an attribute that has the same name. For information about this attribute, see `System.ComponentModel.DefaultValueAttribute`.

The following code shows the use of explicit fields and, for comparison, a `let` binding in a class that has a primary constructor. Note that the `let`-bound field `myInt1` is private. When the `let`-bound field `myInt1` is referenced from a member method, the self identifier `this` is not required. But when you are referencing the explicit fields `myInt2` and `myString`, the self identifier is required.

```
type MyType() =
    let mutable myInt1 = 10
    [<DefaultValue>] val mutable myInt2 : int
    [<DefaultValue>] val mutable myString : string
    member this.SetValsAndPrint( i: int, str: string) =
       myInt1 <- i
       this.myInt2 <- i + 1
       this.myString <- str
       printfn "%d %d %s" myInt1 (this.myInt2) (this.myString)

let myObject = new MyType()
myObject.SetValsAndPrint(11, "abc")
// The following line is not allowed because let bindings are private.
// myObject.myInt1 <- 20
myObject.myInt2 <- 30
myObject.myString <- "def"

printfn "%d %s" (myObject.myInt2) (myObject.myString)
```

The output is as follows:

```
11 12 abc
30 def
```

The following code shows the use of explicit fields in a class that does not have a primary constructor. In this case, the `DefaultValue` attribute is not required, but all the fields must be initialized in the constructors that are defined for the type.

```
type MyClass =
    val a : int
    val b : int
    // The following version of the constructor is an error
    // because b is not initialized.
    // new (a0, b0) = { a = a0; }
    // The following version is acceptable because all fields are initialized.
    new(a0, b0) = { a = a0; b = b0; }

let myClassObj = new MyClass(35, 22)
printfn "%d %d" (myClassObj.a) (myClassObj.b)
```

The output is `35 22`.

The following code shows the use of explicit fields in a structure. Because a structure is a value type, it automatically has a default constructor that sets the values of its fields to zero. Therefore, the `DefaultValue` attribute is not required.

```
type MyStruct =
    struct
        val mutable myInt : int
        val mutable myString : string
    end

let mutable myStructObj = new MyStruct()
myStructObj.myInt <- 11
myStructObj.myString <- "xyz"

printfn "%d %s" (myStructObj.myInt) (myStructObj.myString)
```

The output is `11 xyz`.

Beware, if you are going to initialize your structure with `mutable` fields without `mutable` keyword, your assignments will work on a copy of the structure which will be discarded right after assignment. Therefore your structure won't change.

```fsharp
[<Struct>]
type Foo =
  val mutable bar: string
  member self.ChangeBar bar = self.bar <- bar
  new (bar) = {bar = bar}

let foo = Foo "1"
foo.ChangeBar "2" //make implicit copy of Foo, changes the copy, discards the copy, foo remains unchanged
printfn "%s" foo.bar //prints 1

let mutable foo' = Foo "1"
foo'.ChangeBar "2" //changes foo'
printfn "%s" foo'.bar //prints 2
```

Explicit fields are not intended for routine use. In general, when possible you should use a `let` binding in a class instead of an explicit field. Explicit fields are useful in certain interoperability scenarios, such as when you need to define a structure that will be used in a platform invoke call to a native API, or in COM interop scenarios. For more information, see External Functions. Another situation in which an explicit field might be necessary is when you are working with an F# code generator which emits classes without a primary constructor. Explicit fields are also useful for thread-static variables or similar constructs. For more information, see `System.ThreadStaticAttribute`.

When the keywords `member val` appear together in a type definition, it is a definition of an automatically implemented property. For more information, see Properties.

See also

- Properties
- Members
- `let` Bindings in Classes

Type extensions

7/30/2019 • 5 minutes to read • Edit Online

Type extensions (also called *augmentations*) are a family of features that let you add new members to a previously defined object type. The three features are:

- Intrinsic type extensions
- Optional type extensions
- Extension methods

Each can be used in different scenarios and has different tradeoffs.

Syntax

```
// Intrinsic and optional extensions
type typename with
    member self-identifier.member-name =
        body
    ...

// Extension methods
open System.Runtime.CompilerServices

[<Extension>]
type Extensions() =
    [static] member self-identifier.extension-name (ty: typename, [args]) =
        body
    ...
```

Intrinsic type extensions

An intrinsic type extension is a type extension that extends a user-defined type.

Intrinsic type extensions must be defined in the same file **and** in the same namespace or module as the type they're extending. Any other definition will result in them being optional type extensions.

Intrinsic type extensions are sometimes a cleaner way to separate functionality from the type declaration. The following example shows how to define an intrinsic type extension:

```
namespace Example

type Variant =
    | Num of int
    | Str of string

module Variant =
    let print v =
        match v with
        | Num n -> printf "Num %d" n
        | Str s -> printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
    member x.Print() = Variant.print x
```

Using a type extension allows you to separate each of the following:

- The declaration of a `Variant` type
- Functionality to print the `Variant` class depending on its "shape"
- A way to access the printing functionality with object-style `.`-notation

This is an alternative to defining everything as a member on `Variant`. Although it is not an inherently better approach, it can be a cleaner representation of functionality in some situations.

Intrinsic type extensions are compiled as members of the type they augment, and appear on the type when the type is examined by reflection.

Optional type extensions

An optional type extension is an extension that appears outside the original module, namespace, or assembly of the type being extended.

Optional type extensions are useful for extending a type that you have not defined yourself. For example:

```
module Extensions

open System.Collections.Generic

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for i in 1 .. n do
                    yield x
        }
```

You can now access `RepeatElements` as if it's a member of IEnumerable<T> as long as the `Extensions` module is opened in the scope that you are working in.

Optional extensions do not appear on the extended type when examined by reflection. Optional extensions must be in modules, and they're only in scope when the module that contains the extension is open or is otherwise in scope.

Optional extension members are compiled to static members for which the object instance is passed implicitly as the first parameter. However, they act as if they're instance members or static members according to how they're declared.

Optional extension members are also not visible to C# or VB consumers. They can only be consumed in other F# code.

Generic limitation of intrinsic and optional type extensions

It's possible to declare a type extension on a generic type where the type variable is constrained. The requirement is that the constraint of the extension declaration matches the constraint of the declared type.

However, even when constraints are matched between a declared type and a type extension, it's possible for a constraint to be inferred by the body of an extended member that imposes a different requirement on the type parameter than the declared type. For example:

```
open System.Collections.Generic

// NOT POSSIBLE AND FAILS TO COMPILE!
//
// The member 'Sum' has a different requirement on 'T than the type IEnumerable<'T>
type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
```

There is no way to get this code to work with an optional type extension:

- As is, the `Sum` member has a different constraint on `'T` (`static member get_Zero` and `static member (+)`) than what the type extension defines.
- Modifying the type extension to have the same constraint as `Sum` will no longer match the defined constraint on `IEnumerable<'T>`.
- Changing `member this.Sum` to `member inline this.Sum` will give an error that type constraints are mismatched.

What is desired are static methods that "float in space" and can be presented as if they're extending a type. This is where extension methods become necessary.

Extension methods

Finally, extension methods (sometimes called "C# style extension members") can be declared in F# as a static member method on a class.

Extension methods are useful for when you wish to define extensions on a generic type that will constrain the type variable. For example:

```
namespace Extensions

open System.Runtime.CompilerServices

[<Extension>]
type IEnumerableExtensions() =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs
```

When used, this code will make it appear as if `Sum` is defined on IEnumerable<T>, so long as `Extensions` has been opened or is in scope.

Other remarks

Type extensions also have the following attributes:

- Any type that can be accessed can be extended.
- Intrinsic and optional type extensions can define *any* member type, not just methods. So extension properties are also possible, for example.
- The `self-identifier` token in the syntax represents the instance of the type being invoked, just like ordinary members.
- Extended members can be static or instance members.
- Type variables on a type extension must match the constraints of the declared type.

The following limitations also exist for type extensions:

- Type extensions do not support virtual or abstract methods.
- Type extensions do not support override methods as augmentations.

- Type extensions do not support Statically Resolved Type Parameters.
- Optional Type extensions do not support constructors as augmentations.
- Type extensions cannot be defined on type abbreviations.
- Type extensions are not valid for `byref<'T>` (though they can be declared).
- Type extensions are not valid for attributes (though they can be declared).
- You can define extensions that overload other methods of the same name, but the F# compiler gives preference to non-extension methods if there is an ambiguous call.

Finally, if multiple intrinsic type extensions exist for one type, all members must be unique. For optional type extensions, members in different type extensions to the same type can have the same names. Ambiguity errors occur only if client code opens two different scopes that define the same member names.

See also

- F# Language Reference
- Members

Parameters and Arguments

7/30/2019 • 10 minutes to read • Edit Online

This topic describes language support for defining parameters and passing arguments to functions, methods, and properties. It includes information about how to pass by reference, and how to define and use methods that can take a variable number of arguments.

Parameters and Arguments

The term *parameter* is used to describe the names for values that are expected to be supplied. The term *argument* is used for the values provided for each parameter.

Parameters can be specified in tuple or curried form, or in some combination of the two. You can pass arguments by using an explicit parameter name. Parameters of methods can be specified as optional and given a default value.

Parameter Patterns

Parameters supplied to functions and methods are, in general, patterns separated by spaces. This means that, in principle, any of the patterns described in Match Expressions can be used in a parameter list for a function or member.

Methods usually use the tuple form of passing arguments. This achieves a clearer result from the perspective of other .NET languages because the tuple form matches the way arguments are passed in .NET methods.

The curried form is most often used with functions created by using `let` bindings.

The following pseudocode shows examples of tuple and curried arguments.

```
// Tuple form.
member this.SomeMethod(param1, param2) = ...
// Curried form.
let function1 param1 param2 = ...
```

Combined forms are possible when some arguments are in tuples and some are not.

```
let function2 param1 (param2a, param2b) param3 = ...
```

Other patterns can also be used in parameter lists, but if the parameter pattern does not match all possible inputs, there might be an incomplete match at run time. The exception `MatchFailureException` is generated when the value of an argument does not match the patterns specified in the parameter list. The compiler issues a warning when a parameter pattern allows for incomplete matches. At least one other pattern is commonly useful for parameter lists, and that is the wildcard pattern. You use the wildcard pattern in a parameter list when you simply want to ignore any arguments that are supplied. The following code illustrates the use of the wildcard pattern in an argument list.

```
let makeList _ = [ for i in 1 .. 100 -> i * i ]
// The arguments 100 and 200 are ignored.
let list1 = makeList 100
let list2 = makeList 200
```

The wildcard pattern can be useful whenever you do not need the arguments passed in, such as in the main entry point to a program, when you are not interested in the command-line arguments that are normally supplied as a string array, as in the following code.

```
[<EntryPoint>]
let main _ =
    printfn "Entry point!"
    0
```

Other patterns that are sometimes used in arguments are the `as` pattern, and identifier patterns associated with discriminated unions and active patterns. You can use the single-case discriminated union pattern as follows.

```
type Slice = Slice of int * int * string

let GetSubstring1 (Slice(p0, p1, text)) =
    printfn "Data begins at %d and ends at %d in string %s" p0 p1 text
    text.[p0..p1]

let substring = GetSubstring1 (Slice(0, 4, "Et tu, Brute?"))
printfn "Substring: %s" substring
```

The output is as follows.

```
Data begins at 0 and ends at 4 in string Et tu, Brute?
Et tu
```

Active patterns can be useful as parameters, for example, when transforming an argument into a desired format, as in the following example:

```
type Point = { x : float; y : float }

let (| Polar |) { x = x; y = y} =
    ( sqrt (x*x + y*y), System.Math.Atan (y/ x) )

let radius (Polar(r, _)) = r
let angle (Polar(_, theta)) = theta
```

You can use the `as` pattern to store a matched value as a local value, as is shown in the following line of code.

```
let GetSubstring2 (Slice(p0, p1, text) as s) = s
```

Another pattern that is used occasionally is a function that leaves the last argument unnamed by providing, as the body of the function, a lambda expression that immediately performs a pattern match on the implicit argument. An example of this is the following line of code.

```
let isNil = function [] -> true | _::_ -> false
```

This code defines a function that takes a generic list and returns `true` if the list is empty, and `false` otherwise. The use of such techniques can make code more difficult to read.

Occasionally, patterns that involve incomplete matches are useful, for example, if you know that the lists in your program have only three elements, you might use a pattern like the following in a parameter list.

```
let sum [a; b; c;] = a + b + c
```

The use of patterns that have incomplete matches is best reserved for quick prototyping and other temporary uses. The compiler will issue a warning for such code. Such patterns cannot cover the general case of all possible inputs and therefore are not suitable for component APIs.

Named Arguments

Arguments for methods can be specified by position in a comma-separated argument list, or they can be passed to a method explicitly by providing the name, followed by an equal sign and the value to be passed in. If specified by providing the name, they can appear in a different order from that used in the declaration.

Named arguments can make code more readable and more adaptable to certain types of changes in the API, such as a reordering of method parameters.

Named arguments are allowed only for methods, not for `let`-bound functions, function values, or lambda expressions.

The following code example demonstrates the use of named arguments.

```
type SpeedingTicket() =
    member this.GetMPHOver(speed: int, limit: int) = speed - limit

let CalculateFine (ticket : SpeedingTicket) =
    let delta = ticket.GetMPHOver(limit = 55, speed = 70)
    if delta < 20 then 50.0 else 100.0

let ticket1 : SpeedingTicket = SpeedingTicket()
printfn "%f" (CalculateFine ticket1)
```

In a call to a class constructor, you can set the values of properties of the class by using a syntax similar to that of named arguments. The following example shows this syntax.

```
 type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
       with get() = number
       and set(value) = number <- value
    member this.FirstName
       with get() = firstName
       and set(value) = firstName <- value
    member this.LastName
       with get() = lastName
       and set(value) = lastName <- value
    member this.Balance
       with get() = balance
       and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount


let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)
```

For more information, see Constructors (F#).

Optional Parameters

You can specify an optional parameter for a method by using a question mark in front of the parameter name. Optional parameters are interpreted as the F# option type, so you can query them in the regular way that option types are queried, by using a `match` expression with `Some` and `None`. Optional parameters are permitted only on members, not on functions created by using `let` bindings.

You can pass existing optional values to method by parameter name, such as `?arg=None` or `?arg=Some(3)` or `?arg=arg`. This can be useful when building a method that passes optional arguments to another method.

You can also use a function `defaultArg`, which sets a default value of an optional argument. The `defaultArg` function takes the optional parameter as the first argument and the default value as the second.

The following example illustrates the use of optional parameters.

```
type DuplexType =
    | Full
    | Half

type Connection(?rate0 : int, ?duplex0 : DuplexType, ?parity0 : bool) =
    let duplex = defaultArg duplex0 Full
    let parity = defaultArg parity0 false
    let mutable rate = match rate0 with
                       | Some rate1 -> rate1
                       | None -> match duplex with
                                 | Full -> 9600
                                 | Half -> 4800
    do printfn "Baud Rate: %d Duplex: %A Parity: %b" rate duplex parity

let conn1 = Connection(duplex0 = Full)
let conn2 = Connection(duplex0 = Half)
let conn3 = Connection(300, Half, true)
let conn4 = Connection(?duplex0 = None)
let conn5 = Connection(?duplex0 = Some(Full))

let optionalDuplexValue : option<DuplexType> = Some(Half)
let conn6 = Connection(?duplex0 = optionalDuplexValue)
```

The output is as follows.

```
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
Baud Rate: 300 Duplex: Half Parity: true
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
```

For the purposes of C# and Visual Basic interop you can use the attributes `[<Optional; DefaultParameterValue<(...)>]` in F#, so that callers will see an argument as optional. This is equivalent to defining the argument as optional in C# as in `MyMethod(int i = 3)`.

```
open System
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue("Hello world")>] message) =
        printfn "%s" message
```

You can also specify a new object as a default parameter value. For example, the `Foo` member could have an optional `CancellationToken` as input instead:

```
open System.Threading
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue(CancellationToken())>] ct: CancellationToken) =
        printfn "%A" ct
```

The value given as argument to `DefaultParameterValue` must match the type of the parameter. For example, the following is not allowed:

```
type C =
    static member Wrong([<Optional; DefaultParameterValue("string")>] i:int) = ()
```

In this case, the compiler generates a warning and will ignore both attributes altogether. Note that the default value `null` needs to be type-annotated, as otherwise the compiler infers the wrong type, i.e. `[<Optional; DefaultParameterValue(null:obj)>] o:obj`.

Passing by Reference

Passing an F# value by reference involves byrefs, which are managed pointer types. Guidance for which type to use is as follows:

- Use `inref<'T>` if you only need to read the pointer.
- Use `outref<'T>` if you only need to write to the pointer.
- Use `byref<'T>` if you need to both read from and write to the pointer.

```
let example1 (x: inref<int>) = printfn "It's %d" x

let example2 (x: outref<int>) = x <- x + 1

let example3 (x: byref<int>) =
    printfn "It'd %d" x
    x <- x + 1

// No need to make it mutable, since it's read-only
let x = 1
example1 &x

// Needs to be mutable, since we write to it
let mutable y = 2
example2 &y
example3 &y // Now 'y' is 3
```

Because the parameter is a pointer and the value is mutable, any changes to the value are retained after the execution of the function.

You can use a tuple as a return value to store any `out` parameters in .NET library methods. Alternatively, you can treat the `out` parameter as a `byref` parameter. The following code example illustrates both ways.

```
// TryParse has a second parameter that is an out parameter
// of type System.DateTime.
let (b, dt) = System.DateTime.TryParse("12-20-04 12:21:00")

printfn "%b %A" b dt

// The same call, using an address of operator.
let mutable dt2 = System.DateTime.Now
let b2 = System.DateTime.TryParse("12-20-04 12:21:00", &dt2)

printfn "%b %A" b2 dt2
```

Parameter Arrays

Occasionally it is necessary to define a function that takes an arbitrary number of parameters of heterogeneous type. It would not be practical to create all the possible overloaded methods to account for all the types that could be used. The .NET implementations provide support for such methods through the parameter array feature. A method that takes a parameter array in its signature can be provided with an arbitrary number of parameters. The parameters are put into an array. The type of the array elements determines the parameter types that can be passed to the function. If you define the parameter array with `System.Object` as the element type, then client code can pass values of any type.

In F#, parameter arrays can only be defined in methods. They cannot be used in standalone functions or functions that are defined in modules.

You define a parameter array by using the `ParamArray` attribute. The `ParamArray` attribute can only be applied to the last parameter.

The following code illustrates both calling a .NET method that takes a parameter array and the definition of a type in F# that has a method that takes a parameter array.

```
open System

type X() =
    member this.F([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

[<EntryPoint>]
let main _ =
    // call a .NET method that takes a parameter array, passing values of various types
    Console.WriteLine("a {0} {1} {2} {3} {4}", 1, 10.0, "Hello world", 1u, true)

    let xobj = new X()
    // call an F# method that takes a parameter array, passing values of various types
    xobj.F("a", 1, 10.0, "Hello world", 1u, true)
    0
```

When run in a project, the output of the previous code is as follows:

```
a 1 10 Hello world 1 True
"a"
1
10.0
"Hello world"
1u
true
```

See also

- Members

Operator Overloading

7/30/2019 • 7 minutes to read • Edit Online

This topic describes how to overload arithmetic operators in a class or record type, and at the global level.

Syntax

```
// Overloading an operator as a class or record member.
static member (operator-symbols) (parameter-list) =
    method-body
// Overloading an operator at the global level
let [inline] (operator-symbols) parameter-list = function-body
```

Remarks

In the previous syntax, the *operator-symbol* is one of `+`, `-`, `*`, `/`, `=`, and so on. The *parameter-list* specifies the operands in the order they appear in the usual syntax for that operator. The *method-body* constructs the resulting value.

Operator overloads for operators must be static. Operator overloads for unary operators, such as `+` and `-`, must use a tilde (`~`) in the *operator-symbol* to indicate that the operator is a unary operator and not a binary operator, as shown in the following declaration.

```
static member (~-) (v : Vector)
```

The following code illustrates a vector class that has just two operators, one for unary minus and one for multiplication by a scalar. In the example, two overloads for scalar multiplication are needed because the operator must work regardless of the order in which the vector and scalar appear.

```
type Vector(x: float, y : float) =
   member this.x = x
   member this.y = y
   static member (~-) (v : Vector) =
     Vector(-1.0 * v.x, -1.0 * v.y)
   static member (*) (v : Vector, a) =
     Vector(a * v.x, a * v.y)
   static member (*) (a, v: Vector) =
     Vector(a * v.x, a * v.y)
   override this.ToString() =
     this.x.ToString() + " " + this.y.ToString()

let v1 = Vector(1.0, 2.0)

let v2 = v1 * 2.0
let v3 = 2.0 * v1

let v4 = - v2

printfn "%s" (v1.ToString())
printfn "%s" (v2.ToString())
printfn "%s" (v3.ToString())
printfn "%s" (v4.ToString())
```

Creating New Operators

You can overload all the standard operators, but you can also create new operators out of sequences of certain characters. Allowed operator characters are `!`, `%`, `&`, `*`, `+`, `-`, `.`, `/`, `<`, `=`, `>`, `?`, `@`, `^`, `|`, and `~`. The `~` character has the special meaning of making an operator unary, and is not part of the operator character sequence. Not all operators can be made unary.

Depending on the exact character sequence you use, your operator will have a certain precedence and associativity. Associativity can be either left to right or right to left and is used whenever operators of the same level of precedence appear in sequence without parentheses.

The operator character `.` does not affect precedence, so that, for example, if you want to define your own version of multiplication that has the same precedence and associativity as ordinary multiplication, you could create operators such as `.*`.

Only the operators `?` and `?<-` may start with `?`.

A table that shows the precedence of all operators in F# can be found in Symbol and Operator Reference.

Overloaded Operator Names

When the F# compiler compiles an operator expression, it generates a method that has a compiler-generated name for that operator. This is the name that appears in the Microsoft intermediate language (MSIL) for the method, and also in reflection and IntelliSense. You do not normally need to use these names in F# code.

The following table shows the standard operators and their corresponding generated names.

OPERATOR	GENERATED NAME
`[]`	`op_Nil`
`::`	`op_Cons`
`+`	`op_Addition`
`-`	`op_Subtraction`
`*`	`op_Multiply`
`/`	`op_Division`
`@`	`op_Append`
`^`	`op_Concatenate`
`%`	`op_Modulus`
`&&&`	`op_BitwiseAnd`
`|||`	`op_BitwiseOr`
`^^^`	`op_ExclusiveOr`

OPERATOR	GENERATED NAME
`<<<`	`op_LeftShift`
`~~~`	`op_LogicalNot`
`>>>`	`op_RightShift`
`~+`	`op_UnaryPlus`
`~-`	`op_UnaryNegation`
`=`	`op_Equality`
`<=`	`op_LessThanOrEqual`
`>=`	`op_GreaterThanOrEqual`
`<`	`op_LessThan`
`>`	`op_GreaterThan`
`?`	`op_Dynamic`
`?<-`	`op_DynamicAssignment`
`|>`	`op_PipeRight`
`<|`	`op_PipeLeft`
`!`	`op_Dereference`
`>>`	`op_ComposeRight`
`<<`	`op_ComposeLeft`
`<@ @>`	`op_Quotation`
`<@@ @@>`	`op_QuotationUntyped`
`+=`	`op_AdditionAssignment`
`-=`	`op_SubtractionAssignment`
`*=`	`op_MultiplyAssignment`
`/=`	`op_DivisionAssignment`
`..`	`op_Range`

OPERATOR	GENERATED NAME
`.. ..`	`op_RangeStep`

Other combinations of operator characters that are not listed here can be used as operators and have names that are made up by concatenating names for the individual characters from the following table. For example, +! becomes `op_PlusBang`.

OPERATOR CHARACTER	NAME
`>`	`Greater`
`<`	`Less`
`+`	`Plus`
`-`	`Minus`
`*`	`Multiply`
`/`	`Divide`
`=`	`Equals`
`~`	`Twiddle`
`%`	`Percent`
`.`	`Dot`
`&`	`Amp`
`|`	`Bar`
`@`	`At`
`^`	`Hat`
`!`	`Bang`
`?`	`Qmark`
`(`	`LParen`
`,`	`Comma`
`)`	`RParen`
`[`	`LBrack`

OPERATOR CHARACTER	NAME
`]`	`RBrack`

Prefix and Infix Operators

Prefix operators are expected to be placed in front of an operand or operands, much like a function. *Infix* operators are expected to be placed between the two operands.

Only certain operators can be used as prefix operators. Some operators are always prefix operators, others can be infix or prefix, and the rest are always infix operators. Operators that begin with `!`, except `!=`, and the operator `~`, or repeated sequences of `~`, are always prefix operators. The operators `+`, `-`, `+.`, `-.`, `&`, `&&`, `%`, and `%%` can be prefix operators or infix operators. You distinguish the prefix version of these operators from the infix version by adding a `~` at the beginning of a prefix operator when it is defined. The `~` is not used when you use the operator, only when it is defined.

Example

The following code illustrates the use of operator overloading to implement a fraction type. A fraction is represented by a numerator and a denominator. The function `hcf` is used to determine the highest common factor, which is used to reduce fractions.

```
// Determine the highest common factor between
// two positive integers, a helper for reducing
// fractions.
let rec hcf a b =
  if a = 0u then b
  elif a<b then hcf a (b - a)
  else hcf (a - b) b

// type Fraction: represents a positive fraction
// (positive rational number).
type Fraction =
   {
      // n: Numerator of fraction.
      n : uint32
      // d: Denominator of fraction.
      d : uint32
   }

   // Produce a string representation. If the
   // denominator is "1", do not display it.
   override this.ToString() =
      if (this.d = 1u)
        then this.n.ToString()
        else this.n.ToString() + "/" + this.d.ToString()

   // Add two fractions.
   static member (+) (f1 : Fraction, f2 : Fraction) =
      let nTemp = f1.n * f2.d + f2.n * f1.d
      let dTemp = f1.d * f2.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

   // Adds a fraction and a positive integer.
   static member (+) (f1: Fraction, i : uint32) =
      let nTemp = f1.n + i * f1.d
      let dTemp = f1.d
      let hcfTemp = hcf nTemp dTemp
      { n = nTemp / hcfTemp; d = dTemp / hcfTemp }
```

```
    // Adds a positive integer and a fraction.
    static member (+) (i : uint32, f2: Fraction) =
       let nTemp = f2.n + i * f2.d
       let dTemp = f2.d
       let hcfTemp = hcf nTemp dTemp
       { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

    // Subtract one fraction from another.
    static member (-) (f1 : Fraction, f2 : Fraction) =
       if (f2.n * f1.d > f1.n * f2.d)
         then failwith "This operation results in a negative number, which is not supported."
       let nTemp = f1.n * f2.d - f2.n * f1.d
       let dTemp = f1.d * f2.d
       let hcfTemp = hcf nTemp dTemp
       { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

    // Multiply two fractions.
    static member (*) (f1 : Fraction, f2 : Fraction) =
       let nTemp = f1.n * f2.n
       let dTemp = f1.d * f2.d
       let hcfTemp = hcf nTemp dTemp
       { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

    // Divide two fractions.
    static member (/) (f1 : Fraction, f2 : Fraction) =
       let nTemp = f1.n * f2.d
       let dTemp = f2.n * f1.d
       let hcfTemp = hcf nTemp dTemp
       { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

    // A full set of operators can be quite lengthy. For example,
    // consider operators that support other integral data types,
    // with fractions, on the left side and the right side for each.
    // Also consider implementing unary operators.

let fraction1 = { n = 3u; d = 4u }
let fraction2 = { n = 1u; d = 2u }
let result1 = fraction1 + fraction2
let result2 = fraction1 - fraction2
let result3 = fraction1 * fraction2
let result4 = fraction1 / fraction2
let result5 = fraction1 + 1u
printfn "%s + %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result1.ToString())
printfn "%s - %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result2.ToString())
printfn "%s * %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result3.ToString())
printfn "%s / %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result4.ToString())
printfn "%s + 1 = %s" (fraction1.ToString()) (result5.ToString())
```

Output:

```
3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2
3/4 + 1 = 7/4
```

Operators at the Global Level

You can also define operators at the global level. The following code defines an operator `+?`.

```
let inline (+?) (x: int) (y: int) = x + 2*y
printf "%d" (10 +? 1)
```

The output of the above code is `12`.

You can redefine the regular arithmetic operators in this manner because the scoping rules for F# dictate that newly defined operators take precedence over the built-in operators.

The keyword `inline` is often used with global operators, which are often small functions that are best integrated into the calling code. Making operator functions inline also enables them to work with statically resolved type parameters to produce statically resolved generic code. For more information, see Inline Functions and Statically Resolved Type Parameters.

See also

- Members

Flexible Types

7/30/2019 • 2 minutes to read • Edit Online

A *flexible type annotation* indicates that a parameter, variable, or value has a type that is compatible with a specified type, where compatibility is determined by position in an object-oriented hierarchy of classes or interfaces. Flexible types are useful specifically when the automatic conversion to types higher in the type hierarchy does not occur but you still want to enable your functionality to work with any type in the hierarchy or any type that implements an interface.

Syntax

```
#type
```

Remarks

In the previous syntax, *type* represents a base type or an interface.

A flexible type is equivalent to a generic type that has a constraint that limits the allowed types to types that are compatible with the base or interface type. That is, the following two lines of code are equivalent.

```
#SomeType

'T when 'T :> SomeType
```

Flexible types are useful in several types of situations. For example, when you have a higher order function (a function that takes a function as an argument), it is often useful to have the function return a flexible type. In the following example, the use of a flexible type with a sequence argument in `iterate2` enables the higher order function to work with functions that generate sequences, arrays, lists, and any other enumerable type.

Consider the following two functions, one of which returns a sequence, the other of which returns a flexible type.

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

As another example, consider the Seq.concat library function:

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

You can pass any of the following enumerable sequences to this function:

- A list of lists

- A list of arrays
- An array of lists
- An array of sequences
- Any other combination of enumerable sequences

The following code uses `Seq.concat` to demonstrate the scenarios that you can support by using flexible types.

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

The output is as follows.

```
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
```

In F#, as in other object-oriented languages, there are contexts in which derived types or types that implement interfaces are automatically converted to a base type or interface type. These automatic conversions occur in direct arguments, but not when the type is in a subordinate position, as part of a more complex type such as a return type of a function type, or as a type argument. Thus, the flexible type notation is primarily useful when the type you are applying it to is part of a more complex type.

See also

- F# Language Reference
- Generics

Delegates

7/30/2019 • 3 minutes to read • Edit Online

A delegate represents a function call as an object. In F#, you ordinarily should use function values to represent functions as first-class values; however, delegates are used in the .NET Framework and so are needed when you interoperate with APIs that expect them. They may also be used when authoring libraries designed for use from other .NET Framework languages.

Syntax

```
type delegate-typename = delegate of type1 -> type2
```

Remarks

In the previous syntax, `type1` represents the argument type or types and `type2` represents the return type. The argument types that are represented by `type1` are automatically curried. This suggests that for this type you use a tuple form if the arguments of the target function are curried, and a parenthesized tuple for arguments that are already in the tuple form. The automatic currying removes a set of parentheses, leaving a tuple argument that matches the target method. Refer to the code example for the syntax you should use in each case.

Delegates can be attached to F# function values, and static or instance methods. F# function values can be passed directly as arguments to delegate constructors. For a static method, you construct the delegate by using the name of the class and the method. For an instance method, you provide the object instance and method in one argument. In both cases, the member access operator (`.`) is used.

The `Invoke` method on the delegate type calls the encapsulated function. Also, delegates can be passed as function values by referencing the Invoke method name without the parentheses.

The following code shows the syntax for creating delegates that represent various methods in a class. Depending on whether the method is a static method or an instance method, and whether it has arguments in the tuple form or the curried form, the syntax for declaring and assigning the delegate is a little different.

```
type Test1() =
  static member add(a : int, b : int) =
     a + b
  static member add2 (a : int) (b : int) =
     a + b

  member x.Add(a : int, b : int) =
     a + b
  member x.Add2 (a : int) (b : int) =
     a + b


// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg : Delegate1) (a : int) (b: int) =
   dlg.Invoke(a, b)
let InvokeDelegate2 (dlg : Delegate2) (a : int) (b: int) =
   dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 : Delegate1 = new Delegate1( Test1.add )
let del2 : Delegate2 = new Delegate2( Test1.add2 )

let testObject = Test1()

// For instance methods, use the instance value name, the dot operator, and the instance method name.
let del3 : Delegate1 = new Delegate1( testObject.Add )
let del4 : Delegate2 = new Delegate2( testObject.Add2 )

for (a, b) in [ (100, 200); (10, 20) ] do
  printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del2 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
  printfn "%d + %d = %d" a b (InvokeDelegate2 del4 a b)
```

The following code shows some of the different ways you can work with delegates.

```
type Delegate1 = delegate of int * char -> string

let replicate n c = String.replicate n (string c)

// An F# function value constructed from an unapplied let-bound function
let function1 = replicate

// A delegate object constructed from an F# function value
let delObject = new Delegate1(function1)

// An F# function value constructed from an unapplied .NET member
let functionValue = delObject.Invoke

List.map (fun c -> functionValue(5,c)) ['a'; 'b'; 'c']
|> List.iter (printfn "%s")

// Or if you want to get back the same curried signature
let replicate' n c =  delObject.Invoke(n,c)

// You can pass a lambda expression as an argument to a function expecting a compatible delegate type
// System.Array.ConvertAll takes an array and a converter delegate that transforms an element from
// one type to another according to a specified function.
let stringArray = System.Array.ConvertAll([|'a';'b'|], fun c -> replicate' 3 c)
printfn "%A" stringArray
```

The output of the previous code example is as follows.

```
aaaaa
bbbbb
ccccc
[|"aaa"; "bbb"|]
```

See also

- F# Language Reference
- Parameters and Arguments
- Events

Object Expressions

4/8/2019 • 2 minutes to read • Edit Online

An *object expression* is an expression that creates a new instance of a dynamically created, anonymous object type that is based on an existing base type, interface, or set of interfaces.

Syntax

```
// When typename is a class:
{ new typename [type-params]arguments with
    member-definitions
    [ additional-interface-definitions ]
}
// When typename is not a class:
{ new typename [generic-type-args] with
    member-definitions
    [ additional-interface-definitions ]
}
```

Remarks

In the previous syntax, the *typename* represents an existing class type or interface type. *type-params* describes the optional generic type parameters. The *arguments* are used only for class types, which require constructor parameters. The *member-definitions* are overrides of base class methods, or implementations of abstract methods from either a base class or an interface.

The following example illustrates several different types of object expressions.

```
// This object expression specifies a System.Object but overrides the
// ToString method.
let obj1 = { new System.Object() with member x.ToString() = "F#" }
printfn "%A" obj1

// This object expression implements the IFormattable interface.
let delimiter(delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member x.ToString(format: string, provider: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value }

let obj2 = delimiter("{","}", "Bananas!");

printfn "%A" (System.String.Format("{0:D}", obj2))

// Define two interfaces
type IFirst =
  abstract F : unit -> unit
  abstract G : unit -> unit

type ISecond =
  inherit IFirst
  abstract H : unit -> unit
  abstract J : unit -> unit

// This object expression implements both interfaces.
let implementer() =
    { new ISecond with
        member this.H() = ()
        member this.J() = ()
      interface IFirst with
        member this.F() = ()
        member this.G() = () }
```

Using Object Expressions

You use object expressions when you want to avoid the extra code and overhead that is required to create a new, named type. If you use object expressions to minimize the number of types created in a program, you can reduce the number of lines of code and prevent the unnecessary proliferation of types. Instead of creating many types just to handle specific situations, you can use an object expression that customizes an existing type or provides an appropriate implementation of an interface for the specific case at hand.

See also

- F# Language Reference

Casting and Conversions (F#)

7/30/2019 • 6 minutes to read • Edit Online

This topic describes support for type conversions in F#.

Arithmetic Types

F# provides conversion operators for arithmetic conversions between various primitive types, such as between integer and floating point types. The integral and char conversion operators have checked and unchecked forms; the floating point operators and the `enum` conversion operator do not. The unchecked forms are defined in `Microsoft.FSharp.Core.Operators` and the checked forms are defined in `Microsoft.FSharp.Core.Operators.Checked`. The checked forms check for overflow and generate a runtime exception if the resulting value exceeds the limits of the target type.

Each of these operators has the same name as the name of the destination type. For example, in the following code, in which the types are explicitly annotated, `byte` appears with two different meanings. The first occurrence is the type and the second is the conversion operator.

```
let x : int = 5

let b : byte = byte x
```

The following table shows conversion operators defined in F#.

OPERATOR	DESCRIPTION
`byte`	Convert to byte, an 8-bit unsigned type.
`sbyte`	Convert to signed byte.
`int16`	Convert to a 16-bit signed integer.
`uint16`	Convert to a 16-bit unsigned integer.
`int32, int`	Convert to a 32-bit signed integer.
`uint32`	Convert to a 32-bit unsigned integer.
`int64`	Convert to a 64-bit signed integer.
`uint64`	Convert to a 64-bit unsigned integer.
`nativeint`	Convert to a native integer.
`unativeint`	Convert to an unsigned native integer.
`float, double`	Convert to a 64-bit double-precision IEEE floating point number.

OPERATOR	DESCRIPTION
`float32, single`	Convert to a 32-bit single-precision IEEE floating point number.
`decimal`	Convert to `System.Decimal`.
`char`	Convert to `System.Char`, a Unicode character.
`enum`	Convert to an enumerated type.

In addition to built-in primitive types, you can use these operators with types that implement `op_Explicit` or `op_Implicit` methods with appropriate signatures. For example, the `int` conversion operator works with any type that provides a static method `op_Explicit` that takes the type as a parameter and returns `int`. As a special exception to the general rule that methods cannot be overloaded by return type, you can do this for `op_Explicit` and `op_Implicit`.

Enumerated Types

The `enum` operator is a generic operator that takes one type parameter that represents the type of the `enum` to convert to. When it converts to an enumerated type, type inference attempts to determine the type of the `enum` that you want to convert to. In the following example, the variable `col1` is not explicitly annotated, but its type is inferred from the later equality test. Therefore, the compiler can deduce that you are converting to a `Color` enumeration. Alternatively, you can supply a type annotation, as with `col2` in the following example.

```
type Color =
    | Red = 1
    | Green = 2
    | Blue = 3

// The target type of the conversion is determined by type inference.
let col1 = enum 1
// The target type is supplied by a type annotation.
let col2 : Color = enum 2
do
    if (col1 = Color.Red) then
       printfn "Red"
```

You can also specify the target enumeration type explicitly as a type parameter, as in the following code:

```
let col3 = enum<Color> 3
```

Note that the enumeration casts work only if the underlying type of the enumeration is compatible with the type being converted. In the following code, the conversion fails to compile because of the mismatch between `int32` and `uint32`.

```
// Error: types are incompatible
let col4 : Color = enum 2u
```

For more information, see Enumerations.

Casting Object Types

Conversion between types in an object hierarchy is fundamental to object-oriented programming. There are two basic types of conversions: casting up (upcasting) and casting down (downcasting). Casting up a hierarchy means casting from a derived object reference to a base object reference. Such a cast is guaranteed to work as long as the base class is in the inheritance hierarchy of the derived class. Casting down a hierarchy, from a base object reference to a derived object reference, succeeds only if the object actually is an instance of the correct destination (derived) type or a type derived from the destination type.

F# provides operators for these types of conversions. The `:>` operator casts up the hierarchy, and the `:?>` operator casts down the hierarchy.

Upcasting

In many object-oriented languages, upcasting is implicit; in F#, the rules are slightly different. Upcasting is applied automatically when you pass arguments to methods on an object type. However, for let-bound functions in a module, upcasting is not automatic, unless the parameter type is declared as a flexible type. For more information, see Flexible Types.

The `:>` operator performs a static cast, which means that the success of the cast is determined at compile time. If a cast that uses `:>` compiles successfully, it is a valid cast and has no chance of failure at run time.

You can also use the `upcast` operator to perform such a conversion. The following expression specifies a conversion up the hierarchy:

```
upcast expression
```

When you use the upcast operator, the compiler attempts to infer the type you are converting to from the context. If the compiler is unable to determine the target type, the compiler reports an error.

Downcasting

The `:?>` operator performs a dynamic cast, which means that the success of the cast is determined at run time. A cast that uses the `:?>` operator is not checked at compile time; but at run time, an attempt is made to cast to the specified type. If the object is compatible with the target type, the cast succeeds. If the object is not compatible with the target type, the runtime raises an `InvalidCastException`.

You can also use the `downcast` operator to perform a dynamic type conversion. The following expression specifies a conversion down the hierarchy to a type that is inferred from program context:

```
downcast expression
```

As for the `upcast` operator, if the compiler cannot infer a specific target type from the context, it reports an error.

The following code illustrates the use of the `:>` and `:?>` operators. The code illustrates that the `:?>` operator is best used when you know that conversion will succeed, because it throws `InvalidCastException` if the conversion fails. If you do not know that a conversion will succeed, a type test that uses a `match` expression is better because it avoids the overhead of generating an exception.

```
type Base1() =
    abstract member F : unit -> unit
    default u.F() =
     printfn "F Base1"

type Derived1() =
    inherit Base1()
    override u.F() =
      printfn "F Derived1"


let d1 : Derived1 = Derived1()

// Upcast to Base1.
let base1 = d1 :> Base1

// This might throw an exception, unless
// you are sure that base1 is really a Derived1 object, as
// is the case here.
let derived1 = base1 :?> Derived1

// If you cannot be sure that b1 is a Derived1 object,
// use a type test, as follows:
let downcastBase1 (b1 : Base1) =
   match b1 with
   | :? Derived1 as derived1 -> derived1.F()
   | _ -> ()

downcastBase1 base1
```

Because generic operators `downcast` and `upcast` rely on type inference to determine the argument and return type, in the above code, you can replace

```
let base1 = d1 :> Base1
```

with

```
let base1 = upcast d1
```

In the previous code, the argument type and return types are `Derived1` and `Base1`, respectively.

For more information about type tests, see Match Expressions.

See also

- F# Language Reference

Access Control

8/6/2019 • 3 minutes to read • Edit Online

Access control refers to declaring which clients can use certain program elements, such as types, methods, and functions.

Basics of Access Control

In F#, the access control specifiers `public`, `internal`, and `private` can be applied to modules, types, methods, value definitions, functions, properties, and explicit fields.

- `public` indicates that the entity can be accessed by all callers.
- `internal` indicates that the entity can be accessed only from the same assembly.
- `private` indicates that the entity can be accessed only from the enclosing type or module.

> **NOTE**
>
> The access specifier `protected` is not used in F#, although it is acceptable if you are using types authored in languages that do support `protected` access. Therefore, if you override a protected method, your method remains accessible only within the class and its descendents.

In general, the specifier is put in front of the name of the entity, except when a `mutable` or `inline` specifier is used, which appear after the access control specifier.

If no access specifier is used, the default is `public`, except for `let` bindings in a type, which are always `private` to the type.

Signatures in F# provide another mechanism for controlling access to F# program elements. Signatures are not required for access control. For more information, see Signatures.

Rules for Access Control

Access control is subject to the following rules:

- Inheritance declarations (that is, the use of `inherit` to specify a base class for a class), interface declarations (that is, specifying that a class implements an interface), and abstract members always have the same accessibility as the enclosing type. Therefore, an access control specifier cannot be used on these constructs.
- Accessibility for individual cases in a discriminated union is determined by the accessibility of the discriminated union itself. That is, a particular union case is no less accessible than the union itself.
- Accessibility for individual fields of a record type is determined by the accessibility of the record itself. That is, a particular record label is no less accessible than the record itself.

Example

The following code illustrates the use of access control specifiers. There are two files in the project, `Module1.fs` and `Module2.fs`. Each file is implicitly a module. Therefore, there are two modules, `Module1` and `Module2`. A private type and an internal type are defined in `Module1`. The private type cannot be accessed from `Module2`,

but the internal type can.

```
// Module1.fs

module Module1

// This type is not usable outside of this file
type private MyPrivateType() =
   // x is private since this is an internal let binding
   let x = 5
   // X is private and does not appear in the QuickInfo window
   // when viewing this type in the Visual Studio editor
   member private this.X() = 10
   member this.Z() = x * 100

type internal MyInternalType() =
   let x = 5
   member private this.X() = 10
   member this.Z() = x * 100

// Top-level let bindings are public by default,
// so "private" and "internal" are needed here since a
// value cannot be more accessible than its type.
let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

// let bindings at the top level are public by default,
// so result1 and result2 are public.
let result1 = myPrivateObj.Z
let result2 = myInternalObj.Z
```

The following code tests the accessibility of the types created in `Module1.fs`.

```
// Module2.fs
module Module2

open Module1

// The following line is an error because private means
// that it cannot be accessed from another file or module
// let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

let result = myInternalObj.Z
```

See also

- F# Language Reference
- Signatures

Conditional Expressions: `if...then...else`

7/30/2019 • 2 minutes to read • Edit Online

The `if...then...else` expression runs different branches of code and also evaluates to a different value depending on the Boolean expression given.

Syntax

```
if boolean-expression then expression1 [ else expression2 ]
```

Remarks

In the previous syntax, *expression1* runs when the Boolean expression evaluates to `true`; otherwise, *expression2* runs.

Unlike in other languages, the `if...then...else` construct is an expression, not a statement. That means that it produces a value, which is the value of the last expression in the branch that executes. The types of the values produced in each branch must match. If there is no explicit `else` branch, its type is `unit`. Therefore, if the type of the `then` branch is any type other than `unit`, there must be an `else` branch with the same return type. When chaining `if...then...else` expressions together, you can use the keyword `elif` instead of `else if`; they are equivalent.

Example

The following example illustrates how to use the `if...then...else` expression.

```
let test x y =
  if x = y then "equals"
  elif x < y then "is less than"
  else "is greater than"

printfn "%d %s %d." 10 (test 10 20) 20

printfn "What is your name? "
let nameString = System.Console.ReadLine()

printfn "What is your age? "
let ageString = System.Console.ReadLine()
let age = System.Int32.Parse(ageString)

if age < 10
then printfn "You are only %d years old and already learning F#? Wow!" age
```

```
10 is less than 20
What is your name? John
How old are you? 9
You are only 9 years old and already learning F#? Wow!
```

See also

- F# Language Reference

Match expressions

7/30/2019 • 3 minutes to read • Edit Online

The `match` expression provides branching control that is based on the comparison of an expression with a set of patterns.

Syntax

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

Remarks

The pattern matching expressions allow for complex branching based on the comparison of a test expression with a set of patterns. In the `match` expression, the *test-expression* is compared with each pattern in turn, and when a match is found, the corresponding *result-expression* is evaluated and the resulting value is returned as the value of the match expression.

The pattern matching function shown in the previous syntax is a lambda expression in which pattern matching is performed immediately on the argument. The pattern matching function shown in the previous syntax is equivalent to the following.

```
fun arg ->
    match arg with
    | pattern1 [ when condition ] -> result-expression1
    | pattern2 [ when condition ] -> result-expression2
    | ...
```

For more information about lambda expressions, see Lambda Expressions: The `fun` Keyword.

The whole set of patterns should cover all the possible matches of the input variable. Frequently, you use the wildcard pattern (`_`) as the last pattern to match any previously unmatched input values.

The following code illustrates some of the ways in which the `match` expression is used. For a reference and examples of all the possible patterns that can be used, see Pattern Matching.

```
let list1 = [ 1; 5; 100; 450; 788 ]

// Pattern matching by using the cons pattern and a list
// pattern that tests for an empty list.
let rec printList listx =
    match listx with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1

// Pattern matching with multiple alternatives on the same line.
let filter123 x =
    match x with
    | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
    | a -> printfn "%d" a

// The same function written with the pattern matching
// function syntax.
let filterNumbers =
    function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
             | a -> printfn "%d" a
```

Guards on patterns

You can use a `when` clause to specify an additional condition that the variable must satisfy to match a pattern. Such a clause is referred to as a *guard*. The expression following the `when` keyword is not evaluated unless a match is made to the pattern associated with that guard.

The following example illustrates the use of a guard to specify a numeric range for a variable pattern. Note that multiple conditions are combined by using Boolean operators.

```
let rangeTest testValue mid size =
    match testValue with
    | var1 when var1 >= mid - size/2 && var1 <= mid + size/2 -> printfn "The test value is in range."
    | _ -> printfn "The test value is out of range."

rangeTest 10 20 5
rangeTest 10 20 10
rangeTest 10 20 40
```

Note that because values other than literals cannot be used in the pattern, you must use a `when` clause if you have to compare some part of the input against a value. This is shown in the following code:

```
// This example uses patterns that have when guards.
let detectValue point target =
    match point with
    | (a, b) when a = target && b = target -> printfn "Both values match target %d." target
    | (a, b) when a = target -> printfn "First value matched target in (%d, %d)" target b
    | (a, b) when b = target -> printfn "Second value matched target in (%d, %d)" a target
    | _ -> printfn "Neither value matches target."
detectValue (0, 0) 0
detectValue (1, 0) 0
detectValue (0, 10) 0
detectValue (10, 15) 0
```

Note that when a union pattern is covered by a guard, the guard applies to **all** of the patterns, not just the last one. For example, given the following code, the guard `when a > 12` applies to both `A a` and `B a`:

```
type Union =
    | A of int
    | B of int

let foo() =
    let test = A 42
    match test with
    | A a
    | B a when a > 41 -> a // the guard applies to both patterns
    | _ -> 1

foo() // returns 42
```

See also

- F# Language Reference
- Active Patterns
- Pattern Matching

Pattern Matching

8/2/2019 • 12 minutes to read • Edit Online

Patterns are rules for transforming input data. They are used throughout the F# language to compare data with a logical structure or structures, decompose data into constituent parts, or extract information from data in various ways.

Remarks

Patterns are used in many language constructs, such as the `match` expression. They are used when you are processing arguments for functions in `let` bindings, lambda expressions, and in the exception handlers associated with the `try...with` expression. For more information, see Match Expressions, let Bindings, Lambda Expressions: The `fun` Keyword, and Exceptions: The `try...with` Expression.

For example, in the `match` expression, the *pattern* is what follows the pipe symbol.

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

Each pattern acts as a rule for transforming input in some way. In the `match` expression, each pattern is examined in turn to see if the input data is compatible with the pattern. If a match is found, the result expression is executed. If a match is not found, the next pattern rule is tested. The optional when *condition* part is explained in Match Expressions.

Supported patterns are shown in the following table. At run time, the input is tested against each of the following patterns in the order listed in the table, and patterns are applied recursively, from first to last as they appear in your code, and from left to right for the patterns on each line.

NAME	DESCRIPTION	EXAMPLE
Constant pattern	Any numeric, character, or string literal, an enumeration constant, or a defined literal identifier	`1.0`, `"test"`, `30`, `Color.Red`
Identifier pattern	A case value of a discriminated union, an exception label, or an active pattern case	`Some(x)` `Failure(msg)`
Variable pattern	*identifier*	`a`
`as` pattern	*pattern* as *identifier*	`(a, b) as tuple1`
OR pattern	*pattern1* | *pattern2*	`([h] | [h; _])`
AND pattern	*pattern1* & *pattern2*	`(a, b) & (_, "test")`
Cons pattern	*identifier* :: *list-identifier*	`h :: t`

NAME	DESCRIPTION	EXAMPLE
List pattern	[*pattern_1*; ... ; *pattern_n*]	`[ a; b; c ]`
Array pattern	[| *pattern_1*; ..; *pattern_n* |]	`[| a; b; c |]`
Parenthesized pattern	(*pattern*)	`( a )`
Tuple pattern	(*pattern_1*, ... , *pattern_n*)	`( a, b )`
Record pattern	{ *identifier1* = *pattern_1*; ... ; *identifier_n* = *pattern_n* }	`{ Name = name; }`
Wildcard pattern	_	`_`
Pattern together with type annotation	*pattern* : *type*	`a : int`
Type test pattern	:? *type* [as *identifier*]	`:? System.DateTime as dt`
Null pattern	null	`null`

Constant Patterns

Constant patterns are numeric, character, and string literals, enumeration constants (with the enumeration type name included). A `match` expression that has only constant patterns can be compared to a case statement in other languages. The input is compared with the literal value and the pattern matches if the values are equal. The type of the literal must be compatible with the type of the input.

The following example demonstrates the use of literal patterns, and also uses a variable pattern and an OR pattern.

```
[<Literal>]
let Three = 3

let filter123 x =
    match x with
    // The following line contains literal patterns combined with an OR pattern.
    | 1 | 2 | Three -> printfn "Found 1, 2, or 3!"
    // The following line contains a variable pattern.
    | var1 -> printfn "%d" var1

for x in 1..10 do filter123 x
```

Another example of a literal pattern is a pattern based on enumeration constants. You must specify the enumeration type name when you use enumeration constants.

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2

let printColorName (color:Color) =
    match color with
    | Color.Red -> printfn "Red"
    | Color.Green -> printfn "Green"
    | Color.Blue -> printfn "Blue"
    | _ -> ()

printColorName Color.Red
printColorName Color.Green
printColorName Color.Blue
```

Identifier Patterns

If the pattern is a string of characters that forms a valid identifier, the form of the identifier determines how the pattern is matched. If the identifier is longer than a single character and starts with an uppercase character, the compiler tries to make a match to the identifier pattern. The identifier for this pattern could be a value marked with the Literal attribute, a discriminated union case, an exception identifier, or an active pattern case. If no matching identifier is found, the match fails and the next pattern rule, the variable pattern, is compared to the input.

Discriminated union patterns can be simple named cases or they can have a value, or a tuple containing multiple values. If there is a value, you must specify an identifier for the value. In the case of a tuple, you must supply a tuple pattern with an identifier for each element of the tuple or an identifier with a field name for one or more named union fields. See the code examples in this section for examples.

The `option` type is a discriminated union that has two cases, `Some` and `None`. One case (`Some`) has a value, but the other (`None`) is just a named case. Therefore, `Some` needs to have a variable for the value associated with the `Some` case, but `None` must appear by itself. In the following code, the variable `var1` is given the value that is obtained by matching to the `Some` case.

```
let printOption (data : int option) =
    match data with
    | Some var1  -> printfn "%d" var1
    | None -> ()
```

In the following example, the `PersonName` discriminated union contains a mixture of strings and characters that represent possible forms of names. The cases of the discriminated union are `FirstOnly` , `LastOnly` , and `FirstLast` .

```
type PersonName =
    | FirstOnly of string
    | LastOnly of string
    | FirstLast of string * string

let constructQuery personName =
    match personName with
    | FirstOnly(firstName) -> printf "May I call you %s?" firstName
    | LastOnly(lastName) -> printf "Are you Mr. or Ms. %s?" lastName
    | FirstLast(firstName, lastName) -> printf "Are you %s %s?" firstName lastName
```

For discriminated unions that have named fields, you use the equals sign (=) to extract the value of a named field. For example, consider a discriminated union with a declaration like the following.

```
type Shape =
    | Rectangle of height : float * width : float
    | Circle of radius : float
```

You can use the named fields in a pattern matching expression as follows.

```
let matchShape shape =
    match shape with
    | Rectangle(height = h) -> printfn "Rectangle with length %f" h
    | Circle(r) -> printfn "Circle with radius %f" r
```

The use of the named field is optional, so in the previous example, both `Circle(r)` and `Circle(radius = r)` have the same effect.

When you specify multiple fields, use the semicolon (;) as a separator.

```
match shape with
| Rectangle(height = h; width = w) -> printfn "Rectangle with height %f and width %f" h w
| _ -> ()
```

Active patterns enable you to define more complex custom pattern matching. For more information about active patterns, see Active Patterns.

The case in which the identifier is an exception is used in pattern matching in the context of exception handlers. For information about pattern matching in exception handling, see Exceptions: The `try...with` Expression.

Variable Patterns

The variable pattern assigns the value being matched to a variable name, which is then available for use in the execution expression to the right of the `->` symbol. A variable pattern alone matches any input, but variable patterns often appear within other patterns, therefore enabling more complex structures such as tuples and arrays to be decomposed into variables.

The following example demonstrates a variable pattern within a tuple pattern.

```
let function1 x =
    match x with
    | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2
    | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2
    | (var1, var2) -> printfn "%d equals %d" var1 var2

function1 (1,2)
function1 (2, 1)
function1 (0, 0)
```

as Pattern

The `as` pattern is a pattern that has an `as` clause appended to it. The `as` clause binds the matched value to a name that can be used in the execution expression of a `match` expression, or, in the case where this pattern is used in a `let` binding, the name is added as a binding to the local scope.

The following example uses an `as` pattern.

```
let (var1, var2) as tuple1 = (1, 2)
printfn "%d %d %A" var1 var2 tuple1
```

OR Pattern

The OR pattern is used when input data can match multiple patterns, and you want to execute the same code as a result. The types of both sides of the OR pattern must be compatible.

The following example demonstrates the OR pattern.

```
let detectZeroOR point =
    match point with
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."
    | _ -> printfn "Both nonzero."
detectZeroOR (0, 0)
detectZeroOR (1, 0)
detectZeroOR (0, 10)
detectZeroOR (10, 15)
```

AND Pattern

The AND pattern requires that the input match two patterns. The types of both sides of the AND pattern must be compatible.

The following example is like `detectZeroTuple` shown in the Tuple Pattern section later in this topic, but here both `var1` and `var2` are obtained as values by using the AND pattern.

```
let detectZeroAND point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (var1, var2) & (0, _) -> printfn "First value is 0 in (%d, %d)" var1 var2
    | (var1, var2)  & (_, 0) -> printfn "Second value is 0 in (%d, %d)" var1 var2
    | _ -> printfn "Both nonzero."
detectZeroAND (0, 0)
detectZeroAND (1, 0)
detectZeroAND (0, 10)
detectZeroAND (10, 15)
```

Cons Pattern

The cons pattern is used to decompose a list into the first element, the *head*, and a list that contains the remaining elements, the *tail*.

```
let list1 = [ 1; 2; 3; 4 ]

// This example uses a cons pattern and a list pattern.
let rec printList l =
    match l with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1
```

List Pattern

The list pattern enables lists to be decomposed into a number of elements. The list pattern itself can match only lists of a specific number of elements.

```
// This example uses a list pattern.
let listLength list =
    match list with
    | [] -> 0
    | [ _ ] -> 1
    | [ _; _ ] -> 2
    | [ _; _; _ ] -> 3
    | _ -> List.length list

printfn "%d" (listLength [ 1 ])
printfn "%d" (listLength [ 1; 1 ])
printfn "%d" (listLength [ 1; 1; 1; ])
printfn "%d" (listLength [ ] )
```

Array Pattern

The array pattern resembles the list pattern and can be used to decompose arrays of a specific length.

```
// This example uses array patterns.
let vectorLength vec =
    match vec with
    | [| var1 |] -> var1
    | [| var1; var2 |] -> sqrt (var1*var1 + var2*var2)
    | [| var1; var2; var3 |] -> sqrt (var1*var1 + var2*var2 + var3*var3)
    | _ -> failwith "vectorLength called with an unsupported array size of %d." (vec.Length)

printfn "%f" (vectorLength [| 1. |])
printfn "%f" (vectorLength [| 1.; 1. |])
printfn "%f" (vectorLength [| 1.; 1.; 1.; |])
printfn "%f" (vectorLength [| |] )
```

Parenthesized Pattern

Parentheses can be grouped around patterns to achieve the desired associativity. In the following example, parentheses are used to control associativity between an AND pattern and a cons pattern.

```
let countValues list value =
    let rec checkList list acc =
       match list with
       | (elem1 & head) :: tail when elem1 = value -> checkList tail (acc + 1)
       | head :: tail -> checkList tail acc
       | [] -> acc
    checkList list 0

let result = countValues [ for x in -10..10 -> x*x - 4 ] 0
printfn "%d" result
```

Tuple Pattern

The tuple pattern matches input in tuple form and enables the tuple to be decomposed into its constituent elements by using pattern matching variables for each position in the tuple.

The following example demonstrates the tuple pattern and also uses literal patterns, variable patterns, and the wildcard pattern.

```
let detectZeroTuple point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (0, var2) -> printfn "First value is 0 in (0, %d)" var2
    | (var1, 0) -> printfn "Second value is 0 in (%d, 0)" var1
    | _ -> printfn "Both nonzero."
detectZeroTuple (0, 0)
detectZeroTuple (1, 0)
detectZeroTuple (0, 10)
detectZeroTuple (10, 15)
```

Record Pattern

The record pattern is used to decompose records to extract the values of fields. The pattern does not have to reference all fields of the record; any omitted fields just do not participate in matching and are not extracted.

```
// This example uses a record pattern.

type MyRecord = { Name: string; ID: int }

let IsMatchByName record1 (name: string) =
    match record1 with
    | { MyRecord.Name = nameFound; MyRecord.ID = _; } when nameFound = name -> true
    | _ -> false

let recordX = { Name = "Parker"; ID = 10 }
let isMatched1 = IsMatchByName recordX "Parker"
let isMatched2 = IsMatchByName recordX "Hartono"
```

Wildcard Pattern

The wildcard pattern is represented by the underscore (`_`) character and matches any input, just like the variable pattern, except that the input is discarded instead of assigned to a variable. The wildcard pattern is often used within other patterns as a placeholder for values that are not needed in the expression to the right of the `->` symbol. The wildcard pattern is also frequently used at the end of a list of patterns to match any unmatched input. The wildcard pattern is demonstrated in many code examples in this topic. See the preceding code for one example.

Patterns That Have Type Annotations

Patterns can have type annotations. These behave like other type annotations and guide inference like other type annotations. Parentheses are required around type annotations in patterns. The following code shows a pattern that has a type annotation.

```
let detect1 x =
    match x with
    | 1 -> printfn "Found a 1!"
    | (var1 : int) -> printfn "%d" var1
detect1 0
detect1 1
```

Type Test Pattern

The type test pattern is used to match the input against a type. If the input type is a match to (or a derived type of) the type specified in the pattern, the match succeeds.

The following example demonstrates the type test pattern.

```
open System.Windows.Forms

let RegisterControl(control:Control) =
    match control with
    | :? Button as button -> button.Text <- "Registered."
    | :? CheckBox as checkbox -> checkbox.Text <- "Registered."
    | _ -> ()
```

Null Pattern

The null pattern matches the null value that can appear when you are working with types that allow a null value. Null patterns are frequently used when interoperating with .NET Framework code. For example, the return value of a .NET API might be the input to a `match` expression. You can control program flow based on whether the return value is null, and also on other characteristics of the returned value. You can use the null pattern to prevent null values from propagating to the rest of your program.

The following example uses the null pattern and the variable pattern.

```
let ReadFromFile (reader : System.IO.StreamReader) =
    match reader.ReadLine() with
    | null -> printfn "\n"; false
    | line -> printfn "%s" line; true

let fs = System.IO.File.Open("..\..\Program.fs", System.IO.FileMode.Open)
let sr = new System.IO.StreamReader(fs)
while ReadFromFile(sr) = true do ()
sr.Close()
```

See also

- Match Expressions
- Active Patterns
- F# Language Reference

Active Patterns

7/30/2019 • 7 minutes to read • Edit Online

Active patterns enable you to define named partitions that subdivide input data, so that you can use these names in a pattern matching expression just as you would for a discriminated union. You can use active patterns to decompose data in a customized manner for each partition.

Syntax

```
// Active pattern of one choice.
let (|identifier|) [arguments] valueToMatch= expression

// Active Pattern with multiple choices.
// Uses a FSharp.Core.Choice<_,...,_> based on the number of case names. In F#, the limitation n <= 7
applies.
let (|identifer1|identifier2|...|) valueToMatch = expression

// Partial active pattern definition.
// Uses a FSharp.Core.option<_> to represent if the type is satisfied at the call site.
let (|identifier|_|) [arguments ] valueToMatch = expression
```

Remarks

In the previous syntax, the identifiers are names for partitions of the input data that is represented by *arguments*, or, in other words, names for subsets of the set of all values of the arguments. There can be up to seven partitions in an active pattern definition. The *expression* describes the form into which to decompose the data. You can use an active pattern definition to define the rules for determining which of the named partitions the values given as arguments belong to. The (| and |) symbols are referred to as *banana clips* and the function created by this type of let binding is called an *active recognizer*.

As an example, consider the following active pattern with an argument.

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

You can use the active pattern in a pattern matching expression, as in the following example.

```
let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

The output of this program is as follows:

```
7 is odd
11 is odd
32 is even
```

Another use of active patterns is to decompose data types in multiple ways, such as when the same underlying data has various possible representations. For example, a `Color` object could be decomposed into an RGB representation or an HSB representation.

```
open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
   ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: System.Drawing.Color) =
   match col with
   | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: System.Drawing.Color) =
   match col with
   | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
  printfn "%s" colorString
  printRGB col
  printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"
```

The output of the above program is as follows:

```
Red
 Red: 255 Green: 0 Blue: 0
 Hue: 360.000000 Saturation: 1.000000 Brightness: 0.500000
Black
 Red: 0 Green: 0 Blue: 0
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
 Red: 255 Green: 255 Blue: 255
 Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
 Red: 128 Green: 128 Blue: 128
 Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
 Red: 255 Green: 235 Blue: 205
 Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961
```

In combination, these two ways of using active patterns enable you to partition and decompose data into just the appropriate form and perform the appropriate computations on the appropriate data in the form most convenient for the computation.

The resulting pattern matching expressions enable data to be written in a convenient way that is very readable, greatly simplifying potentially complex branching and data analysis code.

Partial Active Patterns

Sometimes, you need to partition only part of the input space. In that case, you write a set of partial patterns each of which match some inputs but fail to match other inputs. Active patterns that do not always produce a value are called *partial active patterns*; they have a return value that is an option type. To define a partial active pattern, you

use a wildcard character (_) at the end of the list of patterns inside the banana clips. The following code illustrates the use of a partial active pattern.

```
let (|Integer|_|) (str: string) =
   let mutable intvalue = 0
   if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
   else None

let (|Float|_|) (str: string) =
   let mutable floatvalue = 0.0
   if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
   else None

let parseNumeric str =
   match str with
     | Integer i -> printfn "%d : Integer" i
     | Float f -> printfn "%f : Floating point" f
     | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"
```

The output of the previous example is as follows:

```
1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.
```

When using partial active patterns, sometimes the individual choices can be disjoint or mutually exclusive, but they need not be. In the following example, the pattern Square and the pattern Cube are not disjoint, because some numbers are both squares and cubes, such as 64. The following program uses the AND pattern to combine the Square and Cube patterns. It print out all integers up to 1000 that are both squares and cubes, as well as those which are only cubes.

```
let err = 1.e-10

let isNearlyIntegral (x:float) = abs (x - round(x)) < err

let (|Square|_|) (x : int) =
  if isNearlyIntegral (sqrt (float x)) then Some(x)
  else None

let (|Cube|_|) (x : int) =
  if isNearlyIntegral ((float x) ** ( 1.0 / 3.0)) then Some(x)
  else None

let examineNumber x =
   match x with
      | Cube x -> printfn "%d is a cube" x
      | _ -> ()
   match x with
      | Square x -> printfn "%d is a square" x
      | _ -> ()

let findSquareCubes x =
   match x with
       | Cube x & Square _ -> printfn "%d is a cube and a square" x
       | Cube x -> printfn "%d is a cube" x
       | _ -> ()


[ 1 .. 1000 ] |> List.iter (fun elem -> findSquareCubes elem)
```

The output is as follows:

```
1 is a cube and a square
8 is a cube
27 is a cube
64 is a cube and a square
125 is a cube
216 is a cube
343 is a cube
512 is a cube
729 is a cube and a square
1000 is a cube
```

Parameterized Active Patterns

Active patterns always take at least one argument for the item being matched, but they may take additional arguments as well, in which case the name *parameterized active pattern* applies. Additional arguments allow a general pattern to be specialized. For example, active patterns that use regular expressions to parse strings often include the regular expression as an extra parameter, as in the following code, which also uses the partial active pattern `Integer` defined in the previous code example. In this example, strings that use regular expressions for various date formats are given to customize the general ParseRegex active pattern. The Integer active pattern is used to convert the matched strings into integers that can be passed to the DateTime constructor.

```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
   let m = Regex(regex).Match(str)
   if m.Success
   then Some (List.tail [ for x in m.Groups -> x.Value ])
   else None

// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
   match str with
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y + 2000, m, d)
     | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Integer y]
          -> new System.DateTime(y, m, d)
     | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Integer d]
          -> new System.DateTime(y, m, d)
     | _ -> new System.DateTime()

let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.ToString())
```

The output of the previous code is as follows:

```
12/22/2008 12:00:00 AM 1/1/2009 12:00:00 AM 1/15/2008 12:00:00 AM 12/28/1995 12:00:00 AM
```

Active patterns are not restricted only to pattern matching expressions, you can also use them on let-bindings.

```
let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e

let greet (Default "random citizen" name) =
    printfn "Hello, %s!" name

greet None
greet (Some "George")
```

The output of the previous code is as follows:

```
Hello, random citizen!
Hello, George!
```

See also

- F# Language Reference
- Match Expressions

Loops: for...to Expression

7/30/2019 • 2 minutes to read • Edit Online

The `for...to` expression is used to iterate in a loop over a range of values of a loop variable.

Syntax

```
for identifier = start [ to | downto ] finish do
    body-expression
```

Remarks

The type of the identifier is inferred from the type of the *start* and *finish* expressions. Types for these expressions must be 32-bit integers.

Although technically an expression, `for...to` is more like a traditional statement in an imperative programming language. The return type for the *body-expression* must be `unit`. The following examples show various uses of the `for...to` expression.

```
// A simple for...to loop.
let function1() =
  for i = 1 to 10 do
    printf "%d " i
  printfn ""

// A for...to loop that counts in reverse.
let function2() =
  for i = 10 downto 1 do
    printf "%d " i
  printfn ""

function1()
function2()

// A for...to loop that uses functions as the start and finish expressions.
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function3 x y =
  for i = (beginning x y) to (ending x y) do
     printf "%d " i
  printfn ""

function3 10 4
```

The output of the previous code is as follows.

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

See also

- F# Language Reference
- Loops: `for...in` Expression
- Loops: `while...do` Expression

Loops: for...in Expression

7/30/2019 • 3 minutes to read • Edit Online

This looping construct is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.

Syntax

```
for pattern in enumerable-expression do
    body-expression
```

Remarks

The `for...in` expression can be compared to the `for each` statement in other .NET languages because it is used to loop over the values in an enumerable collection. However, `for...in` also supports pattern matching over the collection instead of just iteration over the whole collection.

The enumerable expression can be specified as an enumerable collection or, by using the `..` operator, as a range on an integral type. Enumerable collections include lists, sequences, arrays, sets, maps, and so on. Any type that implements `System.Collections.IEnumerable` can be used.

When you express a range by using the `..` operator, you can use the following syntax.

start .. finish

You can also use a version that includes an increment called the *skip*, as in the following code.

start .. skip .. finish

When you use integral ranges and a simple counter variable as a pattern, the typical behavior is to increment the counter variable by 1 on each iteration, but if the range includes a skip value, the counter is incremented by the skip value instead.

Values matched in the pattern can also be used in the body expression.

The following code examples illustrate the use of the `for...in` expression.

```
// Looping over a list.
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
   printfn "%d" i
```

The output is as follows.

```
1
5
100
450
788
```

The following example shows how to loop over a sequence, and how to use a tuple pattern instead of a simple variable.

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
  printfn "%d squared is %d" a asqr
```

The output is as follows.

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

The following example shows how to loop over a simple integer range.

```
let function1() =
  for i in 1 .. 10 do
    printf "%d " i
  printfn ""
function1()
```

The output of function1 is as follows.

```
1 2 3 4 5 6 7 8 9 10
```

The following example shows how to loop over a range with a skip of 2, which includes every other element of the range.

```
let function2() =
  for i in 1 .. 2 .. 10 do
     printf "%d " i
  printfn ""
function2()
```

The output of `function2` is as follows.

```
1 3 5 7 9
```

The following example shows how to use a character range.

```
let function3() =
  for c in 'a' .. 'z' do
    printf "%c " c
  printfn ""
function3()
```

The output of `function3` is as follows.

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

The following example shows how to use a negative skip value for a reverse iteration.

```
let function4() =
    for i in 10 .. -1 .. 1 do
        printf "%d " i
    printfn " ... Lift off!"
function4()
```

The output of `function4` is as follows.

```
10 9 8 7 6 5 4 3 2 1 ... Lift off!
```

The beginning and ending of the range can also be expressions, such as functions, as in the following code.

```
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function5 x y =
  for i in (beginning x y) .. (ending x y) do
     printf "%d " i
  printfn ""

function5 10 4
```

The output of `function5` with this input is as follows.

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

The next example shows the use of a wildcard character (_) when the element is not needed in the loop.

```
let mutable count = 0
for _ in list1 do
   count <- count + 1
printfn "Number of elements in list1: %d" count
```

The output is as follows.

```
Number of elements in list1: 5
```

`Note` You can use `for...in` in sequence expressions and other computation expressions, in which case a customized version of the `for...in` expression is used. For more information, see Sequences, Asynchronous Workflows, and Computation Expressions.

See also

- F# Language Reference
- Loops: `for...to` Expression
- Loops: `while...do` Expression

Loops: while...do Expression

7/30/2019 • 2 minutes to read • Edit Online

The `while...do` expression is used to perform iterative execution (looping) while a specified test condition is true.

Syntax

```
while test-expression do
    body-expression
```

Remarks

The *test-expression* is evaluated; if it is `true`, the *body-expression* is executed and the test expression is evaluated again. The *body-expression* must have type `unit`. If the test expression is `false`, the iteration ends.

The following example illustrates the use of the `while...do` expression.

```
open System

let lookForValue value maxValue =
  let mutable continueLooping = true
  let randomNumberGenerator = new Random()
  while continueLooping do
    // Generate a random number between 1 and maxValue.
    let rand = randomNumberGenerator.Next(maxValue)
    printf "%d " rand
    if rand = value then
       printfn "\nFound a %d!" value
       continueLooping <- false

lookForValue 10 20
```

The output of the previous code is a stream of random numbers between 1 and 20, the last of which is 10.

```
13 19 8 18 16 2 10
Found a 10!
```

NOTE

You can use `while...do` in sequence expressions and other computation expressions, in which case a customized version of the `while...do` expression is used. For more information, see Sequences, Asynchronous Workflows, and Computation Expressions.

See also

- F# Language Reference
- Loops: `for...in` Expression
- Loops: `for...to` Expression

Assertions

7/30/2019 • 2 minutes to read • Edit Online

The `assert` expression is a debugging feature that you can use to test an expression. Upon failure in Debug mode, an assertion generates a system error dialog box.

Syntax

```
assert condition
```

Remarks

The `assert` expression has type `bool -> unit`.

In the previous syntax, *condition* represents a Boolean expression that is to be tested. If the expression evaluates to `true`, execution continues unaffected. If it evaluates to `false`, a system error dialog box is generated. The error dialog box has a caption that contains the string **Assertion Failed**. The dialog box contains a stack trace that indicates where the assertion failure occurred.

Assertion checking is enabled only when you compile in Debug mode; that is, if the constant `DEBUG` is defined. In the project system, by default, the `DEBUG` constant is defined in the Debug configuration but not in the Release configuration.

The assertion failure error cannot be caught by using F# exception handling.

> **NOTE**
>
> The `assert` function resolves to Debug.Assert.

Example

The following code example illustrates the use of the `assert` expression.

```
let subtractUnsigned (x : uint32) (y : uint32) =
    assert (x > y)
    let z = x - y
    z
// This code does not generate an assertion failure.
let result1 = subtractUnsigned 2u 1u
// This code generates an assertion failure.
let result2 = subtractUnsigned 1u 2u
```

See also

- F# Language Reference

Exception Handling

5/15/2019 • 2 minutes to read • Edit Online

This section contains information about exception handling support in the F# language.

Exception Handling Basics

Exception handling is the standard way of handling error conditions in the .NET Framework. Thus, any .NET language must support this mechanism, including F#. An *exception* is an object that encapsulates information about an error. When errors occur, exceptions are raised and regular execution stops. Instead, the runtime searches for an appropriate handler for the exception. The search starts in the current function, and proceeds up the stack through the layers of callers until a matching handler is found. Then the handler is executed.

In addition, as the stack is unwound, the runtime executes any code in `finally` blocks, to guarantee that objects are cleaned up correctly during the unwinding process.

Related Topics

TITLE	DESCRIPTION
Exception Types	Describes how to declare an exception type.
Exceptions: The `try...with` Expression	Describes the language construct that supports exception handling.
Exceptions: The `try...finally` Expression	Describes the language construct that enables you to execute clean-up code as the stack unwinds when an exception is thrown.
Exceptions: the `raise` Function	Describes how to throw an exception object.
Exceptions: The `failwith` Function	Describes how to generate a general F# exception.
Exceptions: The `invalidArg` Function	Describes how to generate an invalid argument exception.

Exception Types

7/30/2019 • 2 minutes to read • Edit Online

There are two categories of exceptions in F#: .NET exception types and F# exception types. This topic describes how to define and use F# exception types.

Syntax

```
exception exception-type of argument-type
```

Remarks

In the previous syntax, *exception-type* is the name of a new F# exception type, and *argument-type* represents the type of an argument that can be supplied when you raise an exception of this type. You can specify multiple arguments by using a tuple type for *argument-type*.

A typical definition for an F# exception resembles the following.

```
exception MyError of string
```

You can generate an exception of this type by using the `raise` function, as follows.

```
raise (MyError("Error message"))
```

You can use an F# exception type directly in the filters in a `try...with` expression, as shown in the following example.

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

The exception type that you define with the `exception` keyword in F# is a new type that inherits from `System.Exception`.

See also

- Exception Handling
- Exceptions: the `raise` Function
- Exception Hierarchy

Exceptions: The try...with Expression

7/30/2019 • 3 minutes to read • Edit Online

This topic describes the `try...with` expression, the expression that is used for exception handling in the F# language.

Syntax

```
try
    expression1
with
| pattern1 -> expression2
| pattern2 -> expression3
...
```

Remarks

The `try...with` expression is used to handle exceptions in F#. It is similar to the `try...catch` statement in C#. In the preceding syntax, the code in *expression1* might generate an exception. The `try...with` expression returns a value. If no exception is thrown, the whole expression returns the value of *expression1*. If an exception is thrown, each *pattern* is compared in turn with the exception, and for the first matching pattern, the corresponding *expression*, known as the *exception handler*, for that branch is executed, and the overall expression returns the value of the expression in that exception handler. If no pattern matches, the exception propagates up the call stack until a matching handler is found. The types of the values returned from each expression in the exception handlers must match the type returned from the expression in the `try` block.

Frequently, the fact that an error occurred also means that there is no valid value that can be returned from the expressions in each exception handler. A frequent pattern is to have the type of the expression be an option type. The following code example illustrates this pattern.

```
let divide1 x y =
   try
      Some (x / y)
   with
      | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divide1 100 0
```

Exceptions can be .NET exceptions, or they can be F# exceptions. You can define F# exceptions by using the `exception` keyword.

You can use a variety of patterns to filter on the exception type and other conditions; the options are summarized in the following table.

PATTERN	DESCRIPTION
:? *exception-type*	Matches the specified .NET exception type.
:? *exception-type* as *identifier*	Matches the specified .NET exception type, but gives the exception a named value.

PATTERN	DESCRIPTION
exception-name(*arguments*)	Matches an F# exception type and binds the arguments.
identifier	Matches any exception and binds the name to the exception object. Equivalent to **:? System.Exception as***identifier*
identifier when *condition*	Matches any exception if the condition is true.

Examples

The following code examples illustrate the use of the various exception handler patterns.

```
// This example shows the use of the as keyword to assign a name to a
// .NET exception.
let divide2 x y =
  try
    Some( x / y )
  with
    | :? System.DivideByZeroException as ex -> printfn "Exception! %s " (ex.Message); None

// This version shows the use of a condition to branch to multiple paths
// with the same exception.
let divide3 x y flag =
  try
     x / y
  with
     | ex when flag -> printfn "TRUE: %s" (ex.ToString()); 0
     | ex when not flag -> printfn "FALSE: %s" (ex.ToString()); 1

let result2 = divide3 100 0 true

// This version shows the use of F# exceptions.
exception Error1 of string
exception Error2 of string * int

let function1 x y =
   try
      if x = y then raise (Error1("x"))
      else raise (Error2("x", 10))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

NOTE

The `try...with` construct is a separate expression from the `try...finally` expression. Therefore, if your code requires both a `with` block and a `finally` block, you will have to nest the two expressions.

NOTE

You can use `try...with` in asynchronous workflows and other computation expressions, in which case a customized version of the `try...with` expression is used. For more information, see Asynchronous Workflows, and Computation Expressions.

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...finally` Expression

Exceptions: The try...finally Expression

7/30/2019 • 2 minutes to read • Edit Online

The `try...finally` expression enables you to execute clean-up code even if a block of code throws an exception.

Syntax

```
try
    expression1
finally
    expression2
```

Remarks

The `try...finally` expression can be used to execute the code in *expression2* in the preceding syntax regardless of whether an exception is generated during the execution of *expression1*.

The type of *expression2* does not contribute to the value of the whole expression; the type returned when an exception does not occur is the last value in *expression1*. When an exception does occur, no value is returned and the flow of control transfers to the next matching exception handler up the call stack. If no exception handler is found, the program terminates. Before the code in a matching handler is executed or the program terminates, the code in the `finally` branch is executed.

The following code demonstrates the use of the `try...finally` expression.

```
let divide x y =
   let stream : System.IO.FileStream = System.IO.File.Create("test.txt")
   let writer : System.IO.StreamWriter = new System.IO.StreamWriter(stream)
   try
      writer.WriteLine("test1");
      Some( x / y )
   finally
      writer.Flush()
      printfn "Closing stream"
      stream.Close()

let result =
  try
     divide 100 0
  with
     | :? System.DivideByZeroException -> printfn "Exception handled."; None
```

The output to the console is as follows.

```
Closing stream
Exception handled.
```

As you can see from the output, the stream was closed before the outer exception was handled, and the file `test.txt` contains the text `test1`, which indicates that the buffers were flushed and written to disk even though the exception transferred control to the outer exception handler.

Note that the `try...with` construct is a separate construct from the `try...finally` construct. Therefore, if your

code requires both a `with` block and a `finally` block, you have to nest the two constructs, as in the following code example.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
   try
     try
        if x = y then raise (InnerError("inner"))
        else raise (OuterError("outer"))
     with
      | InnerError(str) -> printfn "Error1 %s" str
   finally
      printfn "Always print this."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

In the context of computation expressions, including sequence expressions and asynchronous workflows, **try...finally** expressions can have a custom implementation. For more information, see Computation Expressions.

See also

- Exception Handling
- Exceptions: The `try...with` Expression

Exceptions: the raise Function

7/30/2019 • 2 minutes to read • Edit Online

The `raise` function is used to indicate that an error or exceptional condition has occurred. Information about the error is captured in an exception object.

Syntax

```
raise (expression)
```

Remarks

The `raise` function generates an exception object and initiates a stack unwinding process. The stack unwinding process is managed by the common language runtime (CLR), so the behavior of this process is the same as it is in any other .NET language. The stack unwinding process is a search for an exception handler that matches the generated exception. The search starts in the current `try...with` expression, if there is one. Each pattern in the `with` block is checked, in order. When a matching exception handler is found, the exception is considered handled; otherwise, the stack is unwound and `with` blocks up the call chain are checked until a matching handler is found. Any `finally` blocks that are encountered in the call chain are also executed in sequence as the stack unwinds.

The `raise` function is the equivalent of `throw` in C# or C++. Use `reraise` in a catch handler to propagate the same exception up the call chain.

The following code examples illustrate the use of the `raise` function to generate an exception.

```
exception InnerError of string
exception OuterError of string

let function1 x y =
   try
     try
        if x = y then raise (InnerError("inner"))
        else raise (OuterError("outer"))
     with
      | InnerError(str) -> printfn "Error1 %s" str
   finally
      printfn "Always print this."


let function2 x y =
  try
     function1 x y
  with
     | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

The `raise` function can also be used to raise .NET exceptions, as shown in the following example.

```
let divide x y =
  if (y = 0) then raise (System.ArgumentException("Divisor cannot be zero!"))
  else
     x / y
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: The `failwith` Function
- Exceptions: The `invalidArg` Function

Exceptions: The failwith Function

7/30/2019 • 2 minutes to read • Edit Online

The `failwith` function generates an F# exception.

Syntax

```
failwith error-message-string
```

Remarks

The *error-message-string* in the previous syntax is a literal string or a value of type `string`. It becomes the `Message` property of the exception.

The exception that is generated by `failwith` is a `System.Exception` exception, which is a reference that has the name `Failure` in F# code. The following code illustrates the use of `failwith` to throw an exception.

```
let divideFailwith x y =
  if (y = 0) then failwith "Divisor cannot be zero."
  else
    x / y

let testDivideFailwith x y =
  try
     divideFailwith x y
  with
     | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: the `raise` Function

Exceptions: The invalidArg Function

7/30/2019 • 2 minutes to read • Edit Online

The `invalidArg` function generates an argument exception.

Syntax

```
invalidArg parameter-name error-message-string
```

Remarks

The parameter-name in the previous syntax is a string with the name of the parameter whose argument was invalid. The *error-message-string* is a literal string or a value of type `string`. It becomes the `Message` property of the exception object.

The exception generated by `invalidArg` is a `System.ArgumentException` exception. The following code illustrates the use of `invalidArg` to throw an exception.

```
let months = [| "January"; "February"; "March"; "April";
                "May"; "June"; "July"; "August"; "September";
                "October"; "November"; "December" |]

let lookupMonth month =
   if (month > 12 || month < 1)
     then invalidArg "month" (sprintf "Value passed in was %d." month)
   months.[month - 1]

printfn "%s" (lookupMonth 12)
printfn "%s" (lookupMonth 1)
printfn "%s" (lookupMonth 13)
```

The output is the following, followed by a stack trace (not shown).

```
December
January
System.ArgumentException: Month parameter out of range.
```

See also

- Exception Handling
- Exception Types
- Exceptions: The `try...with` Expression
- Exceptions: The `try...finally` Expression
- Exceptions: the `raise` Function
- Exceptions: The `failwith` Function

Attributes

8/17/2019 • 3 minutes to read • Edit Online

Attributes enable metadata to be applied to a programming construct.

Syntax

```
[<target:attribute-name(arguments)>]
```

Remarks

In the previous syntax, the *target* is optional and, if present, specifies the kind of program entity that the attribute applies to. Valid values for *target* are shown in the table that appears later in this document.

The *attribute-name* refers to the name (possibly qualified with namespaces) of a valid attribute type, with or without the suffix `Attribute` that is usually used in attribute type names. For example, the type `ObsoleteAttribute` can be shortened to just `Obsolete` in this context.

The *arguments* are the arguments to the constructor for the attribute type. If an attribute has a default constructor, the argument list and parentheses can be omitted. Attributes support both positional arguments and named arguments. *Positional arguments* are arguments that are used in the order in which they appear. Named arguments can be used if the attribute has public properties. You can set these by using the following syntax in the argument list.

```
*property-name* = *property-value*
```

Such property initializations can be in any order, but they must follow any positional arguments. Following is an example of an attribute that uses positional arguments and property initializations.

```
open System.Runtime.InteropServices

[<DllImport("kernel32", SetLastError=true)>]
extern bool CloseHandle(nativeint handle)
```

In this example, the attribute is `DllImportAttribute`, here used in shortened form. The first argument is a positional parameter and the second is a property.

Attributes are a .NET programming construct that enables an object known as an *attribute* to be associated with a type or other program element. The program element to which an attribute is applied is known as the *attribute target*. The attribute usually contains metadata about its target. In this context, metadata could be any data about the type other than its fields and members.

Attributes in F# can be applied to the following programming constructs: functions, methods, assemblies, modules, types (classes, records, structures, interfaces, delegates, enumerations, unions, and so on), constructors, properties, fields, parameters, type parameters, and return values. Attributes are not allowed on `let` bindings inside classes, expressions, or workflow expressions.

Typically, the attribute declaration appears directly before the declaration of the attribute target. Multiple attribute declarations can be used together, as follows.

```
[<Owner("Jason Carlson")>]
[<Company("Microsoft")>]
type SomeType1 =
```

You can query attributes at run time by using .NET reflection.

You can declare multiple attributes individually, as in the previous code example, or you can declare them in one set of brackets if you use a semicolon to separate the individual attributes and constructors, as shown here.

```
[<Owner("Darren Parker"); Company("Microsoft")>]
type SomeType2 =
```

Typically encountered attributes include the `Obsolete` attribute, attributes for security considerations, attributes for COM support, attributes that relate to ownership of code, and attributes indicating whether a type can be serialized. The following example demonstrates the use of the `Obsolete` attribute.

```
open System

[<Obsolete("Do not use. Use newFunction instead.")>]
let obsoleteFunction x y =
  x + y

let newFunction x y =
  x + 2 * y

// The use of the obsolete function produces a warning.
let result1 = obsoleteFunction 10 100
let result2 = newFunction 10 100
```

For the attribute targets `assembly` and `module`, you apply the attributes to a top-level `do` binding in your assembly. You can include the word `assembly` or `module` in the attribute declaration, as follows.

```
open System.Reflection
[<assembly:AssemblyVersionAttribute("1.0.0.0")>]
do
   printfn "Executing..."
```

If you omit the attribute target for an attribute applied to a `do` binding, the F# compiler attempts to determine the attribute target that makes sense for that attribute. Many attribute classes have an attribute of type `System.AttributeUsageAttribute` that includes information about the possible targets supported for that attribute. If the `System.AttributeUsageAttribute` indicates that the attribute supports functions as targets, the attribute is taken to apply to the main entry point of the program. If the `System.AttributeUsageAttribute` indicates that the attribute supports assemblies as targets, the compiler takes the attribute to apply to the assembly. Most attributes do not apply to both functions and assemblies, but in cases where they do, the attribute is taken to apply to the program's main function. If the attribute target is specified explicitly, the attribute is applied to the specified target.

Although you do not usually need to specify the attribute target explicitly, valid values for *target* in an attribute are shown in the following table, along with examples of usage.

ATTRIBUTE TARGET	EXAMPLE

assembly	`[<assembly: AssemblyVersionAttribute("1.0.0.0")>]`
return	`let function1 x : [<return: Obsolete>] int = x + 1`
field	`[<field: DefaultValue>] val mutable x: int`
property	`[<property: Obsolete>] this.MyProperty = x`
param	`member this.MyMethod([<param: Out>] x : ref<int>) = x := 10`

type	`[<type: StructLayout(Sequential)>]`
	`type MyStruct =`
	`struct`
	`x : byte`
	`y : int`
	`end`

See also

- F# Language Reference

Resource Management: The use Keyword

7/30/2019 • 3 minutes to read • Edit Online

This topic describes the keyword `use` and the `using` function, which can control the initialization and release of resources.

Resources

The term *resource* is used in more than one way. Yes, resources can be data that an application uses, such as strings, graphics, and the like, but in this context, *resources* refers to software or operating system resources, such as graphics device contexts, file handles, network and database connections, concurrency objects such as wait handles, and so on. The use of these resources by applications involves the acquisition of the resource from the operating system or other resource provider, followed by the later release of the resource to the pool so that it can be provided to another application. Problems occur when applications do not release resources back to the common pool.

Managing Resources

To efficiently and responsibly manage resources in an application, you must release resources promptly and in a predictable manner. The .NET Framework helps you do this by providing the `System.IDisposable` interface. A type that implements `System.IDisposable` has the `System.IDisposable.Dispose` method, which correctly frees resources. Well-written applications guarantee that `System.IDisposable.Dispose` is called promptly when any object that holds a limited resource is no longer needed. Fortunately, most .NET languages provide support to make this easier, and F# is no exception. There are two useful language constructs that support the dispose pattern: the `use` binding and the `using` function.

use Binding

The `use` keyword has a form that resembles that of the `let` binding:

use *value* = *expression*

It provides the same functionality as a `let` binding but adds a call to `Dispose` on the value when the value goes out of scope. Note that the compiler inserts a null check on the value, so that if the value is `null`, the call to `Dispose` is not attempted.

The following example shows how to close a file automatically by using the `use` keyword.

```
open System.IO

let writetofile filename obj =
   use file1 = File.CreateText(filename)
   file1.WriteLine("{0}", obj.ToString() )
   // file1.Dispose() is called implicitly here.

writetofile "abc.txt" "Humpty Dumpty sat on a wall."
```

NOTE

You can use `use` in computation expressions, in which case a customized version of the `use` expression is used. For more information, see Sequences, Asynchronous Workflows, and Computation Expressions.

using Function

The `using` function has the following form:

`using` (*expression1*) *function-or-lambda*

In a `using` expression, *expression1* creates the object that must be disposed. The result of *expression1* (the object that must be disposed) becomes an argument, *value*, to *function-or-lambda*, which is either a function that expects a single remaining argument of a type that matches the value produced by *expression1*, or a lambda expression that expects an argument of that type. At the end of the execution of the function, the runtime calls `Dispose` and frees the resources (unless the value is `null`, in which case the call to Dispose is not attempted).

The following example demonstrates the `using` expression with a lambda expression.

```
open System.IO

let writetofile2 filename obj =
    using (System.IO.File.CreateText(filename)) ( fun file1 ->
        file1.WriteLine("{0}", obj.ToString() )
    )

writetofile2 "abc2.txt" "The quick sly fox jumps over the lazy brown dog."
```

The next example shows the `using` expression with a function.

```
let printToFile (file1 : System.IO.StreamWriter) =
    file1.WriteLine("Test output");

using (System.IO.File.CreateText("test.txt")) printToFile
```

Note that the function could be a function that has some arguments applied already. The following code example demonstrates this. It creates a file that contains the string `XYZ`.

```
let printToFile2 obj (file1 : System.IO.StreamWriter) =
    file1.WriteLine(obj.ToString())

using (System.IO.File.CreateText("test.txt")) (printToFile2 "XYZ")
```

The `using` function and the `use` binding are nearly equivalent ways to accomplish the same thing. The `using` keyword provides more control over when `Dispose` is called. When you use `using`, `Dispose` is called at the end of the function or lambda expression; when you use the `use` keyword, `Dispose` is called at the end of the containing code block. In general, you should prefer to use `use` instead of the `using` function.

See also

- F# Language Reference

Namespaces

7/30/2019 • 4 minutes to read • Edit Online

A namespace lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of F# program elements. Namespaces are typically top-level elements in F# files.

Syntax

```
namespace [rec] [parent-namespaces.]identifier
```

Remarks

If you want to put code in a namespace, the first declaration in the file must declare the namespace. The contents of the entire file then become part of the namespace, provided no other namespaces declaration exists further in the file. If that is the case, then all code up until the next namespace declaration is considered to be within the first namespace.

Namespaces cannot directly contain values and functions. Instead, values and functions must be included in modules, and modules are included in namespaces. Namespaces can contain types, modules.

XML doc comments can be declared above a namespace, but they're ignored. Compiler directives can also be declared above a namespace.

Namespaces can be declared explicitly with the namespace keyword, or implicitly when declaring a module. To declare a namespace explicitly, use the namespace keyword followed by the namespace name. The following example shows a code file that declares a namespace `Widgets` with a type and a module included in that namespace.

```
namespace Widgets

type MyWidget1 =
    member this.WidgetName = "Widget1"

module WidgetsModule =
    let widgetName = "Widget2"
```

If the entire contents of the file are in one module, you can also declare namespaces implicitly by using the `module` keyword and providing the new namespace name in the fully qualified module name. The following example shows a code file that declares a namespace `Widgets` and a module `WidgetsModule`, which contains a function.

```
module Widgets.WidgetModule

let widgetFunction x y =
   printfn "%A %A" x y
```

The following code is equivalent to the preceding code, but the module is a local module declaration. In that case, the namespace must appear on its own line.

```
namespace Widgets

module WidgetModule =

    let widgetFunction x y =
        printfn "%A %A" x y
```

If more than one module is required in the same file in one or more namespaces, you must use local module declarations. When you use local module declarations, you cannot use the qualified namespace in the module declarations. The following code shows a file that has a namespace declaration and two local module declarations. In this case, the modules are contained directly in the namespace; there is no implicitly created module that has the same name as the file. Any other code in the file, such as a `do` binding, is in the namespace but not in the inner modules, so you need to qualify the module member `widgetFunction` by using the module name.

```
namespace Widgets

module WidgetModule1 =
   let widgetFunction x y =
      printfn "Module1 %A %A" x y
module WidgetModule2 =
   let widgetFunction x y =
      printfn "Module2 %A %A" x y

module useWidgets =

  do
     WidgetModule1.widgetFunction 10 20
     WidgetModule2.widgetFunction 5 6
```

The output of this example is as follows.

```
Module1 10 20
Module2 5 6
```

For more information, see Modules.

Nested Namespaces

When you create a nested namespace, you must fully qualify it. Otherwise, you create a new top-level namespace. Indentation is ignored in namespace declarations.

The following example shows how to declare a nested namespace.

```
namespace Outer

    // Full name: Outer.MyClass
    type MyClass() =
       member this.X(x) = x + 1

// Fully qualify any nested namespaces.
namespace Outer.Inner

    // Full name: Outer.Inner.MyClass
    type MyClass() =
       member this.Prop1 = "X"
```

Namespaces in Files and Assemblies

Namespaces can span multiple files in a single project or compilation. The term *namespace fragment* describes the part of a namespace that is included in one file. Namespaces can also span multiple assemblies. For example, the `System` namespace includes the whole .NET Framework, which spans many assemblies and contains many nested namespaces.

Global Namespace

You use the predefined namespace `global` to put names in the .NET top-level namespace.

```
namespace global

type SomeType() =
    member this.SomeMember = 0
```

You can also use global to reference the top-level .NET namespace, for example, to resolve name conflicts with other namespaces.

```
global.System.Console.WriteLine("Hello World!")
```

Recursive namespaces

Namespaces can also be declared as recursive to allow for all contained code to be mutually recursive. This is done via `namespace rec`. Use of `namespace rec` can alleviate some pains in not being able to write mutually referential code between types and modules. The following is an example of this:

```
namespace rec MutualReferences

type Orientation = Up | Down
type PeelState = Peeled | Unpeeled

// This exception depends on the type below.
exception DontSqueezeTheBananaException of Banana

type BananaPeel() = class end

type Banana(orientation : Orientation) =
    member val IsPeeled = false with get, set
    member val Orientation = orientation with get, set
    member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

    member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
    member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on the
exception above.

module BananaHelpers =
    let peel (b: Banana) =
        let flip (banana: Banana) =
            match banana.Orientation with
            | Up ->
                banana.Orientation <- Down
                banana
            | Down -> banana

        let peelSides (banana: Banana) =
            banana.Sides
            |> List.map (function
                         | Unpeeled -> Peeled
                         | Peeled -> Peeled)

        match b.Orientation with
        | Up ->   b |> flip |> peelSides
        | Down -> b |> peelSides
```

Note that the exception `DontSqueezeTheBananaException` and the class `Banana` both refer to each other. Additionally, the module `BananaHelpers` and the class `Banana` also refer to each other. This wouldn't be possible to express in F# if you removed the `rec` keyword from the `MutualReferences` namespace.

This feature is also available for top-level Modules.

See also

- F# Language Reference
- Modules
- F# RFC FS-1009 - Allow mutually referential types and modules over larger scopes within files

Modules

7/30/2019 • 6 minutes to read • Edit Online

In the context of the F# language, a *module* is a grouping of F# code, such as values, types, and function values, in an F# program. Grouping code in modules helps keep related code together and helps avoid name conflicts in your program.

Syntax

```
// Top-level module declaration.
module [accessibility-modifier] [qualified-namespace.]module-name
declarations
// Local module declaration.
module [accessibility-modifier] module-name =
    declarations
```

Remarks

An F# module is a grouping of F# code constructs such as types, values, function values, and code in `do` bindings. It is implemented as a common language runtime (CLR) class that has only static members. There are two types of module declarations, depending on whether the whole file is included in the module: a top-level module declaration and a local module declaration. A top-level module declaration includes the whole file in the module. A top-level module declaration can appear only as the first declaration in a file.

In the syntax for the top-level module declaration, the optional *qualified-namespace* is the sequence of nested namespace names that contains the module. The qualified namespace does not have to be previously declared.

You do not have to indent declarations in a top-level module. You do have to indent all declarations in local modules. In a local module declaration, only the declarations that are indented under that module declaration are part of the module.

If a code file does not begin with a top-level module declaration or a namespace declaration, the whole contents of the file, including any local modules, becomes part of an implicitly created top-level module that has the same name as the file, without the extension, with the first letter converted to uppercase. For example, consider the following file.

```
// In the file program.fs.
let x = 40
```

This file would be compiled as if it were written in this manner:

```
module Program
let x = 40
```

If you have multiple modules in a file, you must use a local module declaration for each module. If an enclosing namespace is declared, these modules are part of the enclosing namespace. If an enclosing namespace is not declared, the modules become part of the implicitly created top-level module. The following code example shows a code file that contains multiple modules. The compiler implicitly creates a top-level module named `Multiplemodules`, and `MyModule1` and `MyModule2` are nested in that top-level module.

```
// In the file multiplemodules.fs.
// MyModule1
module MyModule1 =
    // Indent all program elements within modules that are declared with an equal sign.
    let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

    // Use a qualified name to access the function.
    // from MyModule1.
    let module2Function x =
        x * (MyModule1.module1Function module2Value)
```

If you have multiple files in a project or in a single compilation, or if you are building a library, you must include a namespace declaration or module declaration at the top of the file. The F# compiler only determines a module name implicitly when there is only one file in a project or compilation command line, and you are creating an application.

The *accessibility-modifier* can be one of the following: `public`, `private`, `internal`. For more information, see Access Control. The default is public.

Referencing Code in Modules

When you reference functions, types, and values from another module, you must either use a qualified name or open the module. If you use a qualified name, you must specify the namespaces, the module, and the identifier for the program element you want. You separate each part of the qualified path with a dot (.), as follows.

```
Namespace1.Namespace2.ModuleName.Identifier
```

You can open the module or one or more of the namespaces to simplify the code. For more information about opening namespaces and modules, see Import Declarations: The `open` Keyword.

The following code example shows a top-level module that contains all the code up to the end of the file.

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

To use this code from another file in the same project, you either use qualified names or you open the module before you use the functions, as shown in the following examples.

```
// Fully qualify the function name.
let result1 = Arithmetic.add 5 9
// Open the module.
open Arithmetic
let result2 = add 5 9
```

Nested Modules

Modules can be nested. Inner modules must be indented as far as outer module declarations to indicate that they are inner modules, not new modules. For example, compare the following two examples. Module `Z` is an inner module in the following code.

```
module Y =
    let x = 1

    module Z =
        let z = 5
```

But module `Z` is a sibling to module `Y` in the following code.

```
module Y =
    let x = 1

module Z =
    let z = 5
```

Module `Z` is also a sibling module in the following code, because it is not indented as far as other declarations in module `Y`.

```
module Y =
        let x = 1

    module Z =
        let z = 5
```

Finally, if the outer module has no declarations and is followed immediately by another module declaration, the new module declaration is assumed to be an inner module, but the compiler will warn you if the second module definition is not indented farther than the first.

```
// This code produces a warning, but treats Z as a inner module.
module Y =
module Z =
    let z = 5
```

To eliminate the warning, indent the inner module.

```
module Y =
    module Z =
        let z = 5
```

If you want all the code in a file to be in a single outer module and you want inner modules, the outer module does not require the equal sign, and the declarations, including any inner module declarations, that will go in the outer module do not have to be indented. Declarations inside the inner module declarations do have to be indented. The following code shows this case.

```
// The top-level module declaration can be omitted if the file is named
// TopLevel.fs or topLevel.fs, and the file is the only file in an
// application.
module TopLevel

let topLevelX = 5

module Inner1 =
    let inner1X = 1
module Inner2 =
    let inner2X = 5
```

Recursive modules

F# 4.1 introduces the notion of modules which allow for all contained code to be mutually recursive. This is done via `module rec`. Use of `module rec` can alleviate some pains in not being able to write mutually referential code between types and modules. The following is an example of this:

```
module rec RecursiveModule =
    type Orientation = Up | Down
    type PeelState = Peeled | Unpeeled

    // This exception depends on the type below.
    exception DontSqueezeTheBananaException of Banana

    type BananaPeel() = class end

    type Banana(orientation : Orientation) =
        member val IsPeeled = false with get, set
        member val Orientation = orientation with get, set
        member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

        member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
        member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on
the exception above.

    module BananaHelpers =
        let peel (b: Banana) =
            let flip (banana: Banana) =
                match banana.Orientation with
                | Up ->
                    banana.Orientation <- Down
                    banana
                | Down -> banana

            let peelSides (banana: Banana) =
                banana.Sides
                |> List.map (function
                             | Unpeeled -> Peeled
                             | Peeled -> Peeled)

            match b.Orientation with
            | Up ->   b |> flip |> peelSides
            | Down -> b |> peelSides
```

Note that the exception `DontSqueezeTheBananaException` and the class `Banana` both refer to each other. Additionally, the module `BananaHelpers` and the class `Banana` also refer to each other. This would not be possible to express in F# if you removed the `rec` keyword from the `RecursiveModule` module.

This capability is also possible in Namespaces with F# 4.1.

See also

- F# Language Reference
- Namespaces
- F# RFC FS-1009 - Allow mutually referential types and modules over larger scopes within files

Import Declarations: The `open` Keyword

7/30/2019 • 3 minutes to read • Edit Online

> **NOTE**
> The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

An *import declaration* specifies a module or namespace whose elements you can reference without using a fully qualified name.

Syntax

```
open module-or-namespace-name
```

Remarks

Referencing code by using the fully qualified namespace or module path every time can create code that is hard to write, read, and maintain. Instead, you can use the `open` keyword for frequently used modules and namespaces so that when you reference a member of that module or namespace, you can use the short form of the name instead of the fully qualified name. This keyword is similar to the `using` keyword in C#, `using namespace` in Visual C++, and `Imports` in Visual Basic.

The module or namespace provided must be in the same project or in a referenced project or assembly. If it is not, you can add a reference to the project, or use the `-reference` command `-` line option (or its abbreviation, `-r`). For more information, see Compiler Options.

The import declaration makes the names available in the code that follows the declaration, up to the end of the enclosing namespace, module, or file.

When you use multiple import declarations, they should appear on separate lines.

The following code shows the use of the `open` keyword to simplify code.

```
// Without the import declaration, you must include the full
// path to .NET Framework namespaces such as System.IO.
let writeToFile1 filename (text: string) =
  let stream1 = new System.IO.FileStream(filename, System.IO.FileMode.Create)
  let writer = new System.IO.StreamWriter(stream1)
  writer.WriteLine(text)

// Open a .NET Framework namespace.
open System.IO

// Now you do not have to include the full paths.
let writeToFile2 filename (text: string) =
  let stream1 = new FileStream(filename, FileMode.Create)
  let writer = new StreamWriter(stream1)
  writer.WriteLine(text)

writeToFile2 "file1.txt" "Testing..."
```

The F# compiler does not emit an error or warning when ambiguities occur when the same name occurs in more

than one open module or namespace. When ambiguities occur, F# gives preference to the more recently opened module or namespace. For example, in the following code, `empty` means `Seq.empty`, even though `empty` is located in both the `List` and `Seq` modules.

```
open List
open Seq
printfn "%A" empty
```

Therefore, be careful when you open modules or namespaces such as `List` or `Seq` that contain members that have identical names; instead, consider using the qualified names. You should avoid any situation in which the code is dependent upon the order of the import declarations.

Namespaces That Are Open by Default

Some namespaces are so frequently used in F# code that they are opened implicitly without the need of an explicit import declaration. The following table shows the namespaces that are open by default.

NAMESPACE	DESCRIPTION
`Microsoft.FSharp.Core`	Contains basic F# type definitions for built-in types such as `int` and `float`.
`Microsoft.FSharp.Core.Operators`	Contains basic arithmetic operations such as `+` and `*`.
`Microsoft.FSharp.Collections`	Contains immutable collection classes such as `List` and `Array`.
`Microsoft.FSharp.Control`	Contains types for control constructs such as lazy evaluation and asynchronous workflows.
`Microsoft.FSharp.Text`	Contains functions for formatted IO, such as the `printf` function.

AutoOpen Attribute

You can apply the `AutoOpen` attribute to an assembly if you want to automatically open a namespace or module when the assembly is referenced. You can also apply the `AutoOpen` attribute to a module to automatically open that module when the parent module or namespace is opened. For more information, see Core.AutoOpenAttribute Class.

RequireQualifiedAccess Attribute

Some modules, records, or union types may specify the `RequireQualifiedAccess` attribute. When you reference elements of those modules, records, or unions, you must use a qualified name regardless of whether you include an import declaration. If you use this attribute strategically on types that define commonly used names, you help avoid name collisions and thereby make code more resilient to changes in libraries. For more information, see Core.RequireQualifiedAccessAttribute Class.

See also

- F# Language Reference
- Namespaces
- Modules

Signatures

7/30/2019 • 4 minutes to read • Edit Online

A signature file contains information about the public signatures of a set of F# program elements, such as types, namespaces, and modules. It can be used to specify the accessibility of these program elements.

Remarks

For each F# code file, you can have a *signature file*, which is a file that has the same name as the code file but with the extension .fsi instead of .fs. Signature files can also be added to the compilation command-line if you are using the command line directly. To distinguish between code files and signature files, code files are sometimes referred to as *implementation files*. In a project, the signature file should precede the associated code file.

A signature file describes the namespaces, modules, types, and members in the corresponding implementation file. You use the information in a signature file to specify what parts of the code in the corresponding implementation file can be accessed from code outside the implementation file, and what parts are internal to the implementation file. The namespaces, modules, and types that are included in the signature file must be a subset of the namespaces, modules, and types that are included in the implementation file. With some exceptions noted later in this topic, those language elements that are not listed in the signature file are considered private to the implementation file. If no signature file is found in the project or command line, the default accessibility is used.

For more information about the default accessibility, see Access Control.

In a signature file, you do not repeat the definition of the types and the implementations of each method or function. Instead, you use the signature for each method and function, which acts as a complete specification of the functionality that is implemented by a module or namespace fragment. The syntax for a type signature is the same as that used in abstract method declarations in interfaces and abstract classes, and is also shown by IntelliSense and by the F# interpreter fsi.exe when it displays correctly compiled input.

If there is not enough information in the type signature to indicate whether a type is sealed, or whether it is an interface type, you must add an attribute that indicates the nature of the type to the compiler. Attributes that you use for this purpose are described in the following table.

ATTRIBUTE	DESCRIPTION
`[<Sealed>]`	For a type that has no abstract members, or that should not be extended.
`[<Interface>]`	For a type that is an interface.

The compiler produces an error if the attributes are not consistent between the signature and the declaration in the implementation file.

Use the keyword `val` to create a signature for a value or function value. The keyword `type` introduces a type signature.

You can generate a signature file by using the `--sig` compiler option. Generally, you do not write .fsi files manually. Instead, you generate .fsi files by using the compiler, add them to your project, if you have one, and edit them by removing methods and functions that you do not want to be accessible.

There are several rules for type signatures:

- Type abbreviations in an implementation file must not match a type without an abbreviation in a signature file.
- Records and discriminated unions must expose either all or none of their fields and constructors, and the order in the signature must match the order in the implementation file. Classes can reveal some, all, or none of their fields and methods in the signature.
- Classes and structures that have constructors must expose the declarations of their base classes (the `inherits` declaration). Also, classes and structures that have constructors must expose all of their abstract methods and interface declarations.
- Interface types must reveal all their methods and interfaces.

The rules for value signatures are as follows:

- Modifiers for accessibility (`public` , `internal` , and so on) and the `inline` and `mutable` modifiers in the signature must match those in the implementation.
- The number of generic type parameters (either implicitly inferred or explicitly declared) must match, and the types and type constraints in generic type parameters must match.
- If the `Literal` attribute is used, it must appear in both the signature and the implementation, and the same literal value must be used for both.
- The pattern of parameters (also known as the *arity*) of signatures and implementations must be consistent.
- If parameter names in a signature file differ from the corresponding implementation file, the name in the signature file will be used instead, which may cause issues when debugging or profiling. If you wish to be notified of such mismatches, enable warning 3218 in your project file or when invoking the compiler (see `--warnon` under Compiler Options).

The following code example shows an example of a signature file that has namespace, module, function value, and type signatures together with the appropriate attributes. It also shows the corresponding implementation file.

```
// Module1.fsi

namespace Library1
  module Module1 =
    val function1 : int -> int
    type Type1 =
        new : unit -> Type1
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Sealed>]
    type Type2 =
        new : unit -> Type2
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

The following code shows the implementation file.

```
namespace Library1

module Module1 =

    let function1 x = x + 1


    type Type1() =
        member type1.method1() =
            printfn "type1.method1"
        member type1.method2() =
            printfn "type1.method2"


    [<Sealed>]
    type Type2() =
        member type2.method1() =
            printfn "type2.method1"
        member type2.method2() =
            printfn "type2.method2"

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

See also

- F# Language Reference
- Access Control
- Compiler Options

Units of Measure

7/30/2019 • 7 minutes to read • Edit Online

Floating point and signed integer values in F# can have associated units of measure, which are typically used to indicate length, volume, mass, and so on. By using quantities with units, you enable the compiler to verify that arithmetic relationships have the correct units, which helps prevent programming errors.

Syntax

```
[<Measure>] type unit-name [ = measure ]
```

Remarks

The previous syntax defines *unit-name* as a unit of measure. The optional part is used to define a new measure in terms of previously defined units. For example, the following line defines the measure `cm` (centimeter).

```
[<Measure>] type cm
```

The following line defines the measure `ml` (milliliter) as a cubic centimeter (`cm^3`).

```
[<Measure>] type ml = cm^3
```

In the previous syntax, *measure* is a formula that involves units. In formulas that involve units, integral powers are supported (positive and negative), spaces between units indicate a product of the two units, `*` also indicates a product of units, and `/` indicates a quotient of units. For a reciprocal unit, you can either use a negative integer power or a `/` that indicates a separation between the numerator and denominator of a unit formula. Multiple units in the denominator should be surrounded by parentheses. Units separated by spaces after a `/` are interpreted as being part of the denominator, but any units following a `*` are interpreted as being part of the numerator.

You can use 1 in unit expressions, either alone to indicate a dimensionless quantity, or together with other units, such as in the numerator. For example, the units for a rate would be written as `1/s`, where `s` indicates seconds. Parentheses are not used in unit formulas. You do not specify numeric conversion constants in the unit formulas; however, you can define conversion constants with units separately and use them in unit-checked computations.

Unit formulas that mean the same thing can be written in various equivalent ways. Therefore, the compiler converts unit formulas into a consistent form, which converts negative powers to reciprocals, groups units into a single numerator and a denominator, and alphabetizes the units in the numerator and denominator.

For example, the unit formulas `kg m s^-2` and `m /s s * kg` are both converted to `kg m/s^2`.

You use units of measure in floating point expressions. Using floating point numbers together with associated units of measure adds another level of type safety and helps avoid the unit mismatch errors that can occur in formulas when you use weakly typed floating point numbers. If you write a floating point expression that uses units, the units in the expression must match.

You can annotate literals with a unit formula in angle brackets, as shown in the following examples.

```
1.0<cm>
55.0<miles/hour>
```

You do not put a space between the number and the angle bracket; however, you can include a literal suffix such as `f`, as in the following example.

```
// The f indicates single-precision floating point.
55.0f<miles/hour>
```

Such an annotation changes the type of the literal from its primitive type (such as `float`) to a dimensioned type, such as `float<cm>` or, in this case, `float<miles/hour>`. A unit annotation of `<1>` indicates a dimensionless quantity, and its type is equivalent to the primitive type without a unit parameter.

The type of a unit of measure is a floating point or signed integral type together with an extra unit annotation, indicated in brackets. Thus, when you write the type of a conversion from `g` (grams) to `kg` (kilograms), you describe the types as follows.

```
let convertg2kg (x : float<g>) = x / 1000.0<g/kg>
```

Units of measure are used for compile-time unit checking but are not persisted in the run-time environment. Therefore, they do not affect performance.

Units of measure can be applied to any type, not just floating point types; however, only floating point types, signed integral types, and decimal types support dimensioned quantities. Therefore, it only makes sense to use units of measure on the primitive types and on aggregates that contain these primitive types.

The following example illustrates the use of units of measure.

```
// Mass, grams.
[<Measure>] type g
// Mass, kilograms.
[<Measure>] type kg
// Weight, pounds.
[<Measure>] type lb

// Distance, meters.
[<Measure>] type m
// Distance, cm
[<Measure>] type cm

// Distance, inches.
[<Measure>] type inch
// Distance, feet
[<Measure>] type ft

// Time, seconds.
[<Measure>] type s

// Force, Newtons.
[<Measure>] type N = kg m / s^2

// Pressure, bar.
[<Measure>] type bar
// Pressure, Pascals
[<Measure>] type Pa = N / m^2

// Volume, milliliters.
[<Measure>] type ml
// Volume, liters.
[<Measure>] type L

// Define conversion constants.
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>
let cmPerMeter : float<cm/m> = 100.0<cm/m>
let cmPerInch : float<cm/inch> = 2.54<cm/inch>

let mlPerCubicCentimeter : float<ml/cm^3> = 1.0<ml/cm^3>
let mlPerLiter : float<ml/L> = 1000.0<ml/L>

// Define conversion functions.
let convertGramsToKilograms (x : float<g>) = x / gramsPerKilogram
let convertCentimetersToInches (x : float<cm>) = x / cmPerInch
```

The following code example illustrates how to convert from a dimensionless floating point number to a dimensioned floating point value. You just multiply by 1.0, applying the dimensions to the 1.0. You can abstract this into a function like `degreesFahrenheit`.

Also, when you pass dimensioned values to functions that expect dimensionless floating point numbers, you must cancel out the units or cast to `float` by using the `float` operator. In this example, you divide by `1.0<degC>` for the arguments to `printf` because `printf` expects dimensionless quantities.

```
[<Measure>] type degC // temperature, Celsius/Centigrade
[<Measure>] type degF // temperature, Fahrenheit

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF>
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>)

// Define conversion functions from dimensionless floating point values.
let degreesFahrenheit temp = temp * 1.0<degF>
let degreesCelsius temp = temp * 1.0<degC>

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let parsedOk, floatValue = System.Double.TryParse(input)
if parsedOk
   then
      printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
   else
      printfn "Error parsing input."
```

The following example session shows the outputs from and inputs to this code.

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

Using Generic Units

You can write generic functions that operate on data that has an associated unit of measure. You do this by specifying a type together with a generic unit as a type parameter, as shown in the following code example.

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

let genericSumUnits ( x : float<'u>) (y: float<'u>) = x + y

let v1 = 3.1<m/s>
let v2 = 2.7<m/s>
let x1 = 1.2<m>
let t1 = 1.0<s>

// OK: a function that has unit consistency checking.
let result1 = genericSumUnits v1 v2
// Error reported: mismatched units.
// Uncomment to see error.
// let result2 = genericSumUnits v1 x1
```

Creating Aggregate Types with Generic Units

The following code shows how to create an aggregate type that consists of individual floating point values that have units that are generic. This enables a single type to be created that works with a variety of units. Also, generic units preserve type safety by ensuring that a generic type that has one set of units is a different type than the same generic type with a different set of units. The basis of this technique is that the `Measure` attribute can be applied to the type parameter.

```
 // Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

// Define a vector together with a measure type parameter.
// Note the attribute applied to the type parameter.
type vector3D<[<Measure>] 'u> = { x : float<'u>; y : float<'u>; z : float<'u>}

// Create instances that have two different measures.
// Create a position vector.
let xvec : vector3D<m> = { x = 0.0<m>; y = 0.0<m>; z = 0.0<m> }
// Create a velocity vector.
let v1vec : vector3D<m/s> = { x = 1.0<m/s>; y = -1.0<m/s>; z = 0.0<m/s> }
```

Units at Runtime

Units of measure are used for static type checking. When floating point values are compiled, the units of measure are eliminated, so the units are lost at run time. Therefore, any attempt to implement functionality that depends on checking the units at run time is not possible. For example, implementing a `ToString` function to print out the units is not possible.

Conversions

To convert a type that has units (for example, `float<'u>`) to a type that does not have units, you can use the standard conversion function. For example, you can use `float` to convert to a `float` value that does not have units, as shown in the following code.

```
[<Measure>]
type cm
let length = 12.0<cm>
let x = float length
```

To convert a unitless value to a value that has units, you can multiply by a 1 or 1.0 value that is annotated with the appropriate units. However, for writing interoperability layers, there are also some explicit functions that you can use to convert unitless values to values with units. These are in the Microsoft.FSharp.Core.LanguagePrimitives module. For example, to convert from a unitless `float` to a `float<cm>`, use FloatWithMeasure, as shown in the following code.

```
open Microsoft.FSharp.Core
let height:float<cm> = LanguagePrimitives.FloatWithMeasure x
```

Units of Measure in the F# Core library

A unit library is available in the `FSharp.Data.UnitSystems.SI` namespace. It includes SI units in both their symbol form (like `m` for meter) in the `UnitSymbols` sub-namespace, and their full name (like `meter` for meter) in the `UnitNames` sub-namespace.

See also

- F# Language Reference

XML Documentation

7/30/2019 • 3 minutes to read • Edit Online

You can produce documentation from triple-slash (///) code comments in F#. XML comments can precede declarations in code files (.fs) or signature (.fsi) files.

Generating Documentation from Comments

The support in F# for generating documentation from comments is the same as that in other .NET Framework languages. As in other .NET Framework languages, the -doc compiler option enables you to produce an XML file that contains information that you can convert into documentation by using a tool such as DocFX or Sandcastle. The documentation generated by using tools that are designed for use with assemblies that are written in other .NET Framework languages generally produce a view of the APIs that is based on the compiled form of F# constructs. Unless tools specifically support F#, documentation generated by these tools does not match the F# view of an API.

For more information about how to generate documentation from XML, see XML Documentation Comments (C# Programming Guide).

Recommended Tags

There are two ways to write XML documentation comments. One is to just write the documentation directly in a triple-slash comment, without using XML tags. If you do this, the entire comment text is taken as the summary documentation for the code construct that immediately follows. Use this method when you want to write only a brief summary for each construct. The other method is to use XML tags to provide more structured documentation. The second method enables you to specify separate notes for a short summary, additional remarks, documentation for each parameter and type parameter and exceptions thrown, and a description of the return value. The following table describes XML tags that are recognized in F# XML code comments.

TAG SYNTAX	DESCRIPTION
<c>*text***</c>**	Specifies that *text* is code. This tag can be used by documentation generators to display text in a font that is appropriate for code.
<summary>*text***</summary>**	Specifies that *text* is a brief description of the program element. The description is usually one or two sentences.
<remarks>*text***</remarks>**	Specifies that *text* contains supplementary information about the program element.
<param name="*name***">***description***</param>**	Specifies the name and description for a function or method parameter.
<typeparam name="*name***">***description***</typeparam>**	Specifies the name and description for a type parameter.
<returns>*text***</returns>**	Specifies that *text* describes the return value of a function or method.
<exception cref="*type***">***description***</exception>**	Specifies the type of exception that can be generated and the circumstances under which it is thrown.

TAG SYNTAX	DESCRIPTION
<see cref="*reference***">***text***</see>**	Specifies an inline link to another program element. The *reference* is the name as it appears in the XML documentation file. The *text* is the text shown in the link.
<seealso cref="*reference***"/>**	Specifies a See Also link to the documentation for another type. The *reference* is the name as it appears in the XML documentation file. See Also links usually appear at the bottom of a documentation page.
<para>*text***</para>**	Specifies a paragraph of text. This is used to separate text inside the **remarks** tag.

Example

Description

The following is a typical XML documentation comment in a signature file.

Code

```
/// <summary>Builds a new string whose characters are the results of applying the function <c>mapping</c>
/// to each of the characters of the input string and concatenating the resulting
/// strings.</summary>
/// <param name="mapping">The function to produce a string from each character of the input string.</param>
///<param name="str">The input string.</param>
///<returns>The concatenated string.</returns>
///<exception cref="System.ArgumentNullException">Thrown when the input string is null.</exception>
val collect : (char -> string) -> string -> string
```

Example

Description

The following example shows the alternative method, without XML tags. In this example, the entire text in the comment is considered a summary. Note that if you do not specify a summary tag explicitly, you should not specify other tags, such as **param** or **returns** tags.

Code

```
/// Creates a new string whose characters are the result of applying
/// the function mapping to each of the characters of the input string
/// and concatenating the resulting strings.
val collect : (char -> string) -> string -> string
```

See also

- F# Language Reference
- Compiler Options

Lazy Expressions

7/30/2019 • 2 minutes to read • Edit Online

Lazy expressions are expressions that are not evaluated immediately, but are instead evaluated when the result is needed. This can help to improve the performance of your code.

Syntax

```
let identifier = lazy ( expression )
```

Remarks

In the previous syntax, *expression* is code that is evaluated only when a result is required, and *identifier* is a value that stores the result. The value is of type `Lazy<'T>`, where the actual type that is used for `'T` is determined from the result of the expression.

Lazy expressions enable you to improve performance by restricting the execution of an expressions to only those situations in which a result is needed.

To force the expressions to be performed, you call the method `Force`. `Force` causes the execution to be performed only one time. Subsequent calls to `Force` return the same result, but do not execute any code.

The following code illustrates the use of lazy expressions and the use of `Force`. In this code, the type of `result` is `Lazy<int>`, and the `Force` method returns an `int`.

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

Lazy evaluation, but not the `Lazy` type, is also used for sequences. For more information, see Sequences.

See also

- F# Language Reference
- LazyExtensions module

Computation Expressions

6/17/2019 • 14 minutes to read • Edit Online

Computation expressions in F# provide a convenient syntax for writing computations that can be sequenced and combined using control flow constructs and bindings. Depending on the kind of computation expression, they can be thought of as a way to express monads, monoids, monad transformers, and applicative functors. However, unlike other languages (such as *do-notation* in Haskell), they are not tied to a single abstraction, and do not rely on macros or other forms of metaprogramming to accomplish a convenient and context-sensitive syntax.

Overview

Computations can take many forms. The most common form of computation is single-threaded execution, which is easy to understand and modify. However, not all forms of computation are as straightforward as single-threaded execution. Some examples include:

- Non-deterministic computations
- Asynchronous computations
- Effectful computations
- Generative computations

More generally, there are *context-sensitive* computations that you must perform in certain parts of an application. Writing context-sensitive code can be challenging, as it is easy to "leak" computations outside of a given context without abstractions to prevent you from doing so. These abstractions are often challenging to write by yourself, which is why F# has a generalized way to do so called **computation expressions**.

Computation expressions offer a uniform syntax and abstraction model for encoding context-sensitive computations.

Every computation expression is backed by a *builder* type. The builder type defines the operations that are available for the computation expression. See Creating a New Type of Computation Expression, which shows how to create a custom computation expression.

Syntax overview

All computation expressions have the following form:

```
builder-expr { cexper }
```

where `builder-expr` is the name of a builder type that defines the computation expression, and `cexper` is the expression body of the computation expression. For example, `async` computation expression code can look like this:

```
let fetchAndDownload url =
    async {
        let! data = downloadData url

        let processedData = processData data

        return processedData
    }
```

There is a special, additional syntax available within a computation expression, as shown in the previous example. The following expression forms are possible with computation expressions:

```
expr { let! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }
expr { match! ... }
```

Each of these keywords, and other standard F# keywords are only available in a computation expression if they have been defined in the backing builder type. The only exception to this is `match!`, which is itself syntactic sugar for the use of `let!` followed by a pattern match on the result.

The builder type is an object that defines special methods that govern the way the fragments of the computation expression are combined; that is, its methods control how the computation expression behaves. Another way to describe a builder class is to say that it enables you to customize the operation of many F# constructs, such as loops and bindings.

`let!`

The `let!` keyword binds the result of a call to another computation expression to a name:

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

If you bind the call to a computation expression with `let`, you will not get the result of the computation expression. Instead, you will have bound the value of the *unrealized* call to that computation expression. Use `let!` to bind to the result.

`let!` is defined by the `Bind(x, f)` member on the builder type.

`do!`

The `do!` keyword is for calling a computation expression that returns a `unit`-like type (defined by the `Zero` member on the builder):

```
let doThingsAsync data url =
    async {
        do! submitData data url
        ...
    }
```

For the async workflow, this type is `Async<unit>`. For other computation expressions, the type is likely to be `CExpType<unit>`.

`do!` is defined by the `Bind(x, f)` member on the builder type, where `f` produces a `unit`.

`yield`

The `yield` keyword is for returning a value from the computation expression so that it can be consumed as an IEnumerable<T>:

```
let squares =
    seq {
        for i in 1..10 do
            yield i * i
    }

for sq in squares do
    printfn "%d" sq
```

As with the yield keyword in C#, each element in the computation expression is yielded back as it is iterated.

`yield` is defined by the `Yield(x)` member on the builder type, where `x` is the item to yield back.

`yield!`

The `yield!` keyword is for flattening a collection of values from a computation expression:

```
let squares =
    seq {
        for i in 1..3 -> i * i
    }

let cubes =
    seq {
        for i in 1..3 -> i * i * i
    }

let squaresAndCubes =
    seq {
        yield! squares
        yield! cubes
    }

printfn "%A" squaresAndCubes // Prints - 1; 4; 9; 1; 8; 27
```

When evaluated, the computation expression called by `yield!` will have its items yielded back one-by-one, flattening the result.

`yield!` is defined by the `YieldFrom(x)` member on the builder type, where `x` is a collection of values.

`return`

The `return` keyword wraps a value in the type corresponding to the computation expression. Aside from computation expressions using `yield`, it is used to "complete" a computation expression:

```
let req = // 'req' is of type is 'Async<data>'
    async {
        let! data = fetch url
        return data
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return` is defined by the `Return(x)` member on the builder type, where `x` is the item to wrap.

`return!`

The `return!` keyword realizes the value of a computation expression and wraps that result in the type corresponding to the computation expression:

```
let req = // 'req' is of type is 'Async<data>'
    async {
        return! fetch url
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return!` is defined by the `ReturnFrom(x)` member on the builder type, where `x` is another computation expression.

`match!`

Starting with F# 4.5, the `match!` keyword allows you to inline a call to another computation expression and pattern match on its result:

```
let doThingsAsync url =
    async {
        match! callService url with
        | Some data -> ...
        | None -> ...
    }
```

When calling a computation expression with `match!`, it will realize the result of the call like `let!`. This is often used when calling a computation expression where the result is an optional.

Built-in computation expressions

The F# core library defines three built-in computation expressions: Sequence Expressions, Asynchronous Workflows, and Query Expressions.

Creating a New Type of Computation Expression

You can define the characteristics of your own computation expressions by creating a builder class and defining certain special methods on the class. The builder class can optionally define the methods as listed in the following table.

The following table describes methods that can be used in a workflow builder class.

METHOD	TYPICAL SIGNATURE(S)	DESCRIPTION
`Bind`	`M<'T> * ('T -> M<'U>) -> M<'U>`	Called for `let!` and `do!` in computation expressions.
`Delay`	`(unit -> M<'T>) -> M<'T>`	Wraps a computation expression as a function.
`Return`	`'T -> M<'T>`	Called for `return` in computation expressions.
`ReturnFrom`	`M<'T> -> M<'T>`	Called for `return!` in computation expressions.
`Run`	`M<'T> -> M<'T>` or `M<'T> -> 'T`	Executes a computation expression.

METHOD	TYPICAL SIGNATURE(S)	DESCRIPTION
`Combine`	`M<'T> * M<'T> -> M<'T>` or `M<unit> * M<'T> -> M<'T>`	Called for sequencing in computation expressions.
`For`	`seq<'T> * ('T -> M<'U>) -> M<'U>` or `seq<'T> * ('T -> M<'U>) -> seq<M<'U>>`	Called for `for...do` expressions in computation expressions.
`TryFinally`	`M<'T> * (unit -> unit) -> M<'T>`	Called for `try...finally` expressions in computation expressions.
`TryWith`	`M<'T> * (exn -> M<'T>) -> M<'T>`	Called for `try...with` expressions in computation expressions.
`Using`	`'T * ('T -> M<'U>) -> M<'U> when 'U :> IDisposable`	Called for `use` bindings in computation expressions.
`While`	`(unit -> bool) * M<'T> -> M<'T>`	Called for `while...do` expressions in computation expressions.
`Yield`	`'T -> M<'T>`	Called for `yield` expressions in computation expressions.
`YieldFrom`	`M<'T> -> M<'T>`	Called for `yield!` expressions in computation expressions.
`Zero`	`unit -> M<'T>`	Called for empty `else` branches of `if...then` expressions in computation expressions.
`Quote`	`Quotations.Expr<'T> -> Quotations.Expr<'T>`	Indicates that the computation expression is passed to the `Run` member as a quotation. It translates all instances of a computation into a quotation.

Many of the methods in a builder class use and return an `M<'T>` construct, which is typically a separately defined type that characterizes the kind of computations being combined, for example, `Async<'T>` for asynchronous workflows and `Seq<'T>` for sequence workflows. The signatures of these methods enable them to be combined and nested with each other, so that the workflow object returned from one construct can be passed to the next. The compiler, when it parses a computation expression, converts the expression into a series of nested function calls by using the methods in the preceding table and the code in the computation expression.

The nested expression is of the following form:

```
builder.Run(builder.Delay(fun () -> {| cexpr |}))
```

In the above code, the calls to `Run` and `Delay` are omitted if they are not defined in the computation expression builder class. The body of the computation expression, here denoted as `{| cexpr |}`, is translated

into calls involving the methods of the builder class by the translations described in the following table. The computation expression `{| cexpr |}` is defined recursively according to these translations where `expr` is an F# expression and `cexpr` is a computation expression.

EXPRESSION	TRANSLATION
`{ let binding in cexpr }`	`let binding in {| cexpr |}`
`{ let! pattern = expr in cexpr }`	`builder.Bind(expr, (fun pattern -> {| cexpr |}))`
`{ do! expr in cexpr }`	`builder.Bind(expr, (fun () -> {| cexpr |}))`
`{ yield expr }`	`builder.Yield(expr)`
`{ yield! expr }`	`builder.YieldFrom(expr)`
`{ return expr }`	`builder.Return(expr)`
`{ return! expr }`	`builder.ReturnFrom(expr)`
`{ use pattern = expr in cexpr }`	`builder.Using(expr, (fun pattern -> {| cexpr |}))`
`{ use! value = expr in cexpr }`	`builder.Bind(expr, (fun value -> builder.Using(value, (fun value -> { cexpr }))))`
`{ if expr then cexpr0 |}`	`if expr then { cexpr0 } else binder.Zero()`
`{ if expr then cexpr0 else cexpr1 |}`	`if expr then { cexpr0 } else { cexpr1 }`
`{ match expr with | pattern_i -> cexpr_i }`	`match expr with | pattern_i -> { cexpr_i }`
`{ for pattern in expr do cexpr }`	`builder.For(enumeration, (fun pattern -> { cexpr }))`
`{ for identifier = expr1 to expr2 do cexpr }`	`builder.For(enumeration, (fun identifier -> { cexpr }))`
`{ while expr do cexpr }`	`builder.While(fun () -> expr, builder.Delay({ cexpr }))`
`{ try cexpr with | pattern_i -> expr_i }`	`builder.TryWith(builder.Delay({ cexpr }), (fun value -> match value with | pattern_i -> expr_i | exn -> reraise exn)))`
`{ try cexpr finally expr }`	`builder.TryFinally(builder.Delay( { cexpr }), (fun () -> expr))`
`{ cexpr1; cexpr2 }`	`builder.Combine({ cexpr1 }, { cexpr2 })`
`{ other-expr; cexpr }`	`expr; { cexpr }`
`{ other-expr }`	`expr; builder.Zero()`

In the previous table, `other-expr` describes an expression that is not otherwise listed in the table. A builder

class does not need to implement all of the methods and support all of the translations listed in the previous table. Those constructs that are not implemented are not available in computation expressions of that type. For example, if you do not want to support the `use` keyword in your computation expressions, you can omit the definition of `Use` in your builder class.

The following code example shows a computation expression that encapsulates a computation as a series of steps that can be evaluated one step at a time. A discriminated union type, `OkOrException`, encodes the error state of the expression as evaluated so far. This code demonstrates several typical patterns that you can use in your computation expressions, such as boilerplate implementations of some of the builder methods.

```
// Computations that can be run step by step
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Exception exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Exception exn -> result (Exception exn))

    // The delay operator.
    let delay func = NotYetDone (fun () -> func())

    // The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    // The rest of the operations are boilerplate.
    // The tryFinally operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Exception exn -> raise exn)

    // The tryWith operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
```

```
        |> bind (function Ok value -> result value | Exception exn -> handler exn)

    // The whileLoop operator.
    // This is boilerplate in terms of "result" and "bind".
    let rec whileLoop pred body =
        if pred() then body |> bind (fun _ -> whileLoop pred body)
        else result ()

    // The sequential composition operator.
    // This is boilerplate in terms of "result" and "bind".
    let combine expr1 expr2 =
        expr1 |> bind (fun () -> expr2)

    // The using operator.
    let using (resource: #System.IDisposable) func =
        tryFinally (func resource) (fun () -> resource.Dispose())

    // The forLoop operator.
    // This is boilerplate in terms of "catch", "result", and "bind".
    let forLoop (collection:seq<_>) func =
        let ie = collection.GetEnumerator()
        tryFinally
            (whileLoop
                (fun () -> ie.MoveNext())
                (delay (fun () -> let value = ie.Current in func value)))
            (fun () -> ie.Dispose())

// The builder class.
type EventuallyBuilder() =
    member x.Bind(comp, func) = Eventually.bind func comp
    member x.Return(value) = Eventually.result value
    member x.ReturnFrom(value) = value
    member x.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member x.Delay(func) = Eventually.delay func
    member x.Zero() = Eventually.result ()
    member x.TryWith(expr, handler) = Eventually.tryWith expr handler
    member x.TryFinally(expr, compensation) = Eventually.tryFinally expr compensation
    member x.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member x.Using(resource, expr) = Eventually.using resource expr

let eventually = new EventuallyBuilder()

let comp = eventually {
    for x in 1..2 do
        printfn " x = %d" x
    return 3 + 4 }

// Try the remaining lines in F# interactive to see how this
// computation expression works in practice.
let step x = Eventually.step x

// returns "NotYetDone <closure>"
comp |> step

// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step

// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step
```

A computation expression has an underlying type, which the expression returns. The underlying type may represent a computed result or a delayed computation that can be performed, or it may provide a way to iterate through some type of collection. In the previous example, the underlying type was **Eventually**. For a sequence

expression, the underlying type is System.Collections.Generic.IEnumerable<T>. For a query expression, the underlying type is System.Linq.IQueryable. For an asynchronous workflow, the underlying type is `Async`. The `Async` object represents the work to be performed to compute the result. For example, you call `Async.RunSynchronously` to execute a computation and return the result.

Custom Operations

You can define a custom operation on a computation expression and use a custom operation as an operator in a computation expression. For example, you can include a query operator in a query expression. When you define a custom operation, you must define the Yield and For methods in the computation expression. To define a custom operation, put it in a builder class for the computation expression, and then apply the `CustomOperationAttribute`. This attribute takes a string as an argument, which is the name to be used in a custom operation. This name comes into scope at the start of the opening curly brace of the computation expression. Therefore, you shouldn't use identifiers that have the same name as a custom operation in this block. For example, avoid the use of identifiers such as `all` or `last` in query expressions.

Extending existing Builders with new Custom Operations

If you already have a builder class, its custom operations can be extended from outside of this builder class. Extensions must be declared in modules. Namespaces cannot contain extension members except in the same file and the same namespace declaration group where the type is defined.

The following example shows the extension of the existing `Microsoft.FSharp.Linq.QueryBuilder` class.

```
type Microsoft.FSharp.Linq.QueryBuilder with

    [<CustomOperation("existsNot")>]
    member __.ExistsNot (source: QuerySource<'T, 'Q>, predicate) =
        Enumerable.Any (source.Source, Func<_,_>(predicate)) |> not
```

See also

- F# Language Reference
- Asynchronous Workflows
- Sequences
- Query Expressions

Asynchronous Workflows

7/30/2019 • 4 minutes to read • Edit Online

> **NOTE**
> The API reference link will take you to MSDN. The docs.microsoft.com API reference is not complete.

This topic describes support in F# for performing computations asynchronously, that is, without blocking execution of other work. For example, asynchronous computations can be used to write applications that have UIs that remain responsive to users as the application performs other work.

Syntax

```
async { expression }
```

Remarks

In the previous syntax, the computation represented by `expression` is set up to run asynchronously, that is, without blocking the current computation thread when asynchronous sleep operations, I/O, and other asynchronous operations are performed. Asynchronous computations are often started on a background thread while execution continues on the current thread. The type of the expression is `Async<'T>`, where `'T` is the type returned by the expression when the `return` keyword is used. The code in such an expression is referred to as an *asynchronous block*, or *async block*.

There are a variety of ways of programming asynchronously, and the `Async` class provides methods that support several scenarios. The general approach is to create `Async` objects that represent the computation or computations that you want to run asynchronously, and then start these computations by using one of the triggering functions. The various triggering functions provide different ways of running asynchronous computations, and which one you use depends on whether you want to use the current thread, a background thread, or a .NET Framework task object, and whether there are continuation functions that should run when the computation finishes. For example, to start an asynchronous computation on the current thread, you can use `Async.StartImmediate`. When you start an asynchronous computation from the UI thread, you do not block the main event loop that processes user actions such as keystrokes and mouse activity, so your application remains responsive.

Asynchronous Binding by Using let!

In an asynchronous workflow, some expressions and operations are synchronous, and some are longer computations that are designed to return a result asynchronously. When you call a method asynchronously, instead of an ordinary `let` binding, you use `let!`. The effect of `let!` is to enable execution to continue on other computations or threads as the computation is being performed. After the right side of the `let!` binding returns, the rest of the asynchronous workflow resumes execution.

The following code shows the difference between `let` and `let!`. The line of code that uses `let` just creates an asynchronous computation as an object that you can run later by using, for example, `Async.StartImmediate` or `Async.RunSynchronously`. The line of code that uses `let!` starts the computation, and then the thread is suspended until the result is available, at which point execution continues.

```
// let just stores the result as an asynchronous operation.
let (result1 : Async<byte[]>) = stream.AsyncRead(bufferSize)
// let! completes the asynchronous operation and returns the data.
let! (result2 : byte[])  = stream.AsyncRead(bufferSize)
```

In addition to `let!`, you can use `use!` to perform asynchronous bindings. The difference between `let!` and `use!` is the same as the difference between `let` and `use`. For `use!`, the object is disposed of at the close of the current scope. Note that in the current release of the F# language, `use!` does not allow a value to be initialized to null, even though `use` does.

Asynchronous Primitives

A method that performs a single asynchronous task and returns the result is called an *asynchronous primitive*, and these are designed specifically for use with `let!`. Several asynchronous primitives are defined in the F# core library. Two such methods for Web applications are defined in the module `Microsoft.FSharp.Control.WebExtensions`: `WebRequest.AsyncGetResponse` and `WebClient.AsyncDownloadString`. Both primitives download data from a Web page, given a URL. `AsyncGetResponse` produces a `System.Net.WebResponse` object, and `AsyncDownloadString` produces a string that represents the HTML for a Web page.

Several primitives for asynchronous I/O operations are included in the `Microsoft.FSharp.Control.CommonExtensions` module. These extension methods of the `System.IO.Stream` class are `Stream.AsyncRead` and `Stream.AsyncWrite`.

You can also write your own asynchronous primitives by defining a function whose complete body is enclosed in an async block.

To use asynchronous methods in the .NET Framework that are designed for other asynchronous models with the F# asynchronous programming model, you create a function that returns an F# `Async` object. The F# library has functions that make this easy to do.

One example of using asynchronous workflows is included here; there are many others in the documentation for the methods of the Async class.

This example shows how to use asynchronous workflows to perform computations in parallel.

In the following code example, a function `fetchAsync` gets the HTML text returned from a Web request. The `fetchAsync` function contains an asynchronous block of code. When a binding is made to the result of an asynchronous primitive, in this case `AsyncDownloadString`, let! is used instead of let.

You use the function `Async.RunSynchronously` to execute an asynchronous operation and wait for its result. As an example, you can execute multiple asynchronous operations in parallel by using the `Async.Parallel` function together with the `Async.RunSynchronously` function. The `Async.Parallel` function takes a list of the `Async` objects, sets up the code for each `Async` task object to run in parallel, and returns an `Async` object that represents the parallel computation. Just as for a single operation, you call `Async.RunSynchronously` to start the execution.

The `runAll` function launches three asynchronous workflows in parallel and waits until they have all completed.

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

See also

- F# Language Reference
- Computation Expressions
- Control.Async Class

Query Expressions

5/15/2019 • 41 minutes to read • Edit Online

NOTE

The API reference links in this article will take you to MSDN. The docs.microsoft.com API reference is not complete.

Query expressions enable you to query a data source and put the data in a desired form. Query expressions provide support for LINQ in F#.

Syntax

```
query { expression }
```

Remarks

Query expressions are a type of computation expression similar to sequence expressions. Just as you specify a sequence by providing code in a sequence expression, you specify a set of data by providing code in a query expression. In a sequence expression, the `yield` keyword identifies data to be returned as part of the resulting sequence. In query expressions, the `select` keyword performs the same function. In addition to the `select` keyword, F# also supports a number of query operators that are much like the parts of a SQL SELECT statement. Here is an example of a simple query expression, along with code that connects to the Northwind OData source.

```
// Use the OData type provider to create types that can be used to access the Northwind database.
// Add References to FSharp.Data.TypeProviders and System.Data.Services.Client
open Microsoft.FSharp.Data.TypeProviders

type Northwind = ODataService<"http://services.odata.org/Northwind/Northwind.svc">
let db = Northwind.GetDataContext()

// A query expression.
let query1 =
    query {
        for customer in db.Customers do
            select customer
    }

// Print results
query1
|> Seq.iter (fun customer -> printfn "Company: %s Contact: %s" customer.CompanyName customer.ContactName)
```

In the previous code example, the query expression is in curly braces. The meaning of the code in the expression is, return every customer in the Customers table in the database in the query results. Query expressions return a type that implements IQueryable<T> and IEnumerable<T>, and so they can be iterated using the Seq module as the example shows.

Every computation expression type is built from a builder class. The builder class for the query computation expression is `QueryBuilder`. For more information, see Computation Expressions and Linq.QueryBuilder Class.

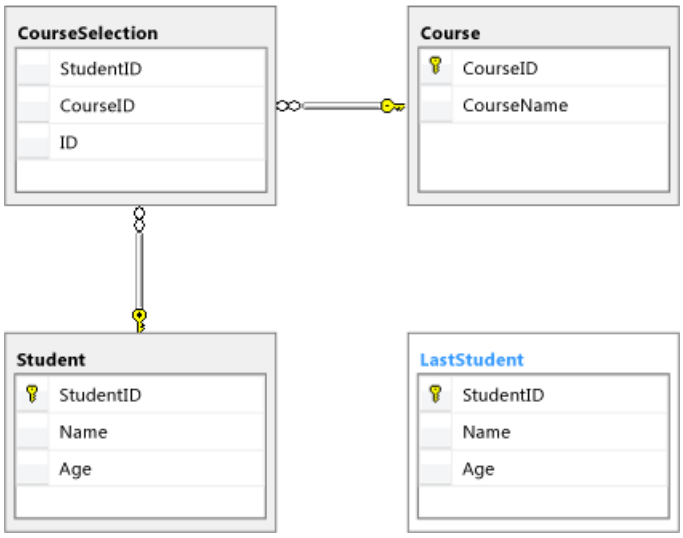
Query Operators

Query operators enable you to specify the details of the query, such as to put criteria on records to be returned, or specify the sorting order of results. The query source must support the query operator. If you attempt to use an unsupported query operator, `System.NotSupportedException` will be thrown.

Only expressions that can be translated to SQL are allowed in query expressions. For example, no function calls are allowed in the expressions when you use the `where` query operator.

Table 1 shows available query operators. In addition, see Table2, which compares SQL queries and the equivalent F# query expressions later in this topic. Some query operators aren't supported by some type providers. In particular, the OData type provider is limited in the query operators that it supports due to limitations in OData. For more information, see ODataService Type Provider (F#).

This table assumes a database in the following form:



The code in the tables that follow also assumes the following database connection code. Projects should add references to System.Data, System.Data.Linq, and FSharp.Data.TypeProviders assemblies. The code that creates this database is included at the end of this topic.

```
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq
open Microsoft.FSharp.Linq

type schema = SqlDataConnection< @"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;" >

let db = schema.GetDataContext()

// Needed for some query operator examples:
let data = [ 1; 5; 7; 11; 18; 21]
```

Table 1. Query Operators

OPERATOR	DESCRIPTION

`contains`	Determines whether the selected elements include a specified element.
	
	`query {`
	`  for student in db.Student do`
	`  select student.Age.Value`
	`  contains 11`
	`}`

`count`	Returns the number of selected elements.
	
	`query {`
	`  for student in db.Student do`
	`  select student`
	`  count`
	`}`

`last`	Selects the last element of those selected so far.
	
	`query {`
	`  for number in data do`
	`  last`
	`}`

`lastOrDefault`	Selects the last element of those selected so far, or a default value if no element is found.
	
	`query {`
	`  for number in data do`
	`  where (number < 0)`
	`  lastOrDefault`
	`}`

`exactlyOne`	Selects the single, specific element selected so far. If multiple elements are present, an exception is thrown.
	
	`query {`
	`  for student in db.Student do`
	`  where (student.StudentID = 1)`
	`  select student`
	`  exactlyOne`
	`}`

`exactlyOneOrDefault`	Selects the single, specific element of those selected so far, or a default value if that element is not found. `query { for student in db.Student do where (student.StudentID = 1) select student exactlyOneOrDefault }`
`headOrDefault`	Selects the first element of those selected so far, or a default value if the sequence contains no elements. `query { for student in db.Student do select student headOrDefault }`
`select`	Projects each of the elements selected so far. `query { for student in db.Student do select student }`
`where`	Selects elements based on a specified predicate. `query { for student in db.Student do where (student.StudentID > 4) select student }`
`minBy`	Selects a value for each element selected so far and returns the minimum resulting value. `query { for student in db.Student do minBy student.StudentID }`

`maxBy`	Selects a value for each element selected so far and returns the maximum resulting value. `query { for student in db.Student do maxBy student.StudentID }`
`groupBy`	Groups the elements selected so far according to a specified key selector. `query { for student in db.Student do groupBy student.Age into g select (g.Key, g.Count()) }`
`sortBy`	Sorts the elements selected so far in ascending order by the given sorting key. `query { for student in db.Student do sortBy student.Name select student }`
`sortByDescending`	Sorts the elements selected so far in descending order by the given sorting key. `query { for student in db.Student do sortByDescending student.Name select student }`
`thenBy`	Performs a subsequent ordering of the elements selected so far in ascending order by the given sorting key. This operator may only be used after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`. `query { for student in db.Student do where student.Age.HasValue sortBy student.Age.Value thenBy student.Name select student }`

`thenByDescending`	Performs a subsequent ordering of the elements selected so far in descending order by the given sorting key. This operator may only be used after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`.
	
	`query {`
	`  for student in db.Student do`
	`  where student.Age.HasValue`
	`  sortBy student.Age.Value`
	`  thenByDescending student.Name`
	`  select student`
	`}`

`groupValBy`	Selects a value for each element selected so far and groups the elements by the given key.
	
	`query {`
	`  for student in db.Student do`
	`  groupValBy student.Name student.Age into g`
	`  select (g, g.Key, g.Count())`
	`}`

`join`	Correlates two sets of selected values based on matching keys. Note that the order of the keys around the = sign in a join expression is significant. In all joins, if the line is split after the `->` symbol, the indentation must be indented at least as far as the keyword `for`.
	
	`query {`
	`  for student in db.Student do`
	`  join selection in db.CourseSelection`
	`  on (student.StudentID =`
	`selection.StudentID)`
	`  select (student, selection)`
	`}`

`groupJoin`	Correlates two sets of selected values based on matching keys and groups the results. Note that the order of the keys around the = sign in a join expression is significant.
	
	`query {`
	`  for student in db.Student do`
	`  groupJoin courseSelection in`
	`db.CourseSelection`
	`  on (student.StudentID =`
	`courseSelection.StudentID) into g`
	`  for courseSelection in g do`
	`  join course in db.Course`
	`  on (courseSelection.CourseID =`
	`course.CourseID)`
	`  select (student.Name, course.CourseName)`
	`}`

`leftOuterJoin`	Correlates two sets of selected values based on matching keys and groups the results. If any group is empty, a group with a single default value is used instead. Note that the order of the keys around the = sign in a join expression is significant. `query { for student in db.Student do leftOuterJoin selection in db.CourseSelection on (student.StudentID = selection.StudentID) into result for selection in result.DefaultIfEmpty() do select (student, selection) }`
`sumByNullable`	Selects a nullable value for each element selected so far and returns the sum of these values. If any nullable does not have a value, it is ignored. `query { for student in db.Student do sumByNullable student.Age }`
`minByNullable`	Selects a nullable value for each element selected so far and returns the minimum of these values. If any nullable does not have a value, it is ignored. `query { for student in db.Student do minByNullable student.Age }`
`maxByNullable`	Selects a nullable value for each element selected so far and returns the maximum of these values. If any nullable does not have a value, it is ignored. `query { for student in db.Student do maxByNullable student.Age }`

`averageByNullable`	Selects a nullable value for each element selected so far and returns the average of these values. If any nullable does not have a value, it is ignored. `query { for student in db.Student do averageByNullable (Nullable.float student.Age) }`
`averageBy`	Selects a value for each element selected so far and returns the average of these values. `query { for student in db.Student do averageBy (float student.StudentID) }`
`distinct`	Selects distinct elements from the elements selected so far. `query { for student in db.Student do join selection in db.CourseSelection on (student.StudentID = selection.StudentID) distinct }`
`exists`	Determines whether any element selected so far satisfies a condition. `query { for student in db.Student do where (query { for courseSelection in db.CourseSelection do exists (courseSelection.StudentID = student.StudentID) }) select student }`
`find`	Selects the first element selected so far that satisfies a specified condition. `query { for student in db.Student do find (student.Name = "Abercrombie, Kim") }`

`all`	Determines whether all elements selected so far satisfy a condition. `query { for student in db.Student do all (SqlMethods.Like(student.Name, "%,%")) }`
`head`	Selects the first element from those selected so far. `query { for student in db.Student do head }`
`nth`	Selects the element at a specified index amongst those selected so far. `query { for numbers in data do nth 3 }`
`skip`	Bypasses a specified number of the elements selected so far and then selects the remaining elements. `query { for student in db.Student do skip 1 }`
`skipWhile`	Bypasses elements in a sequence as long as a specified condition is true and then selects the remaining elements. `query { for number in data do skipWhile (number < 3) select student }`
`sumBy`	Selects a value for each element selected so far and returns the sum of these values. `query { for student in db.Student do sumBy student.StudentID }`

`take`	Selects a specified number of contiguous elements from those selected so far.
	
	`query {`
	`  for student in db.Student do`
	`  select student`
	`  take 2`
	`}`

`takeWhile`	Selects elements from a sequence as long as a specified condition is true, and then skips the remaining elements.
	
	`query {`
	`  for number in data do`
	`  takeWhile (number < 10)`
	`}`

`sortByNullable`	Sorts the elements selected so far in ascending order by the given nullable sorting key.
	
	`query {`
	`  for student in db.Student do`
	`  sortByNullable student.Age`
	`  select student`
	`}`

`sortByNullableDescending`	Sorts the elements selected so far in descending order by the given nullable sorting key.
	
	`query {`
	`  for student in db.Student do`
	`  sortByNullableDescending student.Age`
	`  select student`
	`}`

`thenByNullable`	Performs a subsequent ordering of the elements selected so far in ascending order by the given nullable sorting key. This operator may only be used immediately after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`, or their nullable variants.
	
	`query {`
	`  for student in db.Student do`
	`  sortBy student.Name`
	`  thenByNullable student.Age`
	`  select student`
	`}`

`thenByNullableDescending`	Performs a subsequent ordering of the elements selected so far in descending order by the given nullable sorting key. This operator may only be used immediately after a `sortBy`, `sortByDescending`, `thenBy`, or `thenByDescending`, or their nullable variants. `query { for student in db.Student do sortBy student.Name thenByNullableDescending student.Age select student }`

Comparison of Transact-SQL and F# Query Expressions

The following table shows some common Transact-SQL queries and their equivalents in F#. The code in this table also assumes the same database as the previous table and the same initial code to set up the type provider.

Table 2. Transact-SQL and F# Query Expressions

TRANSACT-SQL (NOT CASE SENSITIVE)	F# QUERY EXPRESSION (CASE SENSITIVE)

Select all fields from table.
 	
`SELECT * FROM Student`
	`// All students.`
	`query {`
	`  for student in db.Student do`
	`  select student`
	`}`

Count records in a table.
 	
`SELECT COUNT( * ) FROM Student`
	`// Count of students.`
	`query {`
	`  for student in db.Student do`
	`  count`
	`}`

`EXISTS`
 	
`SELECT * FROM Student`
`WHERE EXISTS`
`  (SELECT * FROM CourseSelection`
`  WHERE CourseSelection.StudentID = Student.StudentID)`
	`// Find students who have signed up at least one course.`
	`query {`
	`  for student in db.Student do`
	`  where`
	`  (query {`
	`  for courseSelection in db.CourseSelection do`
	`  exists (courseSelection.StudentID = student.StudentID) })`
	`  select student`
	`}`

Grouping

```
SELECT Student.Age, COUNT( * ) FROM Student
GROUP BY Student.Age
```

```
// Group by age and count.
query {
    for n in db.Student do
    groupBy n.Age into g
    select (g.Key, g.Count())
}
// OR
query {
    for n in db.Student do
    groupValBy n.Age n.Age into g
    select (g.Key, g.Count())
}
```

Grouping with condition.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING student.Age > 10
```

```
// Group students by age where age > 10.
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Key.HasValue && g.Key.Value > 10)
    select (g.Key, g.Count())
}
```

Grouping with count condition.

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
```

```
// Group students by age and count number of
students
// at each age with more than 1 student.
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
```

Grouping, counting, and summing.

```
SELECT Student.Age, COUNT( * ), SUM(Student.Age)
as total
FROM Student
GROUP BY Student.Age
```

```
// Group students by age and sum ages.
query {
    for student in db.Student do
    groupBy student.Age into g
    let total =
        query {
            for student in g do
            sumByNullable student.Age
        }
    select (g.Key, g.Count(), total)
}
```

Grouping, counting, and ordering by count.

```
SELECT Student.Age, COUNT( * ) as myCount
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
ORDER BY COUNT( * ) DESC
```

```
// Group students by age, count number of
students
// at each age, and display all with count > 1
// in descending order of count.
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
```

`IN` a set of specified values

```
SELECT *
FROM Student
WHERE Student.StudentID IN (1, 2, 5, 10)
```

```
// Select students where studentID is one of a
given list.
let idQuery =
    query {
        for id in [1; 2; 5; 10] do
        select id
    }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
```

`LIKE` and `TOP`.

```
-- '_e%' matches strings where the second
character is 'e'
SELECT TOP 2 * FROM Student
WHERE Student.Name LIKE '_e%'
```

```
// Look for students with Name match _e% pattern
and take first two.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%")
)
    select student
    take 2
}
```

`LIKE` with pattern match set.

```
-- '[abc]%' matches strings where the first
character is
-- 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[abc]%'
```

```
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[abc]%") )
    select student
}
```

`LIKE` with set exclusion pattern.

```
-- '[^abc]%' matches strings where the first
character is
-- not 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
// Look for students with name matching [^abc]%%
pattern.
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "
[^abc]%") )
    select student
}
```

`LIKE` on one field, but select a different field.

```
SELECT StudentID AS ID FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%") )
    select n.StudentID
}
```

`LIKE`, with substring search.

```
SELECT * FROM Student
WHERE Student.Name like '%A%'
```

```
// Using Contains as a query filter.
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
```

Simple `JOIN` with two tables.

```
SELECT * FROM Student
JOIN CourseSelection
ON Student.StudentID = CourseSelection.StudentID
```

```
// Join Student and CourseSelection tables.
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID =
selection.StudentID)
    select (student, selection)
}
```

`LEFT JOIN` with two tables.

```
SELECT * FROM Student
LEFT JOIN CourseSelection
ON Student.StudentID = CourseSelection.StudentID
```

```
//Left Join Student and CourseSelection tables.
query {
    for student in db.Student do
    leftOuterJoin selection in
db.CourseSelection
        on (student.StudentID =
selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
```

`JOIN` with `COUNT`

```
SELECT COUNT( * ) FROM Student
JOIN CourseSelection
ON Student.StudentID = CourseSelection.StudentID
```

```
// Join with count.
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
```

`DISTINCT`

```
SELECT DISTINCT StudentID FROM CourseSelection
```

```
// Join with distinct.
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID =
selection.StudentID)
    distinct
}
```

Distinct count.

```
SELECT DISTINCT COUNT(StudentID) FROM
CourseSelection
```

```
// Join with distinct and count.
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
```

`BETWEEN`

```
SELECT * FROM Student
WHERE Student.Age BETWEEN 10 AND 15
```

```
// Selecting students with ages between 10 and
15.
query {
    for student in db.Student do
    where (student.Age ?>= 10 && student.Age ?<
15)
    select student
}
```

`OR`

```
SELECT * FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
```

```
// Selecting students with age that's either 11
or 12.
query {
    for student in db.Student do
    where (student.Age.Value = 11 ||
student.Age.Value = 12)
    select student
}
```

`OR` with ordering

```
SELECT * FROM Student
WHERE Student.Age = 12 OR Student.Age = 13
ORDER BY Student.Age DESC
```

```
// Selecting students in a certain age range and
sorting.
query {
    for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
```

`TOP`, `OR`, and ordering.

```
SELECT TOP 2 student.Name FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
ORDER BY Student.Name DESC
```

```
// Selecting students with certain ages,
// taking account of the possibility of nulls.
query {
    for student in db.Student do
    where
        ((student.Age.HasValue &&
student.Age.Value = 11) ||
         (student.Age.HasValue &&
student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
```

`UNION` of two queries.

```
SELECT * FROM Student
UNION
SELECT * FROM lastStudent
```

```
let query1 =
    query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

let query2 =
    query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

query2.Union (query1)
```

Intersection of two queries.

```
SELECT * FROM Student
INTERSECT
SELECT * FROM LastStudent
```

```
let query1 =
    query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

let query2 =
    query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

query1.Intersect(query2)
```

`CASE` condition.

```
SELECT student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special value.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Multiple cases.

```
SELECT Student.StudentID,
CASE Student.Age
  WHEN -1 THEN 100
  WHEN 0 THEN 1000
  ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special values.
query {
    for student in db.Student do
    select
        (if student.Age.HasValue &&
student.Age.Value = -1 then
             (student.StudentID,
System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue &&
student.Age.Value = 0 then
             (student.StudentID,
System.Nullable<int>(1000), student.Age)
         else (student.StudentID, student.Age,
student.Age))
}
```

Multiple tables.

```
SELECT * FROM Student, Course
```

```
// Multiple table select.
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
```

Multiple joins.

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN CourseSelection
ON CourseSelection.StudentID = Student.StudentID
JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Multiple joins.
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID =
course.CourseID)
    select (student.Name, course.CourseName)
}
```

Multiple left outer joins.

```
SELECT Student.Name, Course.CourseName
FROM Student
LEFT OUTER JOIN CourseSelection
ON CourseSelection.StudentID = Student.StudentID
LEFT OUTER JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Using leftOuterJoin with multiple joins.
query {
    for student in db.Student do
    leftOuterJoin courseSelection in
db.CourseSelection
        on (student.StudentID =
courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty()
do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID =
course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
```

The following code can be used to create the sample database for these examples.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

USE [master];
GO

IF EXISTS (SELECT * FROM sys.databases WHERE name = 'MyDatabase')
DROP DATABASE MyDatabase;
GO

-- Create the MyDatabase database.
CREATE DATABASE MyDatabase COLLATE SQL_Latin1_General_CP1_CI_AS;
GO

-- Specify a simple recovery model
-- to keep the log growth to a minimum.
ALTER DATABASE MyDatabase
SET RECOVERY SIMPLE;
GO

USE MyDatabase;
```

```
GO

CREATE TABLE [dbo].[Course] (
[CourseID]   INT           NOT NULL,
[CourseName] NVARCHAR (50) NOT NULL,
PRIMARY KEY CLUSTERED ([CourseID] ASC)
);

CREATE TABLE [dbo].[Student] (
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

CREATE TABLE [dbo].[CourseSelection] (
[ID]        INT NOT NULL,
[StudentID] INT NOT NULL,
[CourseID]  INT NOT NULL,
PRIMARY KEY CLUSTERED ([ID] ASC),
CONSTRAINT [FK_CourseSelection_ToTable] FOREIGN KEY ([StudentID]) REFERENCES [dbo].[Student] ([StudentID]) ON
DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT [FK_CourseSelection_Course_1] FOREIGN KEY ([CourseID]) REFERENCES [dbo].[Course] ([CourseID]) ON
DELETE NO ACTION ON UPDATE NO ACTION
);

CREATE TABLE [dbo].[LastStudent] (
[StudentID] INT           NOT NULL,
[Name]      NVARCHAR (50) NOT NULL,
[Age]       INT           NULL,
PRIMARY KEY CLUSTERED ([StudentID] ASC)
);

-- Insert data into the tables.
USE MyDatabase
INSERT INTO Course (CourseID, CourseName)
VALUES(1, 'Algebra I');
INSERT INTO Course (CourseID, CourseName)
VALUES(2, 'Trigonometry');
INSERT INTO Course (CourseID, CourseName)
VALUES(3, 'Algebra II');
INSERT INTO Course (CourseID, CourseName)
VALUES(4, 'History');
INSERT INTO Course (CourseID, CourseName)
VALUES(5, 'English');
INSERT INTO Course (CourseID, CourseName)
VALUES(6, 'French');
INSERT INTO Course (CourseID, CourseName)
VALUES(7, 'Chinese');

INSERT INTO Student (StudentID, Name, Age)
VALUES(1, 'Abercrombie, Kim', 10);
INSERT INTO Student (StudentID, Name, Age)
VALUES(2, 'Abolrous, Hazen', 14);
INSERT INTO Student (StudentID, Name, Age)
VALUES(3, 'Hance, Jim', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(4, 'Adams, Terry', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(5, 'Hansen, Claus', 11);
INSERT INTO Student (StudentID, Name, Age)
VALUES(6, 'Penor, Lori', 13);
INSERT INTO Student (StudentID, Name, Age)
VALUES(7, 'Perham, Tom', 12);
INSERT INTO Student (StudentID, Name, Age)
VALUES(8, 'Peng, Yun-Feng', NULL);

INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(1, 1, 2);
```

```
VALUES(1, 1, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(2, 1, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(3, 1, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(4, 2, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(5, 2, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(6, 2, 6);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(7, 2, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(8, 3, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(9, 3, 1);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(10, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(11, 4, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(12, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(13, 5, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(14, 5, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(15, 7, 3);
```

The following code contains the sample code that appears in this topic.

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema = SqlDataConnection<"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = schema.GetDataContext()

let data = [1; 5; 7; 11; 18; 21]

type Nullable<'T when 'T : ( new : unit -> 'T) and 'T : struct and 'T :> ValueType > with
    member this.Print() =
        if this.HasValue then this.Value.ToString()
        else "NULL"

printfn "\ncontains query operator"
query {
    for student in db.Student do
    select student.Age.Value
    contains 11
}
|> printfn "Is at least one student age 11? %b"

printfn "\ncount query operator"
query {
    for student in db.Student do
    select student
    count
}
```

```
}
|> printfn "Number of students: %d"

printfn "\nlast query operator."
let num =
    query {
        for number in data do
        sortBy number
        last
    }
printfn "Last number: %d" num

open Microsoft.FSharp.Linq

printfn "\nlastOrDefault query operator."
query {
    for number in data do
    sortBy number
    lastOrDefault
}
|> printfn "lastOrDefault: %d"

printfn "\nexactlyOne query operator."
let student2 =
    query {
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOne
    }
printfn "Student with StudentID = 1 is %s" student2.Name

printfn "\nexactlyOneOrDefault query operator."
let student3 =
    query {
        for student in db.Student do
        where (student.StudentID = 1)
        select student
        exactlyOneOrDefault
    }
printfn "Student with StudentID = 1 is %s" student3.Name

printfn "\nheadOrDefault query operator."
let student4 =
    query {
        for student in db.Student do
        select student
        headOrDefault
    }
printfn "head student is %s" student4.Name

printfn "\nselect query operator."
query {
    for student in db.Student do
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nwhere query operator."
query {
    for student in db.Student do
    where (student.StudentID > 4)
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nminBy query operator."
let student5 =
    query {
        for student in db.Student do
```

```
        for student in db.Student do
        minBy student.StudentID
    }

printfn "\nmaxBy query operator."
let student6 =
    query {
        for student in db.Student do
        maxBy student.StudentID
    }

printfn "\ngroupBy query operator."
query {
    for student in db.Student do
    groupBy student.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "Age: %s Count at that age: %d" (age.Print()) count)

printfn "\nsortBy query operator."
query {
    for student in db.Student do
    sortBy student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nsortByDescending query operator."
query {
    for student in db.Student do
    sortByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nthenBy query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenBy student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\nthenByDescending query operator."
query {
    for student in db.Student do
    where student.Age.HasValue
    sortBy student.Age.Value
    thenByDescending student.Name
    select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\ngroupValBy query operator."
query {
    for student in db.Student do
    groupValBy student.Name student.Age into g
    select (g, g.Key, g.Count())
}
|> Seq.iter (fun (group, age, count) ->
    printfn "Age: %s Count at that age: %d" (age.Print()) count
    group |> Seq.iter (fun name -> printfn "Name: %s" name))

printfn "\n sumByNullable query operator"
query {
    for student in db.Student do
    sumByNullable student.Age
```

```
}
|> (fun sum -> printfn "Sum of ages: %s" (sum.Print()))

printfn "\n minByNullable"
query {
    for student in db.Student do
    minByNullable student.Age
}
|> (fun age -> printfn "Minimum age: %s" (age.Print()))

printfn "\n maxByNullable"
query {
    for student in db.Student do
    maxByNullable student.Age
}
|> (fun age -> printfn "Maximum age: %s" (age.Print()))

printfn "\n averageBy"
query {
    for student in db.Student do
    averageBy (float student.StudentID)
}
|> printfn "Average student ID: %f"

printfn "\n averageByNullable"
query {
    for student in db.Student do
    averageByNullable (Nullable.float student.Age)
}
|> (fun avg -> printfn "Average age: %s" (avg.Print()))

printfn "\n find query operator"
query {
    for student in db.Student do
    find (student.Name = "Abercrombie, Kim")
}
|> (fun student -> printfn "Found a match with StudentID = %d" student.StudentID)

printfn "\n all query operator"
query {
    for student in db.Student do
    all (SqlMethods.Like(student.Name, "%,%"))
}
|> printfn "Do all students have a comma in the name? %b"

printfn "\n head query operator"
query {
    for student in db.Student do
    head
}
|> (fun student -> printfn "Found the head student with StudentID = %d" student.StudentID)

printfn "\n nth query operator"
query {
    for numbers in data do
    nth 3
}
|> printfn "Third number is %d"

printfn "\n skip query operator"
query {
    for student in db.Student do
    skip 1
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n skipWhile query operator"
query {
    for number in data do
```

```
    skipWhile (number < 3)
    select number
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sumBy query operator"
query {
    for student in db.Student do
    sumBy student.StudentID
}
|> printfn "Sum of student IDs: %d"

printfn "\n take query operator"
query {
    for student in db.Student do
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n takeWhile query operator"
query {
    for number in data do
    takeWhile (number < 10)
}
|> Seq.iter (fun number -> printfn "Number = %d" number)

printfn "\n sortByNullable query operator"
query {
    for student in db.Student do
    sortByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n sortByNullableDescending query operator"
query {
    for student in db.Student do
    sortByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullable query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullable student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "\n thenByNullableDescending query operator"
query {
    for student in db.Student do
    sortBy student.Name
    thenByNullableDescending student.Age
    select student
}
|> Seq.iter (fun student ->
    printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

printfn "All students: "
query {
    for student in db.Student do
    select student
```

```
}
|> Seq.iter (fun student -> printfn "%s %d %s" student.Name student.StudentID (student.Age.Print()))

printfn "\nCount of students: "
query {
    for student in db.Student do
    count
}
|> (fun count -> printfn "Student count: %d" count)

printfn "\nExists."
query {
    for student in db.Student do
    where
        (query {
            for courseSelection in db.CourseSelection do
            exists (courseSelection.StudentID = student.StudentID) })
    select student
}
|> Seq.iter (fun student -> printfn "%A" student.Name)

printfn "\n Group by age and count"
query {
    for n in db.Student do
    groupBy n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\n Group value by age."
query {
    for n in db.Student do
    groupValBy n.Age n.Age into g
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\nGroup students by age where age > 10."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Key.HasValue && g.Key.Value > 10)
    select (g, g.Key)
}
|> Seq.iter (fun (students, age) ->
    printfn "Age: %s" (age.Value.ToString())
    students
    |> Seq.iter (fun student -> printfn "%s" student.Name))

printfn "\nGroup students by age and print counts of number of students at each age with more than 1
student."
query {
    for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
|> Seq.iter (fun (age, ageCount) ->
    printfn "Age: %s Count: %d" (age.Print()) ageCount)

printfn "\nGroup students by age and sum ages."
query {
    for student in db.Student do
    groupBy student.Age into g
    let total = query { for student in g do sumByNullable student.Age }
    select (g.Key, g.Count(), total)
}
|> Seq.iter (fun (age, count, total) ->
    printfn "Age: %d" (age.GetValueOrDefault())
```

```
    printfn "Count: %d" count
    printfn "Total years: %s" (total.ToString()))

printfn "\nGroup students by age and count number of students at each age, and display all with count > 1 in
descending order of count."
query {
    for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
|> Seq.iter (fun (age, myCount) ->
    printfn "Age: %s" (age.Print())
    printfn "Count: %d" myCount)

printfn "\n Select students from a set of IDs"
let idList = [1; 2; 5; 10]
let idQuery =
    query { for id in idList do select id }
query {
    for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
|> Seq.iter (fun student ->
    printfn "Name: %s" student.Name)

printfn "\nLook for students with Name match _e%% pattern and take first two."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "_e%") )
    select student
    take 2
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with Name matching [abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern."
query {
    for student in db.Student do
    where (SqlMethods.Like( student.Name, "[^abc]%") )
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern and select ID."
query {
    for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%") )
    select n.StudentID
}
|> Seq.iter (fun id -> printfn "%d" id)

printfn "\n Using Contains as a query filter."
query {
    for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)
```

```
printfn "\nSearching for names from a list."
let names = [|"a";"b";"c"|]
query {
    for student in db.Student do
    if names.Contains (student.Name) then select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nJoin Student and CourseSelection tables."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    select (student, selection)
}
|> Seq.iter (fun (student, selection) -> printfn "%d %s %d" student.StudentID student.Name
selection.CourseID)

printfn "\nLeft Join Student and CourseSelection tables."
query {
    for student in db.Student do
    leftOuterJoin selection in db.CourseSelection
        on (student.StudentID = selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
    select (student, selection)
}
|> Seq.iter (fun (student, selection) ->
    let selectionID, studentID, courseID =
        match selection with
        | null -> "NULL", "NULL", "NULL"
        | sel -> (sel.ID.ToString(), sel.StudentID.ToString(), sel.CourseID.ToString())
    printfn "%d %s %d %s %s %s" student.StudentID student.Name (student.Age.GetValueOrDefault()) selectionID
studentID courseID)

printfn "\nJoin with count"
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    count
}
|> printfn "%d"

printfn "\n Join with distinct."
query {
    for student in db.Student do
    join selection in db.CourseSelection
        on (student.StudentID = selection.StudentID)
    distinct
}
|> Seq.iter (fun (student, selection) -> printfn "%s %d" student.Name selection.CourseID)

printfn "\n Join with distinct and count."
query {
    for n in db.Student do
    join e in db.CourseSelection
        on (n.StudentID = e.StudentID)
    distinct
    count
}
|> printfn "%d"

printfn "\n Selecting students with age between 10 and 15."
query {
    for student in db.Student do
    where (student.Age.Value >= 10 && student.Age.Value < 15)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)
```

```
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students with age either 11 or 12."
query {
    for student in db.Student do
    where (student.Age.Value = 11 || student.Age.Value = 12)
    select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students in a certain age range and sorting."
query {
    for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value = 13)
    sortByNullableDescending n.Age
    select n
}
|> Seq.iter (fun student -> printfn "%s %s" student.Name (student.Age.Print()))

printfn "\n Selecting students with certain ages, taking account of possibility of nulls."
query {
    for student in db.Student do
    where
        ((student.Age.HasValue && student.Age.Value = 11) ||
         (student.Age.HasValue && student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
|> Seq.iter (fun name -> printfn "%s" name)

printfn "\n Union of two queries."
module Queries =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query2.Union (query1)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Intersect of two queries."
module Queries2 =
    let query1 = query {
        for n in db.Student do
        select (n.Name, n.Age)
    }

    let query2 = query {
        for n in db.LastStudent do
        select (n.Name, n.Age)
    }

    query1.Intersect(query2)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Using if statement to alter results for special value."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))
```

```
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Using if statement to alter results special values."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         elif student.Age.HasValue && student.Age.Value = 0 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
         else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Multiple table select."
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
|> Seq.iteri (fun index (student, course) ->
    if index = 0 then
        printfn "StudentID Name Age CourseID CourseName"
    printfn "%d %s %s %d %s" student.StudentID student.Name (student.Age.Print()) course.CourseID
course.CourseName)

printfn "\nMultiple Joins"
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID = course.CourseID)
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

printfn "\nMultiple Left Outer Joins"
query {
    for student in db.Student do
    leftOuterJoin courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty() do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID = course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)
```

And here is the full output when this code is run in F# Interactive.

```
--> Referenced 'C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\3.0\Runtime\v4.0\Type
Providers\FSharp.Data.TypeProviders.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.Linq.dll'

contains query operator
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp5E3C.dll'...
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp611A.dll'...
Is at least one student age 11? true

count query operator
Number of students: 8

last query operator
```

```
last query operator.
Last number: 21

lastOrDefault query operator.
lastOrDefault: 21

exactlyOne query operator.
Student with StudentID = 1 is Abercrombie, Kim

exactlyOneOrDefault query operator.
Student with StudentID = 1 is Abercrombie, Kim

headOrDefault query operator.
head student is Abercrombie, Kim

select query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng

where query operator.
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom
StudentID, Name: 8 Peng, Yun-Feng

minBy query operator.

maxBy query operator.

groupBy query operator.
Age: NULL Count at that age: 1
Age: 10 Count at that age: 1
Age: 11 Count at that age: 1
Age: 12 Count at that age: 3
Age: 13 Count at that age: 1
Age: 14 Count at that age: 1

sortBy query operator.
StudentID, Name: 1 Abercrombie, Kim
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 4 Adams, Terry
StudentID, Name: 3 Hance, Jim
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 6 Penor, Lori
StudentID, Name: 7 Perham, Tom

sortByDescending query operator.
StudentID, Name: 7 Perham, Tom
StudentID, Name: 6 Penor, Lori
StudentID, Name: 8 Peng, Yun-Feng
StudentID, Name: 5 Hansen, Claus
StudentID, Name: 3 Hance, Jim
StudentID, Name: 4 Adams, Terry
StudentID, Name: 2 Abolrous, Hazen
StudentID, Name: 1 Abercrombie, Kim

thenBy query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Adams, Terry
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Perham, Tom
```

```
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

thenByDescending query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Perham, Tom
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Adams, Terry
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

groupValBy query operator.
Age: NULL Count at that age: 1
Name: Peng, Yun-Feng
Age: 10 Count at that age: 1
Name: Abercrombie, Kim
Age: 11 Count at that age: 1
Name: Hansen, Claus
Age: 12 Count at that age: 3
Name: Hance, Jim
Name: Adams, Terry
Name: Perham, Tom
Age: 13 Count at that age: 1
Name: Penor, Lori
Age: 14 Count at that age: 1
Name: Abolrous, Hazen

sumByNullable query operator
Sum of ages: 84

minByNullable
Minimum age: 10

maxByNullable
Maximum age: 14

averageBy
Average student ID: 4.500000

averageByNullable
Average age: 12

find query operator
Found a match with StudentID = 1

all query operator
Do all students have a comma in the name? true

head query operator
Found the head student with StudentID = 1

nth query operator
Third number is 11

skip query operator
StudentID = 2
StudentID = 3
StudentID = 4
StudentID = 5
StudentID = 6
StudentID = 7
StudentID = 8

skipWhile query operator
Number = 5
Number = 7
Number = 11
Number = 18
```

```
Number = 21

sumBy query operator
Sum of student IDs: 36

take query operator
StudentID = 1
StudentID = 2

takeWhile query operator
Number = 1
Number = 5
Number = 7

sortByNullable query operator
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 2 Abolrous, Hazen 14

sortByNullableDescending query operator
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 8 Peng, Yun-Feng NULL

thenByNullable query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12

thenByNullableDescending query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
All students:
Abercrombie, Kim 1 10
Abolrous, Hazen 2 14
Hance, Jim 3 12
Adams, Terry 4 12
Hansen, Claus 5 11
Penor, Lori 6 13
Perham, Tom 7 12
Peng, Yun-Feng 8 NULL

Count of students:
Student count: 8

Exists.
"Abercrombie, Kim"
"Abolrous, Hazen"
```

```
"Hance, Jim"
"Adams, Terry"
"Hansen, Claus"
"Perham, Tom"

Group by age and count
NULL 1
10 1
11 1
12 3
13 1
14 1

Group value by age.
NULL 1
10 1
11 1
12 3
13 1
14 1

Group students by age where age > 10.
Age: 11
Hansen, Claus
Age: 12
Hance, Jim
Adams, Terry
Perham, Tom
Age: 13
Penor, Lori
Age: 14
Abolrous, Hazen

Group students by age and print counts of number of students at each age with more than 1 student.
Age: 12 Count: 3

Group students by age and sum ages.
Age: 0
Count: 1
Total years:
Age: 10
Count: 1
Total years: 10
Age: 11
Count: 1
Total years: 11
Age: 12
Count: 3
Total years: 36
Age: 13
Count: 1
Total years: 13
Age: 14
Count: 1
Total years: 14

Group students by age and count number of students at each age, and display all with count > 1 in descending
order of count.
Age: 12
Count: 3

Select students from a set of IDs
Name: Abercrombie, Kim
Name: Abolrous, Hazen
Name: Hansen, Claus

Look for students with Name match _e% pattern and take first two.
Penor, Lori
Perham, Tom
```

```
Look for students with Name matching [abc]% pattern.
Abercrombie, Kim
Abolrous, Hazen
Adams, Terry

Look for students with name matching [^abc]% pattern.
Hance, Jim
Hansen, Claus
Penor, Lori
Perham, Tom
Peng, Yun-Feng

Look for students with name matching [^abc]% pattern and select ID.
3
5
6
7
8

Using Contains as a query filter.
Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Searching for names from a list.

Join Student and CourseSelection tables.
2 Abolrous, Hazen 2
3 Hance, Jim 3
5 Hansen, Claus 5
2 Abolrous, Hazen 2
5 Hansen, Claus 5
6 Penor, Lori 6
3 Hance, Jim 3
2 Abolrous, Hazen 2
1 Abercrombie, Kim 1
2 Abolrous, Hazen 2
5 Hansen, Claus 5
2 Abolrous, Hazen 2
3 Hance, Jim 3
2 Abolrous, Hazen 2
3 Hance, Jim 3

Left Join Student and CourseSelection tables.
1 Abercrombie, Kim 10 9 3 1
2 Abolrous, Hazen 14 1 1 2
2 Abolrous, Hazen 14 4 2 2
2 Abolrous, Hazen 14 8 3 2
2 Abolrous, Hazen 14 10 4 2
2 Abolrous, Hazen 14 12 4 2
2 Abolrous, Hazen 14 14 5 2
3 Hance, Jim 12 2 1 3
3 Hance, Jim 12 7 2 3
3 Hance, Jim 12 13 5 3
3 Hance, Jim 12 15 7 3
4 Adams, Terry 12 NULL NULL NULL
5 Hansen, Claus 11 3 1 5
5 Hansen, Claus 11 5 2 5
5 Hansen, Claus 11 11 4 5
6 Penor, Lori 13 6 2 6
7 Perham, Tom 12 NULL NULL NULL
8 Peng, Yun-Feng 0 NULL NULL NULL

Join with count
15
```

```
15

Join with distinct.
Abercrombie, Kim 2
Abercrombie, Kim 3
Abercrombie, Kim 5
Abolrous, Hazen 2
Abolrous, Hazen 5
Abolrous, Hazen 6
Abolrous, Hazen 3
Hance, Jim 2
Hance, Jim 1
Adams, Terry 2
Adams, Terry 5
Adams, Terry 2
Hansen, Claus 3
Hansen, Claus 2
Perham, Tom 3

Join with distinct and count.
15

Selecting students with age between 10 and 15.
Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Penor, Lori
Perham, Tom

Selecting students with age either 11 or 12.
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Selecting students in a certain age range and sorting.
Penor, Lori 13
Perham, Tom 12
Hance, Jim 12
Adams, Terry 12

Selecting students with certain ages, taking account of possibility of nulls.
Hance, Jim
Adams, Terry

Union of two queries.
Abercrombie, Kim 10
Abolrous, Hazen 14
Hance, Jim 12
Adams, Terry 12
Hansen, Claus 11
Penor, Lori 13
Perham, Tom 12
Peng, Yun-Feng NULL

Intersect of two queries.

Using if statement to alter results for special value.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Using if statement to alter results special values
```

```
Using if statement to alter results special values.
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Multiple table select.
StudentID Name Age CourseID CourseName
1 Abercrombie, Kim 10 1 Algebra I
2 Abolrous, Hazen 14 1 Algebra I
3 Hance, Jim 12 1 Algebra I
4 Adams, Terry 12 1 Algebra I
5 Hansen, Claus 11 1 Algebra I
6 Penor, Lori 13 1 Algebra I
7 Perham, Tom 12 1 Algebra I
8 Peng, Yun-Feng NULL 1 Algebra I
1 Abercrombie, Kim 10 2 Trigonometry
2 Abolrous, Hazen 14 2 Trigonometry
3 Hance, Jim 12 2 Trigonometry
4 Adams, Terry 12 2 Trigonometry
5 Hansen, Claus 11 2 Trigonometry
6 Penor, Lori 13 2 Trigonometry
7 Perham, Tom 12 2 Trigonometry
8 Peng, Yun-Feng NULL 2 Trigonometry
1 Abercrombie, Kim 10 3 Algebra II
2 Abolrous, Hazen 14 3 Algebra II
3 Hance, Jim 12 3 Algebra II
4 Adams, Terry 12 3 Algebra II
5 Hansen, Claus 11 3 Algebra II
6 Penor, Lori 13 3 Algebra II
7 Perham, Tom 12 3 Algebra II
8 Peng, Yun-Feng NULL 3 Algebra II
1 Abercrombie, Kim 10 4 History
2 Abolrous, Hazen 14 4 History
3 Hance, Jim 12 4 History
4 Adams, Terry 12 4 History
5 Hansen, Claus 11 4 History
6 Penor, Lori 13 4 History
7 Perham, Tom 12 4 History
8 Peng, Yun-Feng NULL 4 History
1 Abercrombie, Kim 10 5 English
2 Abolrous, Hazen 14 5 English
3 Hance, Jim 12 5 English
4 Adams, Terry 12 5 English
5 Hansen, Claus 11 5 English
6 Penor, Lori 13 5 English
7 Perham, Tom 12 5 English
8 Peng, Yun-Feng NULL 5 English
1 Abercrombie, Kim 10 6 French
2 Abolrous, Hazen 14 6 French
3 Hance, Jim 12 6 French
4 Adams, Terry 12 6 French
5 Hansen, Claus 11 6 French
6 Penor, Lori 13 6 French
7 Perham, Tom 12 6 French
8 Peng, Yun-Feng NULL 6 French
1 Abercrombie, Kim 10 7 Chinese
2 Abolrous, Hazen 14 7 Chinese
3 Hance, Jim 12 7 Chinese
4 Adams, Terry 12 7 Chinese
5 Hansen, Claus 11 7 Chinese
6 Penor, Lori 13 7 Chinese
7 Perham, Tom 12 7 Chinese
8 Peng, Yun-Feng NULL 7 Chinese
```

```
Multiple Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Perham, Tom Algebra II

Multiple Left Outer Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Penor, Lori
Perham, Tom Algebra II
Peng, Yun-Feng

type schema
val db : schema.ServiceTypes.SimpleDataContextTypes.MyDatabase1
val student : System.Data.Linq.Table<schema.ServiceTypes.Student>
val data : int list = [1; 5; 7; 11; 18; 21]
type Nullable<'T
                when 'T : (new : unit ->  'T) and 'T : struct and
                     'T :> System.ValueType> with
  member Print : unit -> string
val num : int = 21
val student2 : schema.ServiceTypes.Student
val student3 : schema.ServiceTypes.Student
val student4 : schema.ServiceTypes.Student
val student5 : int = 1
val student6 : int = 8
val idList : int list = [1; 2; 5; 10]
val idQuery : seq<int>
val names : string [] = [|"a"; "b"; "c"|]
module Queries = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
module Queries2 = begin
  val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
  val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
```

See also

- F# Language Reference
- Linq.QueryBuilder Class

- Computation Expressions

Code Quotations

7/30/2019 • 7 minutes to read • Edit Online

NOTE

The API reference link will take you to MSDN. The docs.microsoft.com API reference is not complete.

This topic describes *code quotations*, a language feature that enables you to generate and work with F# code expressions programmatically. This feature lets you generate an abstract syntax tree that represents F# code. The abstract syntax tree can then be traversed and processed according to the needs of your application. For example, you can use the tree to generate F# code or generate code in some other language.

Quoted Expressions

A *quoted expression* is an F# expression in your code that is delimited in such a way that it is not compiled as part of your program, but instead is compiled into an object that represents an F# expression. You can mark a quoted expression in one of two ways: either with type information or without type information. If you want to include type information, you use the symbols `<@` and `@>` to delimit the quoted expression. If you do not need type information, you use the symbols `<@@` and `@@>`. The following code shows typed and untyped quotations.

```
open Microsoft.FSharp.Quotations
// A typed code quotation.
let expr : Expr<int> = <@ 1 + 1 @>
// An untyped code quotation.
let expr2 : Expr = <@@ 1 + 1 @@>
```

Traversing a large expression tree is faster if you do not include type information. The resulting type of an expression quoted with the typed symbols is `Expr<'T>`, where the type parameter has the type of the expression as determined by the F# compiler's type inference algorithm. When you use code quotations without type information, the type of the quoted expression is the non-generic type Expr. You can call the Raw property on the typed `Expr` class to obtain the untyped `Expr` object.

There are a variety of static methods that allow you to generate F# expression objects programmatically in the `Expr` class without using quoted expressions.

Note that a code quotation must include a complete expression. For a `let` binding, for example, you need both the definition of the bound name and an additional expression that uses the binding. In verbose syntax, this is an expression that follows the `in` keyword. At the top-level in a module, this is just the next expression in the module, but in a quotation, it is explicitly required.

Therefore, the following expression is not valid.

```
// Not valid:
// <@ let f x = x + 1 @>
```

But the following expressions are valid.

```
// Valid:
<@ let f x = x + 10 in f 20 @>
// Valid:
<@
    let f x = x + 10
    f 20
@>
```

To evaluate F# quotations, you must use the F# Quotation Evaluator. It provides support for evaluating and executing F# expression objects.

Expr Type

An instance of the `Expr` type represents an F# expression. Both the generic and the non-generic `Expr` types are documented in the F# library documentation. For more information, see Microsoft.FSharp.Quotations Namespace and Quotations.Expr Class.

Splicing Operators

Splicing enables you to combine literal code quotations with expressions that you have created programmatically or from another code quotation. The `%` and `%%` operators enable you to add an F# expression object into a code quotation. You use the `%` operator to insert a typed expression object into a typed quotation; you use the `%%` operator to insert an untyped expression object into an untyped quotation. Both operators are unary prefix operators. Thus if `expr` is an untyped expression of type `Expr`, the following code is valid.

```
<@@ 1 + %%expr @@>
```

And if `expr` is a typed quotation of type `Expr<int>`, the following code is valid.

```
<@ 1 + %expr @>
```

Example

Description

The following example illustrates the use of code quotations to put F# code into an expression object and then print the F# code that represents the expression. A function `println` is defined that contains a recursive function `print` that displays an F# expression object (of type `Expr`) in a friendly format. There are several active patterns in the Microsoft.FSharp.Quotations.Patterns and Microsoft.FSharp.Quotations.DerivedPatterns modules that can be used to analyze expression objects. This example does not include all the possible patterns that might appear in an F# expression. Any unrecognized pattern triggers a match to the wildcard pattern (`_`) and is rendered by using the `ToString` method, which, on the `Expr` type, lets you know the active pattern to add to your match expression.

Code

```
module Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let println expr =
    let rec print expr =
        match expr with
        | Application(expr1, expr2) ->
```

```
            // Function application.
            print expr1
            printf " "
            print expr2
        | SpecificCall <@@ (+) @@> (_, _, exprList) ->
            // Matches a call to (+). Must appear before Call pattern.
            print exprList.Head
            printf " + "
            print exprList.Tail.Head
        | Call(exprOpt, methodInfo, exprList) ->
            // Method or module function call.
            match exprOpt with
            | Some expr -> print expr
            | None -> printf "%s" methodInfo.DeclaringType.Name
            printf ".%s(" methodInfo.Name
            if (exprList.IsEmpty) then printf ")" else
            print exprList.Head
            for expr in exprList.Tail do
                printf ","
                print expr
            printf ")"
        | Int32(n) ->
            printf "%d" n
        | Lambda(param, body) ->
            // Lambda expression.
            printf "fun (%s:%s) -> " param.Name (param.Type.ToString())
            print body
        | Let(var, expr1, expr2) ->
            // Let binding.
            if (var.IsMutable) then
                printf "let mutable %s = " var.Name
            else
                printf "let %s = " var.Name
            print expr1
            printf " in "
            print expr2
        | PropertyGet(_, propOrValInfo, _) ->
            printf "%s" propOrValInfo.Name
        | String(str) ->
            printf "%s" str
        | Value(value, typ) ->
            printf "%s" (value.ToString())
        | Var(var) ->
            printf "%s" var.Name
        | _ -> printf "%s" (expr.ToString())
    print expr
    printfn ""


let a = 2

// exprLambda has type "(int -> int)".
let exprLambda = <@ fun x -> x + 1 @>
// exprCall has type unit.
let exprCall = <@ a + 1 @>

println exprLambda
println exprCall
println <@@ let f x = x + 10 in f 10 @@>
```

Output

```
fun (x:System.Int32) -> x + 1
a + 1
let f = fun (x:System.Int32) -> x + 10 in f 10
```

Example

Description

You can also use the three active patterns in the ExprShape module to traverse expression trees with fewer active patterns. These active patterns can be useful when you want to traverse a tree but you do not need all the information in most of the nodes. When you use these patterns, any F# expression matches one of the following three patterns: `ShapeVar` if the expression is a variable, `ShapeLambda` if the expression is a lambda expression, or `ShapeCombination` if the expression is anything else. If you traverse an expression tree by using the active patterns as in the previous code example, you have to use many more patterns to handle all possible F# expression types, and your code will be more complex. For more information, see ExprShape.ShapeVar|ShapeLambda|ShapeCombination Active Pattern.

The following code example can be used as a basis for more complex traversals. In this code, an expression tree is created for an expression that involves a function call, `add`. The SpecificCall active pattern is used to detect any call to `add` in the expression tree. This active pattern assigns the arguments of the call to the `exprList` value. In this case, there are only two, so these are pulled out and the function is called recursively on the arguments. The results are inserted into a code quotation that represents a call to `mul` by using the splice operator (`%%`). The `println` function from the previous example is used to display the results.

The code in the other active pattern branches just regenerates the same expression tree, so the only change in the resulting expression is the change from `add` to `mul`.

Code

```
module Module1
open Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape

let add x y = x + y
let mul x y = x * y

let rec substituteExpr expression =
    match expression with
    | SpecificCall <@@ add @@> (_, _, exprList) ->
        let lhs = substituteExpr exprList.Head
        let rhs = substituteExpr exprList.Tail.Head
        <@@ mul %%lhs %%rhs @@>
    | ShapeVar var -> Expr.Var var
    | ShapeLambda (var, expr) -> Expr.Lambda (var, substituteExpr expr)
    | ShapeCombination(shapeComboObject, exprList) ->
        RebuildShapeCombination(shapeComboObject, List.map substituteExpr exprList)

let expr1 = <@@ 1 + (add 2 (add 3 4)) @@>
println expr1
let expr2 = substituteExpr expr1
println expr2
```

Output

```
1 + Module1.add(2,Module1.add(3,4))
1 + Module1.mul(2,Module1.mul(3,4))
```

See also

- F# Language Reference

The fixed keyword

12/19/2018 • 2 minutes to read • Edit Online

F# 4.1 introduces the `fixed` keyword, which allows you to "pin" a local onto the stack to prevent it from being collected or moved during garbage-collection. It is used for low-level programming scenarios.

Syntax

```
use ptr = fixed expression
```

Remarks

This extends the syntax of expressions to allow extracting a pointer and binding it to a name which is prevented from being collected or moved during garbage-collection.

A pointer from an expression is fixed via the `fixed` keyword is bound to an identifier via the `use` keyword. The semantics of this are similar to resource management via the `use` keyword. The pointer is fixed while it is in scope, and once it is out of scope, it is no longer fixed. `fixed` cannot be used outside the context of a `use` binding. You must bind the pointer to a name with `use`.

Use of `fixed` must occur within an expression in a function or a method. It cannot be used at a script-level or module-level scope.

Like all pointer code, this is an unsafe feature and will emit a warning when used.

Example

```
open Microsoft.FSharp.NativeInterop

type Point = { mutable X: int; mutable Y: int}

let squareWithPointer (p: nativeptr<int>) =
    // Dereference the pointer at the 0th address.
    let mutable value = NativePtr.get p 0

    // Perform some work
    value <- value * value

    // Set the value in the pointer at the 0th address.
    NativePtr.set p 0 value

let pnt = { X = 1; Y = 2 }
printfn "pnt before - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 2

// Note that the use of 'fixed' is inside a function.
// You cannot fix a pointer at a script-level or module-level scope.
let doPointerWork() =
    use ptr = fixed &pnt.Y

    // Square the Y value
    squareWithPointer ptr
    printfn "pnt after - X: %d Y: %d" pnt.X pnt.Y // prints 1 and 4

doPointerWork()
```

See also

- NativePtr Module

Byrefs

4/5/2019 • 6 minutes to read • Edit Online

F# has two major feature areas that deal in the space of low-level programming:

- The `byref` / `inref` / `outref` types, which are a managed pointers. They have restrictions on usage so that you cannot compile a program that is invalid at runtime.
- A `byref` -like struct, which is a structure that has similar semantics and the same compile-time restrictions as `byref<'T>`. One example is Span<T>.

Syntax

```
// Byref types as parameters
let f (x: byref<'T>) = ()
let g (x: inref<'T>) = ()
let h (x: outref<'T>) = ()

// Calling a function with a byref parameter
let mutable x = 3
f &x

// Declaring a byref-like struct
open System.Runtime.CompilerServices

[<Struct; IsByRefLike>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

Byref, inref, and outref

There are three forms of `byref`:

- `inref<'T>`, a managed pointer for reading the underlying value.
- `outref<'T>`, a managed pointer for writing to the underlying value.
- `byref<'T>`, a managed pointer for reading and writing the underlying value.

A `byref<'T>` can be passed where an `inref<'T>` is expected. Similarly, a `byref<'T>` can be passed where an `outref<'T>` is expected.

Using byrefs

To use a `inref<'T>`, you need to get a pointer value with `&`:

```
open System

let f (dt: inref<DateTime>) =
    printfn "Now: %s" (dt.ToString())

let usage =
    let dt = DateTime.Now
    f &dt // Pass a pointer to 'dt'
```

To write to the pointer by using an `outref<'T>` or `byref<'T>`, you must also make the value you grab a pointer to `mutable`.

```
open System

let f (dt: byref<DateTime>) =
    printfn "Now: %s" (dt.ToString())
    dt <- DateTime.Now

// Make 'dt' mutable
let mutable dt = DateTime.Now

// Now you can pass the pointer to 'dt'
f &dt
```

If you are only writing the pointer instead of reading it, consider using `outref<'T>` instead of `byref<'T>`.

Inref semantics

Consider the following code:

```
let f (x: inref<SomeStruct>) = x.SomeField
```

Semantically, this means the following:

- The holder of the `x` pointer may only use it to read the value.
- Any pointer acquired to `struct` fields nested within `SomeStruct` are given type `inref<_>`.

The following is also true:

- There is no implication that other threads or aliases do not have write access to `x`.
- There is no implication that `SomeStruct` is immutable by virtue of `x` being an `inref`.

However, for F# value types that **are** immutable, the `this` pointer is inferred to be an `inref`.

All of these rules together mean that the holder of an `inref` pointer may not modify the immediate contents of the memory being pointed to.

Outref semantics

The purpose of `outref<'T>` is to indicate that the pointer should only be read from. Unexpectedly, `outref<'T>` permits reading the underlying value despite its name. This is for compatibility purposes. Semantically, `outref<'T>` is no different than `byref<'T>`.

Interop with C#

C# supports the `in ref` and `out ref` keywords, in addition to `ref` returns. The following table shows how F# interprets what C# emits:

C# CONSTRUCT	F# INFERS
`ref` return value	`outref<'T>`
`ref readonly` return value	`inref<'T>`
`in ref` parameter	`inref<'T>`
`out ref` parameter	`outref<'T>`

The following table shows what F# emits:

F# CONSTRUCT	EMITTED CONSTRUCT
`inref<'T>` argument	`[In]` attribute on argument
`inref<'T>` return	`modreq` attribute on value
`inref<'T>` in abstract slot or implementation	`modreq` on argument or return
`outref<'T>` argument	`[Out]` attribute on argument

Type inference and overloading rules

An `inref<'T>` type is inferred by the F# compiler in the following cases:

1. A .NET parameter or return type that has an `IsReadOnly` attribute.
2. The `this` pointer on a struct type that has no mutable fields.
3. The address of a memory location derived from another `inref<_>` pointer.

When an implicit address of an `inref` is being taken, an overload with an argument of type `SomeType` is preferred to an overload with an argument of type `inref<SomeType>`. For example:

```
type C() =
    static member M(x: System.DateTime) = x.AddDays(1.0)
    static member M(x: inref<System.DateTime>) = x.AddDays(2.0)
    static member M2(x: System.DateTime, y: int) = x.AddDays(1.0)
    static member M2(x: inref<System.DateTime>, y: int) = x.AddDays(2.0)

let res = System.DateTime.Now
let v =  C.M(res)
let v2 =  C.M2(res, 4)
```

In both cases, the overloads taking `System.DateTime` are resolved rather than the overloads taking `inref<System.DateTime>`.

Byref-like structs

In addition to the `byref` / `inref` / `outref` trio, you can define your own structs that can adhere to `byref`-like semantics. This is done with the IsByRefLikeAttribute attribute:

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` does not imply `Struct`. Both must be present on the type.

A "`byref`-like" struct in F# is a stack-bound value type. It is never allocated on the managed heap. A `byref`-like struct is useful for high-performance programming, as it is enforced with set of strong checks about lifetime and non-capture. The rules are:

- They can be used as function parameters, method parameters, local variables, method returns.

- They cannot be static or instance members of a class or normal struct.
- They cannot be captured by any closure construct (`async` methods or lambda expressions).
- They cannot be used as a generic parameter.

This last point is crucial for F# pipeline-style programming, as `|>` is a generic function that parameterizes its input types. This restriction may be relaxed for `|>` in the future, as it is inline and does not make any calls to non-inlined generic functions in its body.

Although these rules very strongly restrict usage, they do so to fulfill the promise of high-performance computing in a safe manner.

Byref returns

Byref returns from F# functions or members can be produced and consumed. When consuming a `byref`-returning method, the value is implicitly dereferenced. For example:

```
let safeSum(bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    sum

let sum = safeSum(mySpanOfBytes)
printfn "%d" sum // 'sum' is of type 'int'
```

To avoid the implicit dereference, such as passing a reference through multiple chained calls, use `&x` (where `x` is the value).

You can also directly assign to a return `byref`. Consider the following (highly imperative) program:

```
type C() =
    let mutable nums = [| 1; 3; 7; 15; 31; 63; 127; 255; 511; 1023 |]

    override __.ToString() = String.Join(' ', nums)

    member __.FindLargestSmallerThan(target: int) =
        let mutable ctr = nums.Length - 1

        while ctr > 0 && nums.[ctr] >= target do ctr <- ctr - 1

        if ctr > 0 then &nums.[ctr] else &nums.[0]

[<EntryPoint>]
let main argv =
    let c = C()
    printfn "Original sequence: %s" (c.ToString())

    let v = &c.FindLargestSmallerThan 16

    v <- v*2 // Directly assign to the byref return

    printfn "New sequence:      %s" (c.ToString())

    0 // return an integer exit code
```

This is the output:

```
Original sequence: 1 3 7 15 31 63 127 255 511 1023
New sequence:      1 3 7 30 31 63 127 255 511 1023
```

Scoping for byrefs

A `let`-bound value cannot have its reference exceed the scope in which it was defined. For example, the following is disallowed:

```
let test2 () =
    let x = 12
    &x // Error: 'x' exceeds its defined scope!

let test () =
    let x =
        let y = 1
        &y // Error: `y` exceeds its defined scope!
    ()
```

This prevents you from getting different results depending on if you compile with optimizations on or off.

Reference Cells

7/30/2019 • 2 minutes to read • Edit Online

Reference cells are storage locations that enable you to create mutable values with reference semantics.

Syntax

```
ref expression
```

Remarks

You use the `ref` operator before a value to create a new reference cell that encapsulates the value. You can then change the underlying value because it is mutable.

A reference cell holds an actual value; it is not just an address. When you create a reference cell by using the `ref` operator, you create a copy of the underlying value as an encapsulated mutable value.

You can dereference a reference cell by using the `!` (bang) operator.

The following code example illustrates the declaration and use of reference cells.

```
// Declare a reference.
let refVar = ref 6

// Change the value referred to by the reference.
refVar := 50

// Dereference by using the ! operator.
printfn "%d" !refVar
```

The output is `50`.

Reference cells are instances of the `Ref` generic record type, which is declared as follows.

```
type Ref<'a> =
{ mutable contents: 'a }
```

The type `'a ref` is a synonym for `Ref<'a>`. The compiler and IntelliSense in the IDE display the former for this type, but the underlying definition is the latter.

The `ref` operator creates a new reference cell. The following code is the declaration of the `ref` operator.

```
let ref x = { contents = x }
```

The following table shows the features that are available on the reference cell.

OPERATOR, MEMBER, OR FIELD	DESCRIPTION	TYPE	DEFINITION
`!` (dereference operator)	Returns the underlying value.	`'a ref -> 'a`	`let (!) r = r.contents`

OPERATOR, MEMBER, OR FIELD	DESCRIPTION	TYPE	DEFINITION
`:=` (assignment operator)	Changes the underlying value.	`'a ref -> 'a -> unit`	`let (:=) r x = r.contents <- x`
`ref` (operator)	Encapsulates a value into a new reference cell.	`'a -> 'a ref`	`let ref x = { contents = x }`
`Value` (property)	Gets or sets the underlying value.	`unit -> 'a`	`member x.Value = x.contents`
`contents` (record field)	Gets or sets the underlying value.	`'a`	`let ref x = { contents = x }`

There are several ways to access the underlying value. The value returned by the dereference operator (`!`) is not an assignable value. Therefore, if you are modifying the underlying value, you must use the assignment operator (`:=`) instead.

Both the `Value` property and the `contents` field are assignable values. Therefore, you can use these to either access or change the underlying value, as shown in the following code.

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

The output is as follows.

```
10
10
11
12
```

The field `contents` is provided for compatibility with other versions of ML and will produce a warning during compilation. To disable the warning, use the `--mlcompatibility` compiler option. For more information, see Compiler Options.

C# programmers should know that `ref` in C# is not the same thing as `ref` in F#. The equivalent constructs in F# are byrefs, which are a different concept from reference cells.

Values marked as `mutable` may be automatically promoted to `'a ref` if captured by a closure; see Values.

See also

- F# Language Reference
- Parameters and Arguments
- Symbol and Operator Reference
- Values

Compiler Directives

7/30/2019 • 4 minutes to read • Edit Online

This topic describes processor directives and compiler directives.

Preprocessor Directives

A preprocessor directive is prefixed with the # symbol and appears on a line by itself. It is interpreted by the preprocessor, which runs before the compiler itself.

The following table lists the preprocessor directives that are available in F#.

DIRECTIVE	DESCRIPTION
`#if` *symbol*	Supports conditional compilation. Code in the section after the `#if` is included if the *symbol* is defined. The symbol can also be negated with `!`.
`#else`	Supports conditional compilation. Marks a section of code to include if the symbol used with the previous `#if` is not defined.
`#endif`	Supports conditional compilation. Marks the end of a conditional section of code.
`#` [line] *int*, `#` [line] *int string*, `#` [line] *int verbatim-string*	Indicates the original source code line and file name, for debugging. This feature is provided for tools that generate F# source code.
`#nowarn` *warningcode*	Disables a compiler warning or warnings. To disable a warning, find its number from the compiler output and include it in quotation marks. Omit the "FS" prefix. To disable multiple warning numbers on the same line, include each number in quotation marks, and separate each string by a space. For example:

```
#nowarn "9" "40"
```

The effect of disabling a warning applies to the entire file, including portions of the file that precede the directive.|

Conditional Compilation Directives

Code that is deactivated by one of these directives appears dimmed in the Visual Studio Code Editor.

> **NOTE**
>
> The behavior of the conditional compilation directives is not the same as it is in other languages. For example, you cannot use Boolean expressions involving symbols, and `true` and `false` have no special meaning. Symbols that you use in the `if` directive must be defined by the command line or in the project settings; there is no `define` preprocessor directive.

The following code illustrates the use of the `#if`, `#else`, and `#endif` directives. In this example, the code contains two versions of the definition of `function1`. When `VERSION1` is defined by using the -define compiler

option, the code between the `#if` directive and the `#else` directive is activated. Otherwise, the code between `#else` and `#endif` is activated.

```
#if VERSION1
let function1 x y =
   printfn "x: %d y: %d" x y
   x + 2 * y
#else
let function1 x y =
   printfn "x: %d y: %d" x y
   x - 2*y
#endif

let result = function1 10 20
```

There is no `#define` preprocessor directive in F#. You must use the compiler option or project settings to define the symbols used by the `#if` directive.

Conditional compilation directives can be nested. Indentation is not significant for preprocessor directives.

You can also negate a symbol with `!`. In this example, a string's value is something only when *not* debugging:

```
#if !DEBUG
let str = "Not debugging!"
#else
let str = "Debugging!"
#endif
```

Line Directives

When building, the compiler reports errors in F# code by referencing line numbers on which each error occurs. These line numbers start at 1 for the first line in a file. However, if you are generating F# source code from another tool, the line numbers in the generated code are generally not of interest, because the errors in the generated F# code most likely arise from another source. The `#line` directive provides a way for authors of tools that generate F# source code to pass information about the original line numbers and source files to the generated F# code.

When you use the `#line` directive, file names must be enclosed in quotation marks. Unless the verbatim token (`@`) appears in front of the string, you must escape backslash characters by using two backslash characters instead of one in order to use them in the path. The following are valid line tokens. In these examples, assume that the original file `Script1` results in an automatically generated F# code file when it is run through a tool, and that the code at the location of these directives is generated from some tokens at line 25 in file `Script1`.

```
# 25
#line 25
#line 25 "C:\\Projects\\MyProject\\MyProject\\Script1"
#line 25 @"C:\Projects\MyProject\MyProject\Script1"
# 25 @"C:\Projects\MyProject\MyProject\Script1"
```

These tokens indicate that the F# code generated at this location is derived from some constructs at or near line `25` in `Script1`.

Compiler Directives

Compiler directives resemble preprocessor directives, because they are prefixed with a # sign, but instead of being interpreted by the preprocessor, they are left for the compiler to interpret and act on.

The following table lists the compiler directive that is available in F#.

DIRECTIVE	DESCRIPTION
`#light` ["on"|"off"]	Enables or disables lightweight syntax, for compatibility with other versions of ML. By default, lightweight syntax is enabled. Verbose syntax is always enabled. Therefore, you can use both lightweight syntax and verbose syntax. The directive `#light` by itself is equivalent to `#light "on"`. If you specify `#light "off"`, you must use verbose syntax for all language constructs. Syntax in the documentation for F# is presented with the assumption that you are using lightweight syntax. For more information, see Verbose Syntax.

For interpreter (fsi.exe) directives, see Interactive Programming with F#.

See also

- F# Language Reference
- Compiler Options

Compiler Options

5/15/2019 • 7 minutes to read • Edit Online

This topic describes compiler command-line options for the F# compiler, fsc.exe. The compilation environment can also be controlled by setting the project properties.

Compiler Options Listed Alphabetically

The following table shows compiler options listed alphabetically. Some of the F# compiler options are similar to the C# compiler options. If that is the case, a link to the C# compiler options topic is provided.

COMPILER OPTION	DESCRIPTION
`-a filename.fs`	Generates a library from the specified file. This option is a short form of `--target:library filename.fs`.
`--baseaddress:address`	Specifies the preferred base address at which to load a DLL. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /baseaddress (C# Compiler Options).
`--codepage:id`	Specifies which code page to use during compilation if the required page isn't the current default code page for the system. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /code pages (C# Compiler Options).
`--consolecolors`	Specifies that errors and warnings use color-coded text on the console.
`--crossoptimize[+|-]`	Enables or disables cross-module optimizations.
`--delaysign[+|-]`	Delay-signs the assembly using only the public portion of the strong name key. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /delaysign (C# Compiler Options).
`--checked[+|-]`	Enables or disables the generation of overflow checks. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /checked (C# Compiler Options).

COMPILER OPTION	DESCRIPTION

`--debug[+|-]`
 	
`-g[+|-]`
 	
`--debug:[full|pdbonly]`
 	
`-g: [full|pdbonly]`
	Enables or disables the generation of debug information, or specifies the type of debug information to generate. The default is full, which allows attaching to a running program. Choose **pdbonly** to get limited debugging information stored in a pdb (program database) file.
	
	Equivalent to the C# compiler option of the same name. For more information, see
	
	/debug (C# Compiler Options).

`--define:symbol`
 	
`-d:symbol`
	Defines a symbol for use in conditional compilation.

`--deterministic[+|-]`	Produces a deterministic assembly (including module version GUID and timestamp). This option cannot be used with wildcard version numbers, and only supports embedded and portable debugging types
`--doc:xmldoc-filename`	Instructs the compiler to generate XML documentation comments to the file specified. For more information, see XML Documentation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /doc (C# Compiler Options).
`--fullpaths`	Instructs the compiler to generate fully qualified paths. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /fullpaths (C# Compiler Options).

`--help`
 	
`-?`
	Displays usage information, including a brief description of all the compiler options.

`--highentropyva[+|-]`	Enable or disable high-entropy address space layout randomization (ASLR), an enhanced security feature. The OS randomizes the locations in memory where infrastructure for applications (such as the stack and heap) are loaded. If you enable this option, operating systems can use this randomization to use the full 64-bit address-space on a 64-bit machine.
`--keycontainer:key-container-name`	Specifies a strong name key container.
`--keyfile:filename`	Specifies the name of a public key file for signing the generated assembly.

`--lib:folder-name`
 	
`-I:folder-name`
	Specifies a directory to be searched for assemblies that are referenced.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /lib (C# Compiler Options).

COMPILER OPTION	DESCRIPTION
`--linkresource:resource-info`	Links a specified resource to the assembly. The format of resource-info is `filename[name[public|private]]` Linking a single resource with this option is an alternative to embedding an entire resource file with the `--resource` option. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /linkresource (C# Compiler Options).
`--mlcompatibility`	Ignores warnings that appear when you use features that are designed for compatibility with other versions of ML.
`--noframework`	Disables the default reference to the .NET Framework assembly.
`--nointerfacedata`	Instructs the compiler to omit the resource it normally adds to an assembly that includes F#-specific metadata.
`--nologo`	Doesn't show the banner text when launching the compiler.
`--nooptimizationdata`	Instructs the compiler to only include optimization essential for implementing inlined constructs. Inhibits cross-module inlining but improves binary compatibility.
`--nowin32manifest`	Instructs the compiler to omit the default Win32 manifest.
`--nowarn:warning-number-list`	Disables specific warnings listed by number. Separate each warning number by a comma. You can discover the warning number for any warning from the compilation output. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /nowarn (C# Compiler Options).

`--optimize[+|-][optimization-option-list]`
 	
`-O[+|-] [optimization-option-list]`
	Enables or disables optimizations. Some optimization options can be disabled or enabled selectively by listing them. These are: `nojitoptimize`, `nojittracking`, `nolocaloptimize`, `nocrossoptimize`, `notailcalls`.

`--out:output-filename`
 	
`-o:output-filename`
	Specifies the name of the compiled assembly or module.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /out (C# Compiler Options).

`--pdb:pdb-filename`	Names the output debug PDB (program database) file. This option only applies when `--debug` is also enabled. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /pdb (C# Compiler Options).

COMPILER OPTION	DESCRIPTION
`--platform:platform-name`	Specifies that the generated code will only run on the specified platform (`x86` , `Itanium` , or `x64`), or, if the platform-name `anycpu` is chosen, specifies that the generated code can run on any platform. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /platform (C# Compiler Options).
`--preferreduilang:lang`	Specifies the preferred output language culture name (for example, `es-ES` , `ja-JP`).
`--quotations-debug`	Specifies that extra debugging information should be emitted for expressions that are derived from F# quotation literals and reflected definitions. The debug information is added to the custom attributes of an F# expression tree node. See Code Quotations and Expr.CustomAttributes.

`--reference:assembly-filename`
 	
`-r:assembly-filename`
	Makes code from an F# or .NET Framework assembly available to the code being compiled.
	
	This compiler option is equivalent to the C# compiler option of the same name. For more information, see /reference (C# Compiler Options).

`--resource:resource-filename`	Embeds a managed resource file into the generated assembly. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /resource (C# Compiler Options).
`--sig:signature-filename`	Generates a signature file based on the generated assembly. For more information about signature files, see Signatures.
`--simpleresolution`	Specifies that assembly references should be resolved using directory-based Mono rules rather than MSBuild resolution. The default is to use MSBuild resolution except when running under Mono.
`--standalone`	Specifies to produce an assembly that contains all of its dependencies so that it runs by itself without the need for additional assemblies, such as the F# library.
`--staticlink:assembly-name`	Statically links the given assembly and all referenced DLLs that depend on this assembly. Use the assembly name, not the DLL name.
`--subsystemversion`	Specifies the version of the OS subsystem to be used by the generated executable. Use 6.02 for Windows 8.1, 6.01 for Windows 7, 6.00 for Windows Vista. This option only applies to executables, not DLLs, and need only be used if your application depends on specific security features available only on certain versions of the OS. If this option is used, and a user attempts to execute your application on a lower version of the OS, it will fail with an error message.

COMPILER OPTION	DESCRIPTION
`--tailcalls[+|-]`	Enables or disables the use of the tail IL instruction, which causes the stack frame to be reused for tail recursive functions. This option is enabled by default.
`--target:[exe|winexe|library|module] filename`	Specifies the type and file name of the generated compiled code. • `exe` means a console application. • `winexe` means a Windows application, which differs from the console application in that it does not have standard input/output streams (stdin, stdout, and stderr) defined. • `library` is an assembly without an entry point. • `module` is a .NET Framework module (.netmodule), which can later be combined with other modules into an assembly. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /target (C# Compiler Options).
`--times`	Displays timing information for compilation.
`--utf8output`	Enables printing compiler output in the UTF-8 encoding.
`--warn:warning-level`	Sets a warning level (0 to 5). The default level is 3. Each warning is given a level based on its severity. Level 5 gives more, but less severe, warnings than level 1. Level 5 warnings are: 21 (recursive use checked at runtime), 22 (`let rec` evaluated out of order), 45 (full abstraction), and 52 (defensive copy). All other warnings are level 2. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /warn (C# Compiler Options).
`--warnon:warning-number-list`	Enable specific warnings that might be off by default or disabled by another command-line option. In F# 3.0, only the 1182 (unused variables) warning is off by default.
`--warnaserror[+|-] [warning-number-list]`	Enables or disables the option to report warnings as errors. You can provide specific warning numbers to be disabled or enabled. Options later in the command line override options earlier in the command line. For example, to specify the warnings that you don't want reported as errors, specify `--warnaserror+` `--warnaserror-:warning-number-list`. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /warnaserror (C# Compiler Options).
`--win32manifest:manifest-filename`	Adds a Win32 manifest file to the compilation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /win32manifest (C# Compiler Options).

COMPILER OPTION	DESCRIPTION
`--win32res:resource-filename`	Adds a Win32 resource file to the compilation. This compiler option is equivalent to the C# compiler option of the same name. For more information, see /win32res ((C#) Compiler Options).

Related articles

TITLE	DESCRIPTION
F# Interactive Options	Describes command-line options supported by the F# interpreter, fsi.exe.
Project Properties Reference	Describes the UI for projects, including project property pages that provide build options.

F# Interactive Options

11/13/2018 • 3 minutes to read • Edit Online

> **NOTE**
> This article currently describes the experience for Windows only. It will be rewritten.

This topic describes the command-line options supported by F# Interactive, `fsi.exe`. F# Interactive accepts many of the same command line options as the F# compiler, but also accepts some additional options.

Using F# Interactive for Scripting

F# Interactive, `fsi.exe`, can be launched interactively, or it can be launched from the command line to run a script. The command line syntax is

```
> fsi.exe [options] [ script-file [arguments] ]
```

The file extension for F# script files is `.fsx`.

Table of F# Interactive Options

The following table summarizes the options supported by F# Interactive. You can set these options on the command-line or through the Visual Studio IDE. To set these options in the Visual Studio IDE, open the **Tools** menu, select **Options...**, then expand the **F# Tools** node and select **F# Interactive**.

Where lists appear in F# Interactive option arguments, list elements are separated by semicolons (`;`).

OPTION	DESCRIPTION
--	Used to instruct F# Interactive to treat remaining arguments as command line arguments to the F# program or script, which you can access in code by using the list **fsi.CommandLineArgs**.
--checked[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--codepage:<int>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--consolecolors[+|-]	Outputs warning and error messages in color.
--crossoptimize[+|-]	Enable or disable cross-module optimizations.

OPTION	DESCRIPTION
--debug[+|-] **--debug:**[**full**|**pdbonly**|**portable**|**embedded**] **-g**[+|-] **-g:**[**full**|**pdbonly**|**portable**|**embedded**]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--define:<string>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--deterministic[+|-]	Produces a deterministic assembly (including module version GUID and timestamp).
--exec	Instructs F# interactive to exit after loading the files or running the script file given on the command line.
--fullpaths	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--gui[+|-]	Enables or disables the Windows Forms event loop. The default is enabled.
--help **-?**	Used to display the command line syntax and a brief description of each option.
--lib:<folder-list> **-I:<folder-list>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--load:<filename>	Compiles the given source code at startup and loads the compiled F# constructs into the session. If the target source contains scripting directives such as **#use** or **#load**, then you must use **--use** or **#use** instead of **--load** or **#load**.
--mlcompatibility	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--noframework	Same as the **fsc.exe** compiler option. For more information, see Compiler Options
--nologo	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--nowarn:<warning-list>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--optimize[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--preferreduilang:<lang>	Specifies the preferred output language culture name (for example, es-ES, ja-JP).
--quiet	Suppress F# Interactive's output to the **stdout** stream.

OPTION	DESCRIPTION
--quotations-debug	Specifies that extra debugging information should be emitted for expressions that are derived from F# quotation literals and reflected definitions. The debug information is added to the custom attributes of an F# expression tree node. See Code Quotations and Expr.CustomAttributes.
--readline[+|-]	Enable or disable tab completion in interactive mode.
--reference:<filename> **-r:<filename>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--shadowcopyreferences[+|-]	Prevents references from being locked by the F# Interactive process.
--simpleresolution	Resolves assembly references using directory-based rules rather than MSBuild resolution.
--tailcalls[+|-]	Enable or disable the use of the tail IL instruction, which causes the stack frame to be reused for tail recursive functions. This option is enabled by default.
--targetprofile:<string>	Specifies target framework profile of this assembly. Valid values are mscorlib, netcore or netstandard. The default is mscorlib.
--use:<filename>	Tells the interpreter to use the given file on startup as initial input.
--utf8output	Same as the fsc.exe compiler option. For more information, see Compiler Options.
--warn:<warning-level>	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--warnaserror[+|-]	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.
--warnaserror[+|-]:**<int-list>**	Same as the **fsc.exe** compiler option. For more information, see Compiler Options.

Related Topics

TITLE	DESCRIPTION
Compiler Options	Describes command line options available for the F# compiler, **fsc.exe**.

Source Line, File, and Path Identifiers

7/30/2019 • 2 minutes to read • Edit Online

The identifiers `__LINE__`, `__SOURCE_DIRECTORY__` and `__SOURCE_FILE__` are built-in values that enable you to access the source line number, directory and file name in your code.

Syntax

```
__LINE__
__SOURCE_DIRECTORY__
__SOURCE_FILE__
```

Remarks

Each of these values has type `string`.

The following table summarizes the source line, file, and path identifiers that are available in F#. These identifiers are not preprocessor macros; they are built-in values that are recognized by the compiler.

PREDEFINED IDENTIFIER	DESCRIPTION
`__LINE__`	Evaluates to the current line number, considering `#line` directives.
`__SOURCE_DIRECTORY__`	Evaluates to the current full path of the source directory, considering `#line` directives.
`__SOURCE_FILE__`	Evaluates to the current source file name, without its path, considering `#line` directives.

For more information about the `#line` directive, see Compiler Directives.

Example

The following code example demonstrates the use of these values.

```
let printSourceLocation() =
    printfn "Line: %s" __LINE__
    printfn "Source Directory: %s" __SOURCE_DIRECTORY__
    printfn "Source File: %s" __SOURCE_FILE__
printSourceLocation()
```

Output:

```
Line: 4
Source Directory: C:\Users\username\Documents\Visual Studio 2017\Projects\SourceInfo\SourceInfo
Source File: Program.fs
```

See also

- Compiler Directives
- F# Language Reference

Caller information

4/9/2019 • 2 minutes to read • Edit Online

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePath	Full path of the source file that contains the caller. This is the file path at compile time.	`String`
CallerLineNumber	Line number in the source file at which the method is called.	`Integer`
CallerMemberName	Method or property name of the caller. See the Member Names section later in this topic.	`String`

Example

The following example shows how you might use these attributes to trace a caller.

```
open System.Diagnostics
open System.Runtime.CompilerServices
open System.Runtime.InteropServices

type Tracer() =
    member __.DoTrace(message: string,
                      [<CallerMemberName; Optional; DefaultParameterValue("")>] memberName: string,
                      [<CallerFilePath; Optional; DefaultParameterValue("")>] path: string,
                      [<CallerLineNumber; Optional; DefaultParameterValue(0)>] line: int) =
        Trace.WriteLine(sprintf "Message: %s" message)
        Trace.WriteLine(sprintf "Member name: %s" memberName)
        Trace.WriteLine(sprintf "Source file path: %s" path)
        Trace.WriteLine(sprintf "Source line number: %d" line)
```

Remarks

Caller Info attributes can only be applied to optional parameters. The Caller Info attributes cause the compiler to write the proper value for each optional parameter decorated with a Caller Info attribute.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that Rename Refactoring doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the CallerMemberName attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the member to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- Attributes
- Named arguments
- Optional parameters

Verbose Syntax

5/15/2019 • 2 minutes to read • Edit Online

There are two forms of syntax available for many constructs in the F# language: *verbose syntax* and *lightweight syntax*. The verbose syntax is not as commonly used, but has the advantage of being less sensitive to indentation. The lightweight syntax is shorter and uses indentation to signal the beginning and end of constructs, rather than additional keywords like `begin`, `end`, `in`, and so on. The default syntax is the lightweight syntax. This topic describes the syntax for F# constructs when lightweight syntax is not enabled. Verbose syntax is always enabled, so even if you enable lightweight syntax, you can still use verbose syntax for some constructs. You can disable lightweight syntax by using the `#light "off"` directive.

Table of Constructs

The following table shows the lightweight and verbose syntax for F# language constructs in contexts where there is a difference between the two forms. In this table, angle brackets (<>) enclose user-supplied syntax elements. Refer to the documentation for each language construct for more detailed information about the syntax used within these constructs.

LANGUAGE CONSTRUCT	LIGHTWEIGHT SYNTAX	VERBOSE SYNTAX
compound expressions	`<expression1>` `<expression2>`	`<expression1>; <expression2>`

nested `let` bindings	`let f x =`
	`  let a = 1`
	`  let b = 2`
	`  x + a + b`
		`let f x =`
		`  let a = 1 in`
		`  let b = 2 in`
		`  x + a + b`

code block	`(`
	`  <expression1>`
	`  <expression2>`
	`)`
		`begin`
		`  <expression1>;`
		`  <expression2>;`
		`end`

`for...do`	`for counter = start to`
	`finish do`
	`  ...`
		`for counter = start to`
		`finish do`
		`  ...`
		`done`

`while...do`	`while <condition> do`
	`  ...`
		`while <condition> do`
		`  ...`
		`done`

`for...in`	```
	for var in start .. finish
	do
	    ...
	``
		``
		for var in start .. finish
		do
		    ...
		done
		``

`do`	```
	do
	    ...
	``
		``
		do
		    ...
		in
		``

record	```
	type <record-name> =
	    {
	        <field-declarations>
	    }
	    <value-or-member-
	definitions>
	``
		``
		type <record-name> =
		    {
		        <field-declarations>
		    }
		    with
		        <value-or-member-
		definitions>
		    end
		``

class	```
	type <class-name>(<params>)
	=
	    ...
	``
		``
		type <class-name>(<params>)
		=
		    class
		        ...
		    end
		``

structure	```
	[<StructAttribute>]
	type <structure-name> =
	    ...
	``
		``
		type <structure-name> =
		    struct
		        ...
		    end
		``

discriminated union	```
	type <union-name> =
	    | ...
	    | ...
	    ...
	    <value-or-member
	definitions>
	``
		``
		type <union-name> =
		    | ...
		    | ...
		    ...
		    with
		        <value-or-member-
		definitions>
		    end
		``

interface	```
	type <interface-name> =
	    ...
	``
		``
		type <interface-name> =
		    interface
		        ...
		    end
		``

object expression	`{ new <type-name>`
	`  with`
	`  <value-or-member-definitions>`
	`  <interface-implementations>`
	`}`
		`{ new <type-name>`
		`  with`
		`  <value-or-member-definitions>`
		`  end`
		`  <interface-implementations>`
		`}`

interface implementation	`interface <interface-name>`
	`  with`
	`  <value-or-member-definitions>`
		`interface <interface-name>`
		`  with`
		`  <value-or-member-definitions>`
		`  end`

type extension	`type <type-name>`
	`  with`
	`  <value-or-member-definitions>`
		`type <type-name>`
		`  with`
		`  <value-or-member-definitions>`
		`  end`

module	`module <module-name> =`
	`  ...`
		`module <module-name> =`
		`  begin`
		`  ...`
		`  end`

See also

- F# Language Reference
- Compiler Directives
- Code Formatting Guidelines

F# style guide

1/4/2019 • 2 minutes to read • Edit Online

The following articles describe guidelines for formatting F# code and topical guidance for features of the language and how they should be used.

This guidance has been formulated based on the use of F# in large codebases with a diverse group of programmers. This guidance generally leads to successful use of F# and minimizes frustrations when requirements for programs change over time.

Five principles of good F# code

Keep the following principles in mind any time you write F# code, especially in systems that will change over time. Every piece of guidance in further articles stems from these five points.

1. **Good F# code is succinct, expressive, and composable**

   F# has many features that allow you to express actions in fewer lines of code and reuse generic functionality. The F# core library also contains many useful types and functions for working with common collections of data. Composition of your own functions and those in the F# core library (or other libraries) is a part of routine idiomatic F# programming. As a general rule, if you can express a solution to a problem in fewer lines of code, other developers (or your future self) will be appreciative. It's also highly recommended that you use a library such as FSharp.Core, the vast .NET libraries that F# runs on, or a third-party package on NuGet when you need to do a nontrivial task.

2. **Good F# code is interoperable**

   Interoperation can take multiple forms, including consuming code in different languages. The boundaries of your code that other callers interoperate with are critical pieces to get right, even if the callers are also in F#. When writing F#, you should always be thinking about how other code will call into the code you're writing, including if they do so from another language like C#. The F# Component Design Guidelines describe interoperability in detail.

3. **Good F# code makes use of object programming, not object orientation**

   F# has full support for programming with objects in .NET, including classes, interfaces, access modifiers, abstract classes, and so on. For more complicated functional code, such as functions that must be context-aware, objects can easily encapsulate contextual information in ways that functions cannot. Features such as optional parameters and careful use of overloading can make consumption of this functionality easier for callers.

4. **Good F# code performs well without exposing mutation**

   It's no secret that to write high-performance code, you must use mutation. It's how computers work, after all. Such code is often error-prone and difficult to get right. Avoid exposing mutation to callers. Instead, build a functional interface that hides a mutation-based implementation when performance is critical.

5. **Good F# code is toolable**

   Tools are invaluable for working in large codebases, and you can write F# code such that it can be used more effectively with F# language tooling. One example is making sure you don't overdo it with a point-free style of programming, so that intermediate values can be inspected with a debugger. Another example is using XML documentation comments describing constructs such that tooltips in editors can display those

comments at the call site. Always think about how your code will be read, navigated, debugged, and manipulated by other programmers with their tools.

Next steps

The F# code formatting guidelines provide guidance on how to format code so that it is easy to read.

The F# coding conventions provide guidance for F# programming idioms that will help the long-term maintenance of larger F# codebases.

The F# component design guidelines provide guidance for authoring F# components, such as libraries.

F# code formatting guidelines

6/11/2019 • 17 minutes to read • Edit Online

This article offers guidelines for how to format your code so that your F# code is:

- Generally viewed as more legible
- Is in accordance with conventions applied by formatting tools in Visual Studio and other editors
- Similar to other code online

These guidelines are based on A comprehensive guide to F# Formatting Conventions by Anh-Dung Phan.

General rules for indentation

F# uses significant white space by default. The following guidelines are intended to provide guidance as to how to juggle some challenges this can impose.

Using spaces

When indentation is required, you must use spaces, not tabs. At least one space is required. Your organization can create coding standards to specify the number of spaces to use for indentation; two, three or four spaces of indentation at each level where indentation occurs is typical.

We recommend 4 spaces per indentation.

That said, indentation of programs is a subjective matter. Variations are OK, but the first rule you should follow is *consistency of indentation*. Choose a generally accepted style of indentation and use it systematically throughout your codebase.

Formatting white space

F# is white space sensitive. Although most semantics from white space are covered by proper indentation, there are some other things to consider.

Formatting operators in arithmetic expressions

Always use white space around binary arithmetic expressions:

```
let subtractThenAdd x = x - 1 + 3
```

Unary `-` operators should always have the value they are negating immediately follow:

```
// OK
let negate x = -x

// Bad
let negateBad x = - x
```

Adding a white-space character after the `-` operator can lead to confusion for others.

In summary, it's important to always:

- Surround binary operators with white space
- Never have trailing white space after a unary operator

The binary arithmetic operator guideline is especially important. Failing to surround a binary `-` operator, when combined with certain formatting choices, could lead to interpreting it as a unary `-`.

Surround a custom operator definition with white space

Always use white space to surround an operator definition:

```
// OK
let ( !> ) x f = f x

// Bad
let (!>) x f = f x
```

For any custom operator that starts with `*` and that has more than one character, you need to add a white space to the beginning of the definition to avoid a compiler ambiguity. Because of this, we recommend that you simply surround the definitions of all operators with a single white-space character.

Surround function parameter arrows with white space

When defining the signature of a function, use white space around the `->` symbol:

```
// OK
type MyFun = int -> int -> string

// Bad
type MyFunBad = int->int->string
```

Surround function arguments with white space

When defining a function, use white space around each argument.

```
// OK
let myFun (a: decimal) b c = a + b + c

// Bad
let myFunBad (a:decimal)(b)c = a + b + c
```

Type annotations

Right-pad function argument type annotations

When defining arguments with type annotations, use white space after the `:` symbol:

```
// OK
let complexFunction (a: int) (b: int) c = a + b + c

// Bad
let complexFunctionBad (a :int) (b :int) (c:int) = a + b + c
```

Surround return type annotations with white space

In a let-bound function or value type annotation (return type in the case of a function), use white space before and after the `:` symbol:

```
// OK
let expensiveToCompute : int = 0 // Type annotation for let-bound value
let myFun (a: decimal) b c : decimal = a + b + c // Type annotation for the return type of a function
// Bad
let expensiveToComputeBad1:int = 1
let expensiveToComputeBad2 :int = 2
let myFunBad (a: decimal) b c:decimal = a + b + c
```

Formatting blank lines

- Separate top-level function and class definitions with two blank lines.
- Method definitions inside a class are separated by a single blank line.
- Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (for example, a set of dummy implementations).
- Use blank lines in functions, sparingly, to indicate logical sections.

Formatting comments

Generally prefer multiple double-slash comments over ML-style block comments.

```
// Prefer this style of comments when you want
// to express written ideas on multiple lines.

(*
    ML-style comments are fine, but not a .NET-ism.
    They are useful when needing to modify multi-line comments, though.
*)
```

Inline comments should capitalize the first letter.

```
let f x = x + 1 // Increment by one.
```

Naming conventions

Use camelCase for class-bound, expression-bound and pattern-bound values and functions

It is common and accepted F# style to use camelCase for all names bound as local variables or in pattern matches and function definitions.

```
// OK
let addIAndJ i j = i + j

// Bad
let addIAndJ I J = I+J

// Bad
let AddIAndJ i j = i + j
```

Locally-bound functions in classes should also use camelCase.

```
type MyClass() =

    let doSomething () =

    let firstResult = ...

    let secondResult = ...

    member x.Result = doSomething()
```

Use camelCase for module-bound public functions

When a module-bound function is part of a public API, it should use camelCase:

```
module MyAPI =
    let publicFunctionOne param1 param2 param2 = ...

    let publicFunctionTwo param1 param2 param3 = ...
```

Use camelCase for internal and private module-bound values and functions

Use camelCase for private module-bound values, including the following:

- Ad hoc functions in scripts
- Values making up the internal implementation of a module or type

```
let emailMyBossTheLatestResults =
    ...
```

Use camelCase for parameters

All parameters should use camelCase in accordance with .NET naming conventions.

```
module MyModule =
    let myFunction paramOne paramTwo = ...

type MyClass() =
    member this.MyMethod(paramOne, paramTwo) = ...
```

Use PascalCase for modules

All modules (top-level, internal, private, nested) should use PascalCase.

```
module MyTopLevelModule

module Helpers =
    module private SuperHelpers =
        ...

    ...
```

Use PascalCase for type declarations, members, and labels

Classes, interfaces, structs, enumerations, delegates, records, and discriminated unions should all be named with PascalCase. Members within types and labels for records and discriminated unions should also use PascalCase.

```
type IMyInterface =
    abstract Something: int

type MyClass() =
    member this.MyMethod(x, y) = x + y

type MyRecord = { IntVal: int; StringVal: string }

type SchoolPerson =
    | Professor
    | Student
    | Advisor
    | Administrator
```

Use PascalCase for constructs intrinsic to .NET

Namespaces, exceptions, events, and project/`.dll` names should also use PascalCase. Not only does this make

consumption from other .NET languages feel more natural to consumers, it's also consistent with .NET naming conventions that you are likely to encounter.

Avoid underscores in names

Historically, some F# libraries have used underscores in names. However, this is no longer widely accepted, partly because it clashes with .NET naming conventions. That said, some F# programmers use underscores heavily, partly for historical reasons, and tolerance and respect is important. However, be aware that the style is often disliked by others who have a choice about whether to use it.

Some exceptions includes interoperating with native components, where underscores are very common.

Use standard F# operators

The following operators are defined in the F# standard library and should be used instead of defining equivalents. Using these operators is recommended as it tends to make code more readable and idiomatic. Developers with a background in OCaml or other functional programming language may be accustomed to different idioms. The following list summarizes the recommended F# operators.

```
x |> f // Forward pipeline
f >> g // Forward composition
x |> ignore // Discard away a value
x + y // Overloaded addition (including string concatenation)
x - y // Overloaded subtraction
x * y // Overloaded multiplication
x / y // Overloaded division
x % y // Overloaded modulus
x && y // Lazy/short-cut "and"
x || y // Lazy/short-cut "or"
x <<< y // Bitwise left shift
x >>> y // Bitwise right shift
x ||| y // Bitwise or, also for working with “flags” enumeration
x &&& y // Bitwise and, also for working with “flags” enumeration
x ^^^ y // Bitwise xor, also for working with “flags” enumeration
```

Use prefix syntax for generics (`Foo<T>`) in preference to postfix syntax (`T Foo`)

F# inherits both the postfix ML style of naming generic types (for example, `int list`) as well as the prefix .NET style (for example, `list<int>`). Prefer the .NET style, except for five specific types:

1. For F# Lists, use the postfix form: `int list` rather than `list<int>`.
2. For F# Options, use the postfix form: `int option` rather than `option<int>`.
3. For F# Value Options, use the postfix form: `int voption` rather than `voption<int>`.
4. For F# arrays, use the syntactic name `int[]` rather than `int array` or `array<int>`.
5. For Reference Cells, use `int ref` rather than `ref<int>` or `Ref<int>`.

For all other types, use the prefix form.

Formatting tuples

A tuple instantiation should be parenthesized, and the delimiting commas within should be followed by a single space, for example: `(1, 2)`, `(x, y, z)`.

It is commonly accepted to omit parentheses in pattern matching of tuples:

```
let (x, y) = z // Destructuring
let x, y = z // OK

// OK
match x, y with
| 1, _ -> 0
| x, 1 -> 0
| x, y -> 1
```

It is also commonly accepted to omit parentheses if the tuple is the return value of a function:

```
// OK
let update model msg =
    match msg with
    | 1 -> model + 1, []
    | _ -> model, [ msg ]
```

In summary, prefer parenthesized tuple instantiations, but when using tuples for pattern matching or a return value, it is considered fine to avoid parentheses.

Formatting discriminated union declarations

Indent `|` in type definition by 4 spaces:

```
// OK
type Volume =
    | Liter of float
    | FluidOunce of float
    | ImperialPint of float

// Not OK
type Volume =
| Liter of float
| USPint of float
| ImperialPint of float
```

Formatting discriminated unions

Instantiated Discriminated Unions that split across multiple lines should give contained data a new scope with indentation:

```
let tree1 =
    BinaryNode
        (BinaryNode(BinaryValue 1, BinaryValue 2),
         BinaryNode(BinaryValue 3, BinaryValue 4))
```

The closing parenthesis can also be on a new line:

```
let tree1 =
    BinaryNode(
        BinaryNode(BinaryValue 1, BinaryValue 2),
        BinaryNode(BinaryValue 3, BinaryValue 4)
    )
```

Formatting record declarations

Indent `{` in type definition by 4 spaces and start the field list on the same line:

```
// OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Not OK
type PostalAddress =
  { Address: string
    City: string
    Zip: string }
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

// Unusual in F#
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    }
```

Placing the opening token on a new line and the closing token on a new line is preferable if you are declaring interface implementations or members on the record:

```
// Declaring additional members on PostalAddress
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    } with
    member x.ZipAndCity = sprintf "%s %s" x.Zip x.City

type MyRecord =
    {
        SomeField: int
    }
    interface IMyInterface
```

Formatting records

Short records can be written in one line:

```
let point = { X = 1.0; Y = 0.0 }
```

Records that are longer should use new lines for labels:

```
let rainbow =
    { Boss = "Jeffrey"
      Lackeys = ["Zippy"; "George"; "Bungle"] }
```

Placing the opening token on a new line, the contents tabbed over one scope, and the closing token on a new line is preferable if you are:

- Moving records around in code with different indentation scopes
- Piping them into a function

```
let rainbow =
    {
        Boss1 = "Jeffrey"
        Boss2 = "Jeffrey"
        Boss3 = "Jeffrey"
        Boss4 = "Jeffrey"
        Boss5 = "Jeffrey"
        Boss6 = "Jeffrey"
        Boss7 = "Jeffrey"
        Boss8 = "Jeffrey"
        Lackeys = ["Zippy"; "George"; "Bungle"]
    }

type MyRecord =
    {
        SomeField: int
    }
    interface IMyInterface

let foo a =
    a
    |> Option.map (fun x ->
        {
            MyField = x
        })
```

The same rules apply for list and array elements.

Formatting copy-and-update record expressions

A copy-and-update record expression is still a record, so similar guidelines apply.

Short expressions can fit on one line:

```
let point2 = { point with X = 1; Y = 2 }
```

Longer expressions should use new lines:

```
let rainbow2 =
    { rainbow with
        Boss = "Jeffrey"
        Lackeys = ["Zippy"; "George"; "Bungle"] }
```

And as with the record guidance, you may want to dedicate separate lines for the braces and indent one scope to the right with the expression. Note that in some special cases, such as wrapping a value with an optional without parentheses, you may need to keep a brace on one line:

```
type S = { F1: int; F2: string }
type State = { F:  S option }

let state = { F = Some { F1 = 1; F2 = "Hello" } }
let newState =
    {
        state with
            F = Some {
                    F1 = 0
                    F2 = ""
                }
    }
```

Formatting lists and arrays

Write `x :: l` with spaces around the `::` operator (`::` is an infix operator, hence surrounded by spaces).

List and arrays declared on a single line should have a space after the opening bracket and before the closing bracket:

```
let xs = [ 1; 2; 3 ]
let ys = [| 1; 2; 3; |]
```

Always use at least one space between two distinct brace-like operators. For example, leave a space between a `[` and a `{`.

```
// OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 } ]

// Not OK
[{ IngredientName = "Green beans"; Quantity = 250 }
 { IngredientName = "Pine nuts"; Quantity = 250 }
 { IngredientName = "Feta cheese"; Quantity = 250 }
 { IngredientName = "Olive oil"; Quantity = 10 }
 { IngredientName = "Lemon"; Quantity = 1 }]
```

The same guideline applies for lists or arrays of tuples.

Lists and arrays that split across multiple lines follow a similar rule as records do:

```
let pascalsTriangle =
    [|
        [| 1 |]
        [| 1; 1 |]
        [| 1; 2; 1 |]
        [| 1; 3; 3; 1 |]
        [| 1; 4; 6; 4; 1 |]
        [| 1; 5; 10; 10; 5; 1 |]
        [| 1; 6; 15; 20; 15; 6; 1 |]
        [| 1; 7; 21; 35; 35; 21; 7; 1 |]
        [| 1; 8; 28; 56; 70; 56; 28; 8; 1 |]
    |]
```

And as with records, declaring the opening and closing brackets on their own line will make moving code around and piping into functions easier.

Formatting if expressions

Indentation of conditionals depends on the sizes of the expressions that make them up. If `cond`, `e1` and `e2` are short, simply write them on one line:

```
if cond then e1 else e2
```

If either `cond`, `e1` or `e2` are longer, but not multi-line:

```
if cond
then e1
else e2
```

If any of the expressions are multi-line:

```
if cond then
    e1
else
    e2
```

Multiple conditionals with `elif` and `else` are indented at the same scope as the `if`:

```
if cond1 then e1
elif cond2 then e2
elif cond3 then e3
else e4
```

Pattern matching constructs

Use a `|` for each clause of a match with no indentation. If the expression is short, you can consider using a single line if each subexpression is also simple.

```
// OK
match l with
| { him = x; her = "Posh" } :: tail -> x
| _ :: tail -> findDavid tail
| [] -> failwith "Couldn't find David"

// Not OK
match l with
    | { him = x; her = "Posh" } :: tail -> x
    | _ :: tail -> findDavid tail
    | [] -> failwith "Couldn't find David"
```

If the expression on the right of the pattern matching arrow is too large, move it to the following line, indented one step from the `match` / `|`.

```
match lam with
| Var v -> 1
| Abs(x, body) ->
    1 + sizeLambda body
| App(lam1, lam2) ->
    sizeLambda lam1 + sizeLambda lam2
```

Pattern matching of anonymous functions, starting by `function`, should generally not indent too far. For example, indenting one scope as follows is fine:

```
lambdaList
|> List.map (function
    | Abs(x, body) -> 1 + sizeLambda 0 body
    | App(lam1, lam2) -> sizeLambda (sizeLambda 0 lam1) lam2
    | Var v -> 1)
```

Pattern matching in functions defined by `let` or `let rec` should be indented 4 spaces after starting of `let`, even

if `function` keyword is used:

```
let rec sizeLambda acc = function
    | Abs(x, body) -> sizeLambda (succ acc) body
    | App(lam1, lam2) -> sizeLambda (sizeLambda acc lam1) lam2
    | Var v -> succ acc
```

We do not recommend aligning arrows.

Formatting try/with expressions

Pattern matching on the exception type should be indented at the same level as `with`.

```
try
    if System.DateTime.Now.Second % 3 = 0 then
        raise (new System.Exception())
    else
        raise (new System.ApplicationException())
with
| :? System.ApplicationException ->
    printfn "A second that was not a multiple of 3"
| _ ->
    printfn "A second that was a multiple of 3"
```

Formatting function parameter application

In general, most function parameter application is done on the same line.

If you wish to apply parameters to a function on a new line, indent them by one scope.

```
// OK
sprintf "\t%s - %i\n\r"
     x.IngredientName x.Quantity

// OK
sprintf
     "\t%s - %i\n\r"
     x.IngredientName x.Quantity

// OK
let printVolumes x =
    printf "Volume in liters = %f, in us pints = %f, in imperial = %f"
        (convertVolumeToLiter x)
        (convertVolumeUSPint x)
        (convertVolumeImperialPint x)
```

The same guidelines apply for lambda expressions as function arguments. If the body of a lambda expression, the body can have another line, indented by one scope

```
let printListWithOffset a list1 =
    List.iter
        (fun elem -> printfn "%d" (a + elem))
        list1

// OK if lambda body is long enough
let printListWithOffset a list1 =
    List.iter
        (fun elem ->
            printfn "%d" (a + elem))
        list1
```

However, if the body of a lambda expression is more than one line, consider factoring it out into a separate function rather than have a multi-line construct applied as a single argument to a function.

Formatting infix operators

Separate operators by spaces. Obvious exceptions to this rule are the `!` and `.` operators.

Infix expressions are OK to lineup on same column:

```
acc +
(sprintf "\t%s - %i\n\r"
     x.IngredientName x.Quantity)

let function1 arg1 arg2 arg3 arg4 =
    arg1 + arg2 +
    arg3 + arg4
```

Formatting pipeline operators

Pipeline `|>` operators should go underneath the expressions they operate on.

```
// Preferred approach
let methods2 =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun assm -> assm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// Not OK
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
            |> List.ofArray
            |> List.map (fun assm -> assm.GetTypes())
            |> Array.concat
            |> List.ofArray
            |> List.map (fun t -> t.GetMethods())
            |> Array.concat
```

Formatting modules

Code in a local module must be indented relative to the module, but code in a top-level module should not be indented. Namespace elements do not have to be indented.

```
// A is a top-level module.
module A

let function1 a b = a - b * b
```

```
// A1 and A2 are local modules.
module A1 =
    let function1 a b = a*a + b*b

module A2 =
    let function2 a b = a*a - b*b
```

Formatting object expressions and interfaces

Object expressions and interfaces should be aligned in the same way with `member` being indented after 4 spaces.

```
let comparer =
    { new IComparer<string> with
          member x.Compare(s1, s2) =
              let rev (s: String) =
                  new String (Array.rev (s.ToCharArray()))
              let reversed = rev s1
              reversed.CompareTo (rev s2) }
```

Formatting white space in expressions

Avoid extraneous white space in F# expressions.

```
// OK
spam (ham.[1])

// Not OK
spam ( ham.[ 1 ] )
```

Named arguments should also not have space surrounding the `=`:

```
// OK
let makeStreamReader x = new System.IO.StreamReader(path=x)

// Not OK
let makeStreamReader x = new System.IO.StreamReader(path = x)
```

Formatting attributes

Attributes are placed above a construct:

```
[<SomeAttribute>]
type MyClass() = ...

[<RequireQualifiedAccess>]
module M =
    let f x = x

[<Struct>]
type MyRecord =
    { Label1: int
      Label2: string }
```

Formatting attributes on parameters

Attributes can also be places on parameters. In this case, place then on the same line as the parameter and before the name:

```
// Defines a class that takes an optional value as input defaulting to false.
type C() =
    member __.M([<Optional; DefaultParameterValue(false)>] doSomething: bool)
```

Formatting multiple attributes

When multiple attributes are applied to a construct that is not a parameter, they should be placed such that there is one attribute per line:

```
[<Struct>]
[<IsByRefLike>]
type MyRecord =
    { Label1: int
      Label2: string }
```

When applied to a parameter, they must be on the same line and separated by a `;` separator.

Formatting literals

F# literals using the `Literal` attribute should place the attribute on its own line and use PascalCase naming:

```
[<Literal>]
let Path = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let MyUrl = "www.mywebsitethatiamworkingwith.com"
```

Avoid placing the attribute on the same line as the value.

F# coding conventions

6/26/2019 • 22 minutes to read • Edit Online

The following conventions are formulated from experience working with large F# codebases. The Five principles of good F# code are the foundation of each recommendation. They are related to the F# component design guidelines, but are applicable for any F# code, not just components such as libraries.

Organizing code

F# features two primary ways to organize code: modules and namespaces. These are similar, but do have the following differences:

- Namespaces are compiled as .NET namespaces. Modules are compiled as static classes.
- Namespaces are always top level. Modules can be top-level and nested within other modules.
- Namespaces can span multiple files. Modules cannot.
- Modules can be decorated with `[<RequireQualifiedAccess>]` and `[<AutoOpen>]`.

The following guidelines will help you use these to organize your code.

Prefer namespaces at the top level

For any publicly consumable code, namespaces are preferential to modules at the top level. Because they are compiled as .NET namespaces, they are consumable from C# with no issue.

```
// Good!
namespace MyCode

type MyClass() =
    ...
```

Using a top-level module may not appear different when called only from F#, but for C# consumers, callers may be surprised by having to qualify `MyClass` with the `MyCode` module.

```
// Bad!
module MyCode

type MyClass() =
    ...
```

Carefully apply `[<AutoOpen>]`

The `[<AutoOpen>]` construct can pollute the scope of what is available to callers, and the answer to where something comes from is "magic". This is generally not a good thing. An exception to this rule is the F# Core Library itself (though this fact is also a bit controversial).

However, it is a convenience if you have helper functionality for a public API that you wish to organize separately from that public API.

```
module MyAPI =
    [<AutoOpen>]
    module private Helpers =
        let helper1 x y z =
            ...

    let myFunction1 x =
        let y = ...
        let z = ...

        helper1 x y z
```

This lets you cleanly separate implementation details from the public API of a function without having to fully qualify a helper each time you call it.

Additionally, exposing extension methods and expression builders at the namespace level can be neatly expressed with `[<AutoOpen>]`.

Use `[<RequireQualifiedAccess>]` whenever names could conflict or you feel it helps with readability

Adding the `[<RequireQualifiedAccess>]` attribute to a module indicates that the module may not be opened and that references to the elements of the module require explicit qualified access. For example, the `Microsoft.FSharp.Collections.List` module has this attribute.

This is useful when functions and values in the module have names that are likely to conflict with names in other modules. Requiring qualified access can greatly increase the long-term maintainability and evolvability of a library.

```
[<RequireQualifiedAccess>]
module StringTokenization =
    let parse s = ...

...

let s = getAString()
let parsed = StringTokenization.parse s // Must qualify to use 'parse'
```

Sort `open` statements topologically

In F#, the order of declarations matters, including with `open` statements. This is unlike C#, where the effect of `using` and `using static` is independent of the ordering of those statements in a file.

In F#, elements opened into a scope can shadow others already present. This means that reordering `open` statements could alter the meaning of code. As a result, any arbitrary sorting of all `open` statements (for example, alphanumerically) is generally not recommended, lest you generate different behavior that you might expect.

Instead, we recommend that you sort them topologically; that is, order your `open` statements in the order in which *layers* of your system are defined. Doing alphanumeric sorting within different topological layers may also be considered.

As an example, here is the topological sorting for the F# compiler service public API file:

```
namespace Microsoft.FSharp.Compiler.SourceCodeServices

open System
open System.Collections.Generic
open System.Collections.Concurrent
open System.Diagnostics
open System.IO
open System.Reflection
open System.Text

open Microsoft.FSharp.Compiler
open Microsoft.FSharp.Compiler.AbstractIL
open Microsoft.FSharp.Compiler.AbstractIL.Diagnostics
open Microsoft.FSharp.Compiler.AbstractIL.IL
open Microsoft.FSharp.Compiler.AbstractIL.ILBinaryReader
open Microsoft.FSharp.Compiler.AbstractIL.Internal
open Microsoft.FSharp.Compiler.AbstractIL.Internal.Library

open Microsoft.FSharp.Compiler.AccessibilityLogic
open Microsoft.FSharp.Compiler.Ast
open Microsoft.FSharp.Compiler.CompileOps
open Microsoft.FSharp.Compiler.CompileOptions
open Microsoft.FSharp.Compiler.Driver
open Microsoft.FSharp.Compiler.ErrorLogger
open Microsoft.FSharp.Compiler.Infos
open Microsoft.FSharp.Compiler.InfoReader
open Microsoft.FSharp.Compiler.Lexhelp
open Microsoft.FSharp.Compiler.Layout
open Microsoft.FSharp.Compiler.Lib
open Microsoft.FSharp.Compiler.NameResolution
open Microsoft.FSharp.Compiler.PrettyNaming
open Microsoft.FSharp.Compiler.Parser
open Microsoft.FSharp.Compiler.Range
open Microsoft.FSharp.Compiler.Tast
open Microsoft.FSharp.Compiler.Tastops
open Microsoft.FSharp.Compiler.TcGlobals
open Microsoft.FSharp.Compiler.TypeChecker
open Microsoft.FSharp.Compiler.SourceCodeServices.SymbolHelpers

open Internal.Utilities
open Internal.Utilities.Collections
```

Note that a line break separates topological layers, with each layer being sorted alphanumerically afterwards. This cleanly organizes code without accidentally shadowing values.

Use classes to contain values that have side effects

There are many times when initializing a value can have side effects, such as instantiating a context to a database or other remote resource. It is tempting to initialize such things in a module and use it in subsequent functions:

```
// This is bad!
module MyApi =
    let dep1 = File.ReadAllText "/Users/{your name}/connectionstring.txt"
    let dep2 = Environment.GetEnvironmentVariable "DEP_2"

    let private r = Random()
    let dep3() = r.Next() // Problematic if multiple threads use this

    let function1 arg = doStuffWith dep1 dep2 dep3 arg
    let function2 arg = doSutffWith dep1 dep2 dep3 arg
```

This is frequently a bad idea for a few reasons:

First, application configuration is pushed into the codebase with `dep1` and `dep2`. This is difficult to maintain in larger codebases.

Second, statically initialized data should not include values that are not thread safe if your component will itself use multiple threads. This is clearly violated by `dep3`.

Finally, module initialization compiles into a static constructor for the entire compilation unit. If any error occurs in let-bound value initialization in that module, it manifests as a `TypeInitializationException` that is then cached for the entire lifetime of the application. This can be difficult to diagnose. There is usually an inner exception that you can attempt to reason about, but if there is not, then there is no telling what the root cause is.

Instead, just use a simple class to hold dependencies:

```
type MyParametricApi(dep1, dep2, dep3) =
    member __.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1
    member __.Function2 arg2 = doStuffWith dep1 dep2 dep3 arg2
```

This enables the following:

1. Pushing any dependent state outside of the API itself.
2. Configuration can now be done outside of the API.
3. Errors in initialization for dependent values are not likely to manifest as a `TypeInitializationException`.
4. The API is now easier to test.

Error management

Error management in large systems is a complex and nuanced endeavor, and there are no silver bullets in ensuring your systems are fault-tolerant and behave well. The following guidelines should offer guidance in navigating this difficult space.

Represent error cases and illegal state in types intrinsic to your domain

With Discriminated Unions, F# gives you the ability to represent faulty program state in your type system. For example:

```
type MoneyWithdrawalResult =
    | Success of amount:decimal
    | InsufficientFunds of balance:decimal
    | CardExpired of DateTime
    | UndisclosedFailure
```

In this case, there are three known ways that withdrawing money from a bank account can fail. Each error case is represented in the type, and can thus be dealt with safely throughout the program.

```
let handleWithdrawal amount =
    let w = withdrawMoney amount
    match w with
    | Success am -> printfn "Successfully withdrew %f" am
    | InsufficientFunds balance -> printfn "Failed: balance is %f" balance
    | CardExpired expiredDate -> printfn "Failed: card expired on %O" expiredDate
    | UndisclosedFailure -> printfn "Failed: unknown"
```

In general, if you can model the different ways that something can **fail** in your domain, then error handling code is no longer treated as something you must deal with in addition to regular program flow. It is simply a part of normal program flow, and not considered **exceptional**. There are two primary benefits to this:

1. It is easier to maintain as your domain changes over time.

2. Error cases are easier to unit test.

Use exceptions when errors cannot be represented with types

Not all errors can be represented in a problem domain. These kinds of faults are *exceptional* in nature, hence the ability to raise and catch exceptions in F#.

First, it is recommended that you read the Exception Design Guidelines. These are also applicable to F#.

The main constructs available in F# for the purposes of raising exceptions should be considered in the following order of preference:

FUNCTION	SYNTAX	PURPOSE
`nullArg`	`nullArg "argumentName"`	Raises a `System.ArgumentNullException` with the specified argument name.
`invalidArg`	`invalidArg "argumentName" "message"`	Raises a `System.ArgumentException` with a specified argument name and message.
`invalidOp`	`invalidOp "message"`	Raises a `System.InvalidOperationException` with the specified message.
`raise`	`raise (ExceptionType("message"))`	General-purpose mechanism for throwing exceptions.
`failwith`	`failwith "message"`	Raises a `System.Exception` with the specified message.
`failwithf`	`failwithf "format string" argForFormatString`	Raises a `System.Exception` with a message determined by the format string and its inputs.

Use `nullArg`, `invalidArg` and `invalidOp` as the mechanism to throw `ArgumentNullException`, `ArgumentException` and `InvalidOperationException` when appropriate.

The `failwith` and `failwithf` functions should generally be avoided because they raise the base `Exception` type, not a specific exception. As per the Exception Design Guidelines, you want to raise more specific exceptions when you can.

Using exception-handling syntax

F# supports exception patterns via the `try...with` syntax:

```
try
    tryGetFileContents()
with
| :? System.IO.FileNotFoundException as e -> // Do something with it here
| :? System.Security.SecurityException as e -> // Do something with it here
```

Reconciling functionality to perform in the face of an exception with pattern matching can be a bit tricky if you wish to keep the code clean. One such way to handle this is to use active patterns as a means to group functionality surrounding an error case with an exception itself. For example, you may be consuming an API that, when it throws an exception, encloses valuable information in the exception metadata. Unwrapping a useful value in the body of the captured exception inside the Active Pattern and returning that value can be helpful in some situations.

Do not use monadic error handling to replace exceptions

Exceptions are seen as somewhat taboo in functional programming. Indeed, exceptions violate purity, so it's safe to consider them not-quite functional. However, this ignores the reality of where code must run, and that runtime errors can occur. In general, write code on the assumption that most things are neither pure nor total, to minimize unpleasant surprises.

It is important to consider the following core strengths/aspects of Exceptions with respect to their relevance and appropriateness in the .NET runtime and cross-language ecosystem as a whole:

1. They contain detailed diagnostic information, which is very helpful when debugging an issue.
2. They are well-understood by the runtime and other .NET languages.
3. They can reduce significant boilerplate when compared with code that goes out of its way to *avoid* exceptions by implementing some subset of their semantics on an ad-hoc basis.

This third point is critical. For nontrivial complex operations, failing to use exceptions can result in dealing with structures like this:

```
Result<Result<MyType, string>, string list>
```

Which can easily lead to fragile code like pattern matching on "stringly-typed" errors:

```
let result = doStuff()
match result with
| Ok r -> ...
| Error e ->
    if e.Contains "Error string 1" then ...
    elif e.Contains "Error string 2" then ...
    else ... // Who knows?
```

Additionally, it can be tempting to swallow any exception in the desire for a "simple" function that returns a "nicer" type:

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with _ -> None
```

Unfortunately, `tryReadAllText` can throw numerous exceptions based on the myriad of things that can happen on a file system, and this code discards away any information about what might actually be going wrong in your environment. If you replace this code with a result type, then you're back to "stringly-typed" error message parsing:

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Ok
    with e -> Error e.Message

let r = tryReadAllText "path-to-file"
match r with
| Ok text -> ...
| Error e ->
    if e.Contains "uh oh, here we go again..." then ...
    else ...
```

And placing the exception object itself in the `Error` constructor just forces you to properly deal with the exception type at the call site rather than in the function. Doing this effectively creates checked exceptions, which are notoriously unfun to deal with as a caller of an API.

A good alternative to the above examples is to catch *specific* exceptions and return a meaningful value in the context of that exception. If you modify the `tryReadAllText` function as follows, `None` has more meaning:

```
let tryReadAllTextIfPresent (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with :? FileNotFoundException -> None
```

Instead of functioning as a catch-all, this function will now properly handle the case when a file was not found and assign that meaning to a return. This return value can map to that error case, while not discarding any contextual information or forcing callers to deal with a case that may not be relevant at that point in the code.

Types such as `Result<'Success, 'Error>` are appropriate for basic operations where they aren't nested, and F# optional types are perfect for representing when something could either return *something* or *nothing*. They are not a replacement for exceptions, though, and should not be used in an attempt to replace exceptions. Rather, they should be applied judiciously to address specific aspects of exception and error management policy in targeted ways.

Partial application and point-free programming

F# supports partial application, and thus, various ways to program in a point-free style. This can be beneficial for code reuse within a module or the implementation of something, but it is generally not something to expose publicly. In general, point-free programming is not a virtue in and of itself, and can add a significant cognitive barrier for people who are not immersed in the style.

Do not use partial application and currying in public APIs

With little exception, the use of partial application in public APIs can be confusing for consumers. Usually, `let`-bound values in F# code are **values**, not **function values**. Mixing together values and function values can result in saving a small number of lines of code in exchange for quite a bit of cognitive overhead, especially if combined with operators such as `>>` to compose functions.

Consider the tooling implications for point-free programming

Curried functions do not label their arguments. This has tooling implications. Consider the following two functions:

```
let func name age =
    printfn "My name is %s and I am %d years old!" name age

let funcWithApplication =
    printfn "My name is %s and I am %d years old!"
```

Both are valid functions, but `funcWithApplication` is a curried function. When you hover over their types in an editor, you see this:

```
val func : name:string -> age:int -> unit

val funcWithApplication : (string -> int -> unit)
```

At the call site, tooltips in tooling such as Visual Studio will not give you meaningful information as to what the `string` and `int` input types actually represent.

If you encounter point-free code like `funcWithApplication` that is publicly consumable, it is recommended to do a full η-expansion so that tooling can pick up on meaningful names for arguments.

Furthermore, debugging point-free code can be challenging, if not impossible. Debugging tools rely on values bound to names (for example, `let` bindings) so that you can inspect intermediate values midway through

execution. When your code has no values to inspect, there is nothing to debug. In the future, debugging tools may evolve to synthesize these values based on previously executed paths, but it's not a good idea to hedge your bets on *potential* debugging functionality.

Consider partial application as a technique to reduce internal boilerplate

In contrast to the previous point, partial application is a wonderful tool for reducing boilerplate inside of an application or the deeper internals of an API. It can be helpful for unit testing the implementation of more complicated APIs, where boilerplate is often a pain to deal with. For example, the following code shows how you can accomplish what most mocking frameworks give you without taking an external dependency on such a framework and having to learn a related bespoke API.

For example, consider the following solution topography:

```
MySolution.sln
|_/ImplementationLogic.fsproj
|_/ImplementationLogic.Tests.fsproj
|_/API.fsproj
```

`ImplementationLogic.fsproj` might expose code such as:

```
module Transactions =
    let doTransaction txnContext txnType balance =
        ...

type Transactor(ctx, currentBalance) =
    member __.ExecuteTransaction(txnType) =
        Transactions.doTransaction ctx txtType currentBalance
        ...
```

Unit testing `Transactions.doTransaction` in `ImplementationLogic.Tests.fsproj` is easy:

```
namespace TransactionsTestingUtil

open Transactions

module TransactionsTestable =
    let getTestableTransactionRoutine mockContext = Transactions.doTransaction mockContext
```

Partially applying `doTransaction` with a mocking context object lets you call the function in all of your unit tests without needing to construct a mocked context each time:

```
namespace TransactionTests

open Xunit
open TransactionTypes
open TransactionsTestingUtil
open TransactionsTestingUtil.TransactionsTestable

let testableContext =
    { new ITransactionContext with
        member __.TheFirstMember() = ...
        member __.TheSecondMember() = ... }

let transactionRoutine = getTestableTransactionRoutine testableContext

[<Fact>]
let ``Test withdrawal transaction with 0.0 for balance``() =
    let expected = ...
    let actual = transactionRoutine TransactionType.Withdraw 0.0
    Assert.Equal(expected, actual)
```

This technique should not be universally applied to your entire codebase, but it is a good way to reduce boilerplate for complicated internals and unit testing those internals.

Access control

F# has multiple options for Access control, inherited from what is available in the .NET runtime. These are not just usable for types - you can use them for functions, too.

- Prefer non-`public` types and members until you need them to be publicly consumable. This also minimizes what consumers couple to.
- Strive to keep all helper functionality `private`.
- Consider the use of `[<AutoOpen>]` on a private module of helper functions if they become numerous.

Type inference and generics

Type inference can save you from typing a lot of boilerplate. And automatic generalization in the F# compiler can help you write more generic code with almost no extra effort on your part. However, these features are not universally good.

- Consider labeling argument names with explicit types in public APIs and do not rely on type inference for this.

  The reason for this is that **you** should be in control of the shape of your API, not the compiler. Although the compiler can do a fine job at inferring types for you, it is possible to have the shape of your API change if the internals it relies on have changed types. This may be what you want, but it will almost certainly result in a breaking API change that downstream consumers will then have to deal with. Instead, if you explicitly control the shape of your public API, then you can control these breaking changes. In DDD terms, this can be thought of as an Anti-corruption layer.

- Consider giving a meaningful name to your generic arguments.

  Unless you are writing truly generic code that is not specific to a particular domain, a meaningful name can help other programmers understanding the domain they're working in. For example, a type parameter named `'Document` in the context of interacting with a document database makes it clearer that generic document types can be accepted by the function or member you are working with.

- Consider naming generic type parameters with PascalCase.

This is the general way to do things in .NET, so it's recommended that you use PascalCase rather than snake_case or camelCase.

Finally, automatic generalization is not always a boon for people who are new to F# or a large codebase. There is cognitive overhead in using components that are generic. Furthermore, if automatically generalized functions are not used with different input types (let alone if they are intended to be used as such), then there is no real benefit to them being generic at that point in time. Always consider if the code you are writing will actually benefit from being generic.

Performance

F# values are immutable by default, which allows you to avoid certain classes of bugs (especially those involving concurrency and parallelism). However, in certain cases, in order to achieve optimal (or even reasonable) efficiency of execution time or memory allocations, a span of work may best be implemented by using in-place mutation of state. This is possible in an opt-in basis with F# with the `mutable` keyword.

However, use of `mutable` in F# may feel at odds with functional purity. This is fine, if you adjust expectations from purity to referential transparency. Referential transparency - not purity - is the end goal when writing F# functions. This allows you to write a functional interface over a mutation-based implementation for performance critical code.

Wrap mutable code in immutable interfaces

With referential transparency as a goal, it is critical to write code that does not expose the mutable underbelly of performance-critical functions. For example, the following code implements the `Array.contains` function in the F# core library:

```
[<CompiledName("Contains")>]
let inline contains value (array:'T[]) =
    checkNonNull "array" array
    let mutable state = false
    let mutable i = 0
    while not state && i < array.Length do
        state <- value = array.[i]
        i <- i + 1
    state
```

Calling this function multiple times does not change the underlying array, nor does it require you to maintain any mutable state in consuming it. It is referentially transparent, even though almost every line of code within it uses mutation.

Consider encapsulating mutable data in classes

The previous example used a single function to encapsulate operations using mutable data. This is not always sufficient for more complex sets of data. Consider the following sets of functions:

```
open System.Collections.Generic

let addToClosureTable (key, value) (t: Dictionary<_,_>) =
    if not (t.ContainsKey(key)) then
        t.Add(key, value)
    else
        t.[key] <- value

let closureTableCount (t: Dictionary<_,_>) = t.Count

let closureTableContains (key, value) (t: Dictionary<_, HashSet<_>>) =
    match t.TryGetValue(key) with
    | (true, v) -> v.Equals(value)
    | (false, _) -> false
```

This code is performant, but it exposes the mutation-based data structure that callers are responsible for maintaining. This can be wrapped inside of a class with no underlying members that can change:

```
open System.Collections.Generic

/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =
    let t = Dictionary<Item0, HashSet<TerminalIndex>>()

    member __.Add(key, value) =
        if not (t.ContainsKey(key)) then
            t.Add(key, value)
        else
            t.[key] <- value

    member __.Count = t.Count

    member __.Contains(key, value) =
        match t.TryGetValue(key) with
        | (true, v) -> v.Equals(value)
        | (false, _) -> false
```

`Closure1Table` encapsulates the underlying mutation-based data structure, thereby not forcing callers to maintain the underlying data structure. Classes are a powerful way to encapsulate data and routines that are mutation-based without exposing the details to callers.

Prefer `let mutable` to reference cells

Reference cells are a way to represent the reference to a value rather than the value itself. Although they can be used for performance-critical code, they are generally not recommended. Consider the following example:

```
let kernels =
    let acc = ref Set.empty

    processWorkList startKernels (fun kernel ->
        if not ((!acc).Contains(kernel)) then
            acc := (!acc).Add(kernel)
        ...)

    !acc |> Seq.toList
```

The use of a reference cell now "pollutes" all subsequent code with having to dereference and re-reference the underlying data. Instead, consider `let mutable`:

```
let kernels =
    let mutable acc = Set.empty

    processWorkList startKernels (fun kernel ->
        if not (acc.Contains(kernel)) then
            acc <- acc.Add(kernel)
        ...)

    acc |> Seq.toList
```

Aside from the single point of mutation in the middle of the lambda expression, all other code that touches `acc` can do so in a manner that is no different to the usage of a normal `let`-bound immutable value. This will make it easier to change over time.

Object programming

F# has full support for objects and object-oriented (OO) concepts. Although many OO concepts are powerful and useful, not all of them are ideal to use. The following lists offer guidance on categories of OO features at a high level.

Consider using these features in many situations:

- Dot notation (`x.Length`)
- Instance members
- Implicit constructors
- Static members
- Indexer notation (`arr.[x]`)
- Named and Optional arguments
- Interfaces and interface implementations

Don't reach for these features first, but do judiciously apply them when they are convenient to solve a problem:

- Method overloading
- Encapsulated mutable data
- Operators on types
- Auto properties
- Implementing `IDisposable` and `IEnumerable`
- Type extensions
- Events
- Structs
- Delegates
- Enums

Generally avoid these features unless you must use them:

- Inheritance-based type hierarchies and implementation inheritance
- Nulls and `Unchecked.defaultof<_>`

Prefer composition over inheritance

Composition over inheritance is a long-standing idiom that good F# code can adhere to. The fundamental principle is that you should not expose a base class and force callers to inherit from that base class to get functionality.

Use object expressions to implement interfaces if you don't need a class

Object Expressions allow you to implement interfaces on the fly, binding the implemented interface to a value without needing to do so inside of a class. This is convenient, especially if you *only* need to implement the interface and have no need for a full class.

For example, here is the code that is run in Ionide to provide a code fix action if you've added a symbol that you don't have an `open` statement for:

```
    let private createProvider () =
        { new CodeActionProvider with
            member this.provideCodeActions(doc, range, context, ct) =
                let diagnostics = context.diagnostics
                let diagnostic = diagnostics |> Seq.tryFind (fun d -> d.message.Contains "Unused open
statement")
                let res =
                    match diagnostic with
                    | None -> [||]
                    | Some d ->
                        let line = doc.lineAt d.range.start.line
                        let cmd = createEmpty<Command>
                        cmd.title <- "Remove unused open"
                        cmd.command <- "fsharp.unusedOpenFix"
                        cmd.arguments <- Some ([| doc |> unbox; line.range |> unbox; |] |> ResizeArray)
                        [|cmd |]
                res
                |> ResizeArray
                |> U2.Case1
        }
```

Because there is no need for a class when interacting with the Visual Studio Code API, Object Expressions are an ideal tool for this. They are also valuable for unit testing, when you want to stub out an interface with test routines in an ad hoc manner.

Type Abbreviations

Type Abbreviations are a convenient way to assign a label to another type, such as a function signature or a more complex type. For example, the following alias assigns a label to what's needed to define a computation with CNTK, a deep learning library:

```
open CNTK

// DeviceDescriptor, Variable, and Function all come from CNTK
type Computation = DeviceDescriptor -> Variable -> Function
```

The `Computation` name is a convenient way to denote any function that matches the signature it is aliasing. Using Type Abbreviations like this is convenient and allows for more succinct code.

Avoid using Type Abbreviations to represent your domain

Although Type Abbreviations are convenient for giving a name to function signatures, they can be confusing when abbreviating other types. Consider this abbreviation:

```
// Does not actually abstract integers.
type BufferSize = int
```

This can be confusing in multiple ways:

- `BufferSize` is not an abstraction; it is just another name for an integer.
- If `BufferSize` is exposed in a public API, it can easily be misinterpreted to mean more than just `int`. Generally, domain types have multiple attributes to them and are not primitive types like `int`. This abbreviation violates that assumption.
- The casing of `BufferSize` (PascalCase) implies that this type holds more data.
- This alias does not offer increased clarity compared with providing a named argument to a function.
- The abbreviation will not manifest in compiled IL; it is just an integer and this alias is a compile-time construct.

```
module Networking =
    ...
    let send data (bufferSize: int) =
        ...
```

In summary, the pitfall with Type Abbreviations is that they are **not** abstractions over the types they are abbreviating. In the previous example, `BufferSize` is just an `int` under the covers, with no additional data, nor any benefits from the type system besides what `int` already has.

F# component design guidelines

1/25/2019 • 27 minutes to read • Edit Online

This document is a set of component design guidelines for F# programming, based on the F# Component Design Guidelines, v14, Microsoft Research, and another version originally curated and maintained by the F# Software Foundation.

This document assumes you are familiar with F# programming. Many thanks to the F# community for their contributions and helpful feedback on various versions of this guide.

Overview

This document looks at some of the issues related to F# component design and coding. A component can mean any of the following:

- A layer in your F# project that has external consumers within that project.
- A library intended for consumption by F# code across assembly boundaries.
- A library intended for consumption by any .NET language across assembly boundaries.
- A library intended for distribution via a package repository, such as NuGet.

Techniques described in this article follow the Five principles of good F# code, and thus utilize both functional and object programming as appropriate.

Regardless of the methodology, the component and library designer faces a number of practical and prosaic issues when trying to craft an API that is most easily usable by developers. Conscientious application of the .NET Library Design Guidelines will steer you towards creating a consistent set of APIs that are pleasant to consume.

General guidelines

There are a few universal guidelines that apply to F# libraries, regardless of the intended audience for the library.

Learn the .NET Library Design Guidelines

Regardless of the kind of F# coding you are doing, it is valuable to have a working knowledge of the .NET Library Design Guidelines. Most other F# and .NET programmers will be familiar with these guidelines, and expect .NET code to conform to them.

The .NET Library Design Guidelines provide general guidance regarding naming, designing classes and interfaces, member design (properties, methods, events, etc.) and more, and are a useful first point of reference for a variety of design guidance.

Add XML documentation comments to your code

XML documentation on public APIs ensure that users can get great Intellisense and Quickinfo when using these types and members, and enable building documentation files for the library. See the XML Documentation about various xml tags that can be used for additional markup within xmldoc comments.

```
/// A class for representing (x,y) coordinates
type Point =

    /// Computes the distance between this point and another
    member DistanceTo: otherPoint:Point -> float
```

You can use either the short form XML comments (`/// comment`), or standard XML comments (`///<summary>comment</summary>`).

Consider using explicit signature files (.fsi) for stable library and component APIs

Using explicit signatures files in an F# library provides a succinct summary of public API, which both helps to ensure that you know the full public surface of your library, as well as provides a clean separation between public documentation and internal implementation details. Note that signature files add friction to changing the public API, by requiring changes to be made in both the implementation and signature files. As a result, signature files should typically only be introduced when an API has become solidified and is no longer expected to change significantly.

Always follow best practices for using strings in .NET

Follow Best Practices for Using Strings in .NET guidance. In particular, always explicitly state *cultural intent* in the conversion and comparison of strings (where applicable).

Guidelines for F#-facing libraries

This section presents recommendations for developing public F#-facing libraries; that is, libraries exposing public APIs that are intended to be consumed by F# developers. There are a variety of library-design recommendations applicable specifically to F#. In the absence of the specific recommendations that follow, the .NET Library Design Guidelines are the fallback guidance.

Naming conventions

Use .NET naming and capitalization conventions

The following table follows .NET naming and capitalization conventions. There are small additions to also include F# constructs.

CONSTRUCT	CASE	PART	EXAMPLES	NOTES
Concrete types	PascalCase	Noun/ adjective	List, Double, Complex	Concrete types are structs, classes, enumerations, delegates, records, and unions. Though type names are traditionally lowercase in OCaml, F# has adopted the .NET naming scheme for types.
DLLs	PascalCase		Fabrikam.Core.dll	
Union tags	PascalCase	Noun	Some, Add, Success	Do not use a prefix in public APIs. Optionally use a prefix when internal, such as `type Teams = TAlpha | TBeta | TDelta.`
Event	PascalCase	Verb	ValueChanged / ValueChanging	
Exceptions	PascalCase		WebException	Name should end with "Exception".

CONSTRUCT	CASE	PART	EXAMPLES	NOTES
Field	PascalCase	Noun	CurrentName	
Interface types	PascalCase	Noun/ adjective	IDisposable	Name should start with "I".
Method	PascalCase	Verb	ToString	
Namespace	PascalCase		Microsoft.FSharp.Core	Generally use `<Organization>.<Technology>[.<Subnamespace>]`, though drop the organization if the technology is independent of organization.
Parameters	camelCase	Noun	typeName, transform, range	
let values (internal)	camelCase or PascalCase	Noun/ verb	getValue, myTable	
let values (external)	camelCase or PascalCase	Noun/verb	List.map, Dates.Today	let-bound values are often public when following traditional functional design patterns. However, generally use PascalCase when the identifier can be used from other .NET languages.
Property	PascalCase	Noun/ adjective	IsEndOfFile, BackColor	Boolean properties generally use Is and Can and should be affirmative, as in IsEndOfFile, not IsNotEndOfFile.

Avoid abbreviations

The .NET guidelines discourage the use of abbreviations (for example, "use `OnButtonClick` rather than `OnBtnClick` "). Common abbreviations, such as `Async` for "Asynchronous", are tolerated. This guideline is sometimes ignored for functional programming; for example, `List.iter` uses an abbreviation for "iterate". For this reason, using abbreviations tends to be tolerated to a greater degree in F#-to-F# programming, but should still generally be avoided in public component design.

Avoid casing name collisions

The .NET guidelines say that casing alone cannot be used to disambiguate name collisions, since some client languages (for example, Visual Basic) are case-insensitive.

Use acronyms where appropriate

Acronyms such as XML are not abbreviations and are widely used in .NET libraries in uncapitalized form (Xml). Only well-known, widely recognized acronyms should be used.

Use PascalCase for generic parameter names

Do use PascalCase for generic parameter names in public APIs, including for F#-facing libraries. In particular, use names like `T`, `U`, `T1`, `T2` for arbitrary generic parameters, and when specific names make sense, then for F#-facing libraries use names like `Key`, `Value`, `Arg` (but not for example, `TKey`).

Use either PascalCase or camelCase for public functions and values in F# modules

camelCase is used for public functions that are designed to be used unqualified (for example, `invalidArg`), and for the "standard collection functions" (for example, List.map). In both these cases, the function names act much like keywords in the language.

Object, Type, and Module design

Use namespaces or modules to contain your types and modules

Each F# file in a component should begin with either a namespace declaration or a module declaration.

```
namespace Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
     ...

module CommonOperations =
    ...
```

or

```
module Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...
```

The differences between using modules and namespaces to organize code at the top level are as follows:

- Namespaces can span multiple files
- Namespaces cannot contain F# functions unless they are within an inner module
- The code for any given module must be contained within a single file
- Top-level modules can contain F# functions without the need for an inner module

The choice between a top-level namespace or module affects the compiled form of the code, and thus will affect the view from other .NET languages should your API eventually be consumed outside of F# code.

Use methods and properties for operations intrinsic to object types

When working with objects, it is best to ensure that consumable functionality is implemented as methods and properties on that type.

```
type HardwareDevice() =

    member this.ID = ...

    member this.SupportedProtocols = ...

type HashTable<'Key,'Value>(comparer: IEqualityComparer<'Key>) =

    member this.Add(key, value) = ...

    member this.ContainsKey(key) = ...

    member this.ContainsValue(value) = ...
```

The bulk of functionality for a given member need not necessarily be implemented in that member, but the consumable piece of that functionality should be.

Use classes to encapsulate mutable state

In F#, this only needs to be done where that state is not already encapsulated by another language construct, such as a closure, sequence expression, or asynchronous computation.

```
type Counter() =
    // let-bound values are private in classes.
    let mutable count = 0

    member this.Next() =
        count <- count + 1
        count
```

Use interfaces to group related operations

Use interface types to represent a set of operations. This is preferred to other options, such as tuples of functions or records of functions.

```
type Serializer =
    abstract Serialize<'T>: preserveRefEq: bool -> value: 'T -> string
    abstract Deserialize<'T>: preserveRefEq: bool -> pickle: string -> 'T
```

In preference to:

```
type Serializer<'T> = {
    Serialize: bool -> 'T -> string
    Deserialize: bool -> string -> 'T
}
```

Interfaces are first-class concepts in .NET, which you can use to achieve what Functors would normally give you. Additionally, they can be used to encode existential types into your program, which records of functions cannot.

Use a module to group functions which act on collections

When you define a collection type, consider providing a standard set of operations like `CollectionType.map` and `CollectionType.iter`) for new collection types.

```
module CollectionType =
    let map f c =
        ...
    let iter f c =
        ...
```

If you include such a module, follow the standard naming conventions for functions found in FSharp.Core.

Use a module to group functions for common, canonical functions, especially in math and DSL libraries

For example, `Microsoft.FSharp.Core.Operators` is an automatically opened collection of top-level functions (like `abs` and `sin`) provided by FSharp.Core.dll.

Likewise, a statistics library might include a module with functions `erf` and `erfc`, where this module is designed to be explicitly or automatically opened.

Consider using RequireQualifiedAccess and carefully apply AutoOpen attributes

Adding the `[<RequireQualifiedAccess>]` attribute to a module indicates that the module may not be opened and that references to the elements of the module require explicit qualified access. For example, the `Microsoft.FSharp.Collections.List` module has this attribute.

This is useful when functions and values in the module have names that are likely to conflict with names in other modules. Requiring qualified access can greatly increase the long-term maintainability and evolvability of a library.

Adding the `[<AutoOpen>]` attribute to a module means the module will be opened when the containing namespace is opened. The `[<AutoOpen>]` attribute may also be applied to an assembly to indicate a module that is automatically opened when the assembly is referenced.

For example, a statistics library **MathsHeaven.Statistics** might contain a `module MathsHeaven.Statistics.Operators` containing functions `erf` and `erfc`. It is reasonable to mark this module as `[<AutoOpen>]`. This means `open MathsHeaven.Statistics` will also open this module and bring the names `erf` and `erfc` into scope. Another good use of `[<AutoOpen>]` is for modules containing extension methods.

Overuse of `[<AutoOpen>]` leads to polluted namespaces, and the attribute should be used with care. For specific libraries in specific domains, judicious use of `[<AutoOpen>]` can lead to better usability.

Consider defining operator members on classes where using well-known operators is appropriate

Sometimes classes are used to model mathematical constructs such as Vectors. When the domain being modeled has well-known operators, defining them as members intrinsic to the class is helpful.

```
type Vector(x: float) =

    member v.X = x

    static member (*) (vector: Vector, scalar: float) = Vector(vector.X * scalar)

    static member (+) (vector1: Vector, vector2: Vector) = Vector(vector1.X + vector2.X)

let v = Vector(5.0)

let u = v * 10.0
```

This guidance corresponds to general .NET guidance for these types. However, it can be additionally important in F# coding as this allows these types to be used in conjunction with F# functions and methods with member constraints, such as List.sumBy.

Consider using CompiledName to provide a .NET-friendly name for other .NET language consumers

Sometimes you may wish to name something in one style for F# consumers (such as a static member in lower case so that it appears as if it were a module-bound function), but have a different style for the name when it is compiled into an assembly. You can use the `[<CompiledName>]` attribute to provide a different style for non F# code consuming the assembly.

```
type Vector(x:float, y:float) =

    member v.X = x
    member v.Y = y

    [<CompiledName("Create")>]
    static member create x y = Vector (x, y)

let v = Vector.create 5.0 3.0
```

By using `[<CompiledName>]`, you can use .NET naming conventions for non F# consumers of the assembly.

Use method overloading for member functions, if doing so provides a simpler API

Method overloading is a powerful tool for simplifying an API that may need to perform similar functionality, but with different options or arguments.

```
type Logger() =

    member this.Log(message) =
        ...
    member this.Log(message, retryPolicy) =
        ...
```

In F#, it is more common to overload on number of arguments rather than types of arguments.

Hide the representations of record and union types if the design of these types is likely to evolve

Avoid revealing concrete representations of objects. For example, the concrete representation of DateTime values is not revealed by the external, public API of the .NET library design. At run time, the Common Language Runtime knows the committed implementation that will be used throughout execution. However, compiled code doesn't itself pick up dependencies on the concrete representation.

Avoid the use of implementation inheritance for extensibility

In F#, implementation inheritance is rarely used. Furthermore, inheritance hierarchies are often complex and difficult to change when new requirements arrive. Inheritance implementation still exists in F# for compatibility and rare cases where it is the best solution to a problem, but alternative techniques should be sought in your F# programs when designing for polymorphism, such as interface implementation.

Function and member signatures

Use tuples for return values when returning a small number of multiple unrelated values

Here is a good example of using a tuple in a return type:

```
val divrem: BigInteger -> BigInteger -> BigInteger * BigInteger
```

For return types containing many components, or where the components are related to a single identifiable entity, consider using a named type instead of a tuple.

Use `Async<T>` for async programming at F# API boundaries

If there is a corresponding synchronous operation named `Operation` that returns a `T`, then the async operation should be named `AsyncOperation` if it returns `Async<T>` or `OperationAsync` if it returns `Task<T>`. For commonly used .NET types that expose Begin/End methods, consider using `Async.FromBeginEnd` to write extension methods as a façade to provide the F# async programming model to those .NET APIs.

```
type SomeType =
    member this.Compute(x:int): int =
        ...
    member this.AsyncCompute(x:int): Async<int> =
        ...

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        ...
```

Exceptions

See Error Management to learn about appropriate use of exceptions, results, and options.

Extension Members

Carefully apply F# extension members in F#-to-F# components

F# extension members should generally only be used for operations that are in the closure of intrinsic operations associated with a type in the majority of its modes of use. One common use is to provide APIs that are more idiomatic to F# for various .NET types:

```
type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

type System.Collections.Generic.IDictionary<'Key,'Value> with
    member this.TryGet key =
        let ok, v = this.TryGetValue key
        if ok then Some v else None
```

Union Types

Use discriminated unions instead of class hierarchies for tree-structured data

Tree-like structures are recursively defined. This is awkward with inheritance, but elegant with Discriminated Unions.

```
type BST<'T> =
    | Empty
    | Node of 'T * BST<'T> * BST<'T>
```

Representing tree-like data with Discriminated Unions also allows you to benefit from exhaustiveness in pattern matching.

Use `[<RequireQualifiedAccess>]` on union types whose case names are not sufficiently unique

You may find yourself in a domain where the same name is the best name for different things, such as Discriminated Union cases. You can use `[<RequireQualifiedAccess>]` to disambiguate case names in order to avoid triggering confusing errors due to shadowing dependent on the ordering of `open` statements

Hide the representations of discriminated unions for binary compatible APIs if the design of these types is likely to evolve

Unions types rely on F# pattern-matching forms for a succinct programming model. As mentioned previously, you should avoid revealing concrete data representations if the design of these types is likely to evolve.

For example, the representation of a discriminated union can be hidden using a private or internal declaration, or by using a signature file.

```
type Union =
    private
    | CaseA of int
    | CaseB of string
```

If you reveal discriminated unions indiscriminately, you may find it hard to version your library without breaking user code. Instead, consider revealing one or more active patterns to permit pattern matching over values of your type.

Active patterns provide an alternate way to provide F# consumers with pattern matching while avoiding exposing F# Union Types directly.

Inline Functions and Member Constraints

Define generic numeric algorithms using inline functions with implied member constraints and statically resolved generic types

Arithmetic member constraints and F# comparison constraints are a standard for F# programming. For example, consider the following code:

```
let inline highestCommonFactor a b =
    let rec loop a b =
        if a = LanguagePrimitives.GenericZero<_> then b
        elif a < b then loop a (b - a)
        else loop (a - b) b
    loop a b
```

The type of this function is as follows:

```
val inline highestCommonFactor : ^T -> ^T -> ^T
                when ^T : (static member Zero : ^T)
                and ^T : (static member ( - ) : ^T * ^T -> ^T)
                and ^T : equality
                and ^T : comparison
```

This is a suitable function for a public API in a mathematical library.

Avoid using member constraints to simulate type classes and duck typing

It is possible to simulate "duck typing" using F# member constraints. However, members that make use of this should not in general be used in F#-to-F# library designs. This is because library designs based on unfamiliar or non-standard implicit constraints tend to cause user code to become inflexible and tied to one particular framework pattern.

Additionally, there is a good chance that heavy use of member constraints in this manner can result in very long compile times.

Operator Definitions

Avoid defining custom symbolic operators

Custom operators are essential in some situations and are highly useful notational devices within a large body of implementation code. For new users of a library, named functions are often easier to use. In addition, custom symbolic operators can be hard to document, and users find it more difficult to look up help on operators, due to existing limitations in IDE and search engines.

As a result, it is best to publish your functionality as named functions and members, and additionally expose operators for this functionality only if the notational benefits outweigh the documentation and cognitive cost of having them.

Units of Measure

Carefully use units of measure for added type safety in F# code

Additional typing information for units of measure is erased when viewed by other .NET languages. Be aware that .NET components, tools, and reflection will see types-sans-units. For example, C# consumers will see `float` rather than `float<kg>`.

Type Abbreviations

Carefully use type abbreviations to simplify F# code

.NET components, tools, and reflection will not see abbreviated names for types. Significant usage of type abbreviations can also make a domain appear more complex than it actually is, which could confuse consumers.

Avoid type abbreviations for public types whose members and properties should be intrinsically different to those available on the type being abbreviated

In this case, the type being abbreviated reveals too much about the representation of the actual type being defined. Instead, consider wrapping the abbreviation in a class type or a single-case discriminated union (or, when performance is essential, consider using a struct type to wrap the abbreviation).

For example, it is tempting to define a multi-map as a special case of an F# map, for example:

```
type MultiMap<'Key,'Value> = Map<'Key,'Value list>
```

However, the logical dot-notation operations on this type are not the same as the operations on a Map – for example, it is reasonable that the lookup operator map.[key] return the empty list if the key is not in the dictionary, rather than raising an exception.

Guidelines for libraries for Use from other .NET Languages

When designing libraries for use from other .NET languages, it is important to adhere to the .NET Library Design Guidelines. In this document, these libraries are labeled as vanilla .NET libraries, as opposed to F#-facing libraries that use F# constructs without restriction. Designing vanilla .NET libraries means providing familiar and idiomatic APIs consistent with the rest of the .NET Framework by minimizing the use of F#-specific constructs in the public API. The rules are explained in the following sections.

Namespace and Type design (for libraries for use from other .NET Languages)

Apply the .NET naming conventions to the public API of your components

Pay special attention to the use of abbreviated names and the .NET capitalization guidelines.

```
type pCoord = ...
    member this.theta = ...

type PolarCoordinate = ...
    member this.Theta = ...
```

Use namespaces, types, and members as the primary organizational structure for your components

All files containing public functionality should begin with a `namespace` declaration, and the only public-facing entities in namespaces should be types. Do not use F# modules.

Use non-public modules to hold implementation code, utility types, and utility functions.

Static types should be preferred over modules, as they allow for future evolution of the API to use overloading and other .NET API design concepts that may not be used within F# modules.

For example, in place of the following public API:

```
module Fabrikam

module Utilities =
    let Name = "Bob"
    let Add2 x y = x + y
    let Add3 x y z = x + y + z
```

Consider instead:

```
namespace Fabrikam

[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
```

Use F# record types in vanilla .NET APIs if the design of the types won't evolve

F# record types compile to a simple .NET class. These are suitable for some simple, stable types in APIs. You should consider using the `[<NoEquality>]` and `[<NoComparison>]` attributes to suppress the automatic generation of interfaces. Also avoid using mutable record fields in vanilla .NET APIs as these exposes a public field. Always consider whether a class would provide a more flexible option for future evolution of the API.

For example, the following F# code exposes the public API to a C# consumer:

F#:

```
[<NoEquality; NoComparison>]
type MyRecord =
    { FirstThing: int
        SecondThing: string }
```

C#:

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

Hide the representation of F# union types in vanilla .NET APIs

F# union types are not commonly used across component boundaries, even for F#-to-F# coding. They are an excellent implementation device when used internally within components and libraries.

When designing a vanilla .NET API, consider hiding the representation of a union type by using either a private declaration or a signature file.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True
```

You may also augment types that use a union representation internally with members to provide a desired .NET-facing API.

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

    /// A public member for use from C#
    member x.Evaluate =
        match x with
        | And(a,b) -> a.Evaluate && b.Evaluate
        | Not a -> not a.Evaluate
        | True -> true

    /// A public member for use from C#
    static member CreateAnd(a,b) = And(a,b)
```

Design GUI and other components using the design patterns of the framework

There are many different frameworks available within .NET, such as WinForms, WPF, and ASP.NET. Naming and design conventions for each should be used if you are designing components for use in these frameworks. For example, for WPF programming, adopt WPF design patterns for the classes you are designing. For models in user interface programming, use design patterns such as events and notification-based collections such as those found in System.Collections.ObjectModel.

Object and Member design (for libraries for use from other .NET Languages)

Use the CLIEvent attribute to expose .NET events

Construct a `DelegateEvent` with a specific .NET delegate type that takes an object and `EventArgs` (rather than an `Event`, which just uses the `FSharpHandler` type by default) so that the events are published in the familiar way to other .NET languages.

```
type MyBadType() =
    let myEv = new Event<int>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

type MyEventArgs(x: int) =
    inherit System.EventArgs()
    member this.X = x

    /// A type in a component designed for use from other .NET languages
type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

Expose asynchronous operations as methods which return .NET tasks

Tasks are used in .NET to represent active asynchronous computations. Tasks are in general less compositional than F# `Async<T>` objects, since they represent "already executing" tasks and can't be composed together in ways that perform parallel composition, or which hide the propagation of cancellation signals and other contextual parameters.

However, despite this, methods which return Tasks are the standard representation of asynchronous programming on .NET.

```
/// A type in a component designed for use from other .NET languages
type MyType() =

    let compute (x: int): Async<int> = async { ... }

    member this.ComputeAsync(x) = compute x |> Async.StartAsTask
```

You will frequently also want to accept an explicit cancellation token:

```
/// A type in a component designed for use from other .NET languages
type MyType() =
    let compute(x: int): Async<int> = async { ... }
    member this.ComputeAsTask(x, cancellationToken) = Async.StartAsTask(compute x, cancellationToken)
```

Use .NET delegate types instead of F# function types

Here "F# function types" mean "arrow" types like `int -> int`.

Instead of this:

```
member this.Transform(f: int->int) =
    ...
```

Do this:

```
member this.Transform(f: Func<int,int>) =
    ...
```

The F# function type appears as `class FSharpFunc<T,U>` to other .NET languages, and is less suitable for language features and tooling that understand delegate types. When authoring a higher-order method targeting .NET Framework 3.5 or higher, the `System.Func` and `System.Action` delegates are the right APIs to publish to enable .NET developers to consume these APIs in a low-friction manner. (When targeting .NET Framework 2.0, the system-defined delegate types are more limited; consider using predefined delegate types such as `System.Converter<T,U>` or defining a specific delegate type.)

On the flip side, .NET delegates are not natural for F#-facing libraries (see the next Section on F#-facing libraries). As a result, a common implementation strategy when developing higher-order methods for vanilla .NET libraries is to author all the implementation using F# function types, and then create the public API using delegates as a thin façade atop the actual F# implementation.

Use the TryGetValue pattern instead of returning F# option values, and prefer method overloading to taking F# option values as arguments

Common patterns of use for the F# option type in APIs are better implemented in vanilla .NET APIs using standard .NET design techniques. Instead of returning an F# option value, consider using the bool return type plus an out parameter as in the "TryGetValue" pattern. And instead of taking F# option values as parameters, consider using method overloading or optional arguments.

```
member this.ReturnOption() = Some 3

member this.ReturnBoolAndOut(outVal: byref<int>) =
    outVal <- 3
    true

member this.ParamOption(x: int, y: int option) =
    match y with
    | Some y2 -> x + y2
    | None -> x

member this.ParamOverload(x: int) = x

member this.ParamOverload(x: int, y: int) = x + y
```

Use the .NET collection interface types IEnumerable<T> and IDictionary<Key,Value> for parameters and return values

Avoid the use of concrete collection types such as .NET arrays `T[]`, F# types `list<T>`, `Map<Key,Value>` and `Set<T>`, and .NET concrete collection types such as `Dictionary<Key,Value>`. The .NET Library Design Guidelines have good advice regarding when to use various collection types like `IEnumerable<T>`. Some use of arrays (`T[]`) is acceptable in some circumstances, on performance grounds. Note especially that `seq<T>` is just the F# alias for `IEnumerable<T>`, and thus seq is often an appropriate type for a vanilla .NET API.

Instead of F# lists:

```
member this.PrintNames(names: string list) =
    ...
```

Use F# sequences:

```
member this.PrintNames(names: seq<string>) =
    ...
```

Use the unit type as the only input type of a method to define a zero-argument method, or as the only return type to define a void-returning method

Avoid other uses of the unit type. These are good:

```
✔ member this.NoArguments() = 3

✔ member this.ReturnVoid(x: int) = ()
```

This is bad:

```
member this.WrongUnit( x: unit, z: int) = ((), ())
```

Check for null values on vanilla .NET API boundaries

F# implementation code tends to have fewer null values, due to immutable design patterns and restrictions on use of null literals for F# types. Other .NET languages often use null as a value much more frequently. Because of this, F# code that is exposing a vanilla .NET API should check parameters for null at the API boundary, and prevent these values from flowing deeper into the F# implementation code. The `isNull` function or pattern matching on the `null` pattern can be used.

```
let checkNonNull argName (arg: obj) =
    match arg with
    | null -> nullArg argName
    | _ -> ()

let checkNonNull` argName (arg: obj) =
    if isNull arg then nullArg argName
    else ()
```

Avoid using tuples as return values

Instead, prefer returning a named type holding the aggregate data, or using out parameters to return multiple values. Although tuples and struct tuples exist in .NET (including C# language support for struct tuples), they will most often not provide the ideal and expected API for .NET developers.

Avoid the use of currying of parameters

Instead, use .NET calling conventions `Method(arg1,arg2,…,argN)`.

```
member this.TupledArguments(str, num) = String.replicate num str
```

Tip: If you're designing libraries for use from any .NET language, then there's no substitute for actually doing some experimental C# and Visual Basic programming to ensure that your libraries "feel right" from these languages. You can also use tools such as .NET Reflector and the Visual Studio Object Browser to ensure that libraries and their documentation appear as expected to developers.

Appendix

End-to-end example of designing F# code for use by other .NET languages

Consider the following class:

```
open System

type Point1(angle,radius) =
    new() = Point1(angle=0.0, radius=0.0)
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
```

The inferred F# type of this class is as follows:

```
type Point1 =
    new : unit -> Point1
    new : angle:double * radius:double -> Point1
    static member Circle : n:int -> Point1 list
    member Stretch : l:double -> Point1
    member Warp : f:(double -> double) -> Point1
    member Angle : double
    member Radius : double
```

Let's take a look at how this F# type appears to a programmer using another .NET language. For example, the approximate C# "signature" is as follows:

```
// C# signature for the unadjusted Point1 class
public class Point1
{
    public Point1();

    public Point1(double angle, double radius);

    public static Microsoft.FSharp.Collections.List<Point1> Circle(int count);

    public Point1 Stretch(double factor);

    public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

There are some important points to notice about how F# represents constructs here. For example:

- Metadata such as argument names has been preserved.
- F# methods that take two arguments become C# methods that take two arguments.
- Functions and lists become references to corresponding types in the F# library.

The following code shows how to adjust this code to take these things into account.

```
namespace SuperDuperFSharpLibrary.Types

type RadialPoint(angle:double, radius:double) =

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,_>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
                RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }
```

The inferred F# type of the code is as follows:

```
type RadialPoint =
    new : unit -> RadialPoint
    new : angle:double * radius:double -> RadialPoint
    static member Circle : count:int -> seq<RadialPoint>
    member Stretch : factor:double -> RadialPoint
    member Warp : transform:System.Func<double,double> -> RadialPoint
    member Angle : double
    member Radius : double
```

The C# signature is now as follows:

```
public class RadialPoint
{
    public RadialPoint();

    public RadialPoint(double angle, double radius);

    public static System.Collections.Generic.IEnumerable<RadialPoint> Circle(int count);

    public RadialPoint Stretch(double factor);

    public RadialPoint Warp(System.Func<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

The fixes made to prepare this type for use as part of a vanilla .NET library are as follows:

- Adjusted several names: `Point1`, `n`, `l`, and `f` became `RadialPoint`, `count`, `factor`, and `transform`, respectively.
- Used a return type of `seq<RadialPoint>` instead of `RadialPoint list` by changing a list construction using `[ ... ]` to a sequence construction using `IEnumerable<RadialPoint>`.
- Used the .NET delegate type `System.Func` instead of an F# function type.

This makes it far nicer to consume in C# code.

Using F# on Azure

4/8/2019 • 4 minutes to read • Edit Online

F# is a superb language for cloud programming and is frequently used to write web applications, cloud services, cloud-hosted microservices, and for scalable data processing.

In the following sections, you will find resources on how to use a range of Azure services with F#.

> **NOTE**
> If a particular Azure service isn't in this documentation set, please consult either the Azure Functions or .NET documentation for that service. Some Azure services are language-independent and require no language-specific documentation and are not listed here.

Using Azure Virtual Machines with F#

Azure supports a wide range of virtual machine (VM) configurations, see Linux and Azure Virtual Machines.

To install F# on a virtual machine for execution, compilation and/or scripting see Using F# on Linux and Using F# on Windows.

Using Azure Functions with F#

Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Your functions are connected to events in Azure storage and other cloud-hosted resources. Data flows into your F# functions via function arguments. You can use your development language of choice, trusting Azure to scale as needed.

Azure Functions support F# as a first-class language with efficient, reactive, scalable execution of F# code. See the Azure Functions F# Developer Reference for reference documentation on how to use F# with Azure Functions.

Other resources for using Azure Functions and F#:

- Scale Up Azure Functions in F# Using Suave
- How to create Azure function in F#
- Using the Azure Type Provider with Azure Functions

Using Azure Storage with F#

Azure Storage is a base layer of storage services for modern applications that rely on durability, availability, and scalability to meet the needs of customers. F# programs can interact directly with Azure storage services, using the techniques described in the following articles.

- Get started with Azure Blob storage using F#
- Get started with Azure File storage using F#
- Get started with Azure Queue storage using F#
- Get started with Azure Table storage using F#

Azure Storage can also be used in conjunction with Azure Functions through declarative configuration rather than explicit API calls. See Azure Functions triggers and bindings for Azure Storage which includes F# examples.

Using Azure App Service with F#

Azure App Service is a cloud platform to build powerful web and mobile apps that connect to data anywhere, in the cloud or on-premises.

- F# Azure Web API example
- Hosting F# in a web application on Azure

Using Apache Spark with F# with Azure HDInsight

Apache Spark for Azure HDInsight is an open source processing framework that runs large-scale data analytics applications. Azure makes Apache Spark easy and cost effective to deploy. Develop your Spark application in F# using Mobius, a .NET API for Spark.

- Implementing Spark Apps in F# using Mobius
- Example F# Spark Apps using Mobius

Using Azure Cosmos DB with F#

Azure Cosmos DB is a NoSQL service for highly available, globally distributed apps.

Azure Cosmos DB can be used with F# in two ways:

1. Through the creation of F# Azure Functions which react to or cause changes to Azure Cosmos DB collections. See Azure Cosmos DB bindings for Azure Functions, or
2. By using the Azure Cosmos DB .NET SDK for SQL API. The related samples are in C#.

Using Azure Event Hubs with F#

Azure Event Hubs provide cloud-scale telemetry ingestion from websites, apps, and devices.

Azure Event Hubs can be used with F# in two ways:

1. Through the creation of F# Azure Functions which are triggered by events. See Azure Function triggers for Event Hubs, or
2. By using the .NET SDK for Azure. Note these examples are in C#.

Using Azure Notification Hubs with F#

Azure Notification Hubs are multiplatform, scaled-out push infrastructure that enable you to send mobile push notifications from any backend (in the cloud or on-premises) to any mobile platform.

Azure Notification Hubs can be used with F# in two ways:

1. Through the creation of F# Azure Functions which send results to a notification hub. See Azure Function output triggers for Notification Hubs, or
2. By using the .NET SDK for Azure. Note these examples are in C#.

Implementing WebHooks on Azure with F#

A Webhook is a callback triggered via a web request. Webhooks are used by sites such as GitHub to signal events.

Webhooks can be implemented in F# and hosted on Azure via an Azure Function in F# with a Webhook Binding.

Using Webjobs with F#

Webjobs are programs you can run in your App Service web app in three ways: on demand, continuously, or on a

schedule.

Example F# Webjob

Implementing Timers on Azure with F#

Timer triggers call functions based on a schedule, one time or recurring.

Timers can be implemented in F# and hosted on Azure via an Azure Function in F# with a Timer Trigger.

Deploying and Managing Azure Resources with F# Scripts

Azure VMs may be programmatically deployed and managed from F# scripts by using the Microsoft.Azure.Management packages and APIs. For example, see Get Started with the Management Libraries for .NET and Using Azure Resource Manager.

Likewise, other Azure resources may also be deployed and managed from F# scripts using the same components. For example, you can create storage accounts, deploy Azure Cloud Services, create Azure Cosmos DB instances and manage Azure Notifcation Hubs programmatically from F# scripts.

Using F# scripts to deploy and manage resources is not normally necessary. For example, Azure resources may also be deployed directly from JSON template descriptions, which can be parameterized. See Azure Resource Manager Templates including examples such as the Azure Quickstart Templates.

Other resources

- Full documentation on all Azure services

Get started with Azure Blob storage using F#

7/30/2019 • 12 minutes to read • Edit Online

Azure Blob storage is a service that stores unstructured data in the cloud as objects/blobs. Blob storage can store any type of text or binary data, such as a document, media file, or application installer. Blob storage is also referred to as object storage.

This article shows you how to perform common tasks using Blob storage. The samples are written using F# using the Azure Storage Client Library for .NET. The tasks covered include how to upload, list, download, and delete blobs.

For a conceptual overview of blob storage, see the .NET guide for blob storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You also need your storage access key for this account.

Create an F# Script and Start F# Interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `blobs.fsx`, in your F# development environment.

Next, use a package manager such as Paket or NuGet to install the `WindowsAzure.Storage` and `Microsoft.WindowsAzure.ConfigurationManager` packages and reference `WindowsAzure.Storage.dll` and `Microsoft.WindowsAzure.Configuration.dll` in your script using a `#r` directive.

Add namespace declarations

Add the following `open` statements to the top of the `blobs.fsx` file:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Blob // Namespace for Blob storage types
```

Get your connection string

You need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

However, this is **not recommended** for real projects. Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. You can regenerate your key using the Azure Portal if you believe it may have been compromised.

For real applications, the best way to maintain your storage connection string is in a configuration file. To fetch the connection string from a configuration file, you can do this:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Using Azure Configuration Manager is optional. You can also use an API such as the .NET Framework's `ConfigurationManager` type.

Parse the connection string

To parse the connection string, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

This returns a `CloudStorageAccount`.

Create some local dummy data

Before you begin, create some dummy local data in the directory of our script. Later you upload this data.

```
// Create a dummy file to upload
let localFile = __SOURCE_DIRECTORY__ + "/myfile.txt"
File.WriteAllText(localFile, "some data")
```

Create the Blob service client

The `CloudBlobClient` type enables you to retrieve containers and blobs stored in Blob storage. Here's one way to create the service client:

```
let blobClient = storageAccount.CreateCloudBlobClient()
```

Now you are ready to write code that reads data from and writes data to Blob storage.

Create a container

This example shows how to create a container if it does not already exist:

```
 // Retrieve a reference to a container.
let container = blobClient.GetContainerReference("mydata")

// Create the container if it doesn't already exist.
container.CreateIfNotExists()
```

By default, the new container is private, meaning that you must specify your storage access key to download blobs from this container. If you want to make the files within the container available to everyone, you can set the container to be public using the following code:

```
let permissions = BlobContainerPermissions(PublicAccess=BlobContainerPublicAccessType.Blob)
container.SetPermissions(permissions)
```

Anyone on the Internet can see blobs in a public container, but you can modify or delete them only if you have the appropriate account access key or a shared access signature.

Upload a blob into a container

Azure Blob Storage supports block blobs and page blobs. In most cases, a block blob is the recommended type to use.

To upload a file to a block blob, get a container reference and use it to get a block blob reference. Once you have a blob reference, you can upload any stream of data to it by calling the `UploadFromFile` method. This operation creates the blob if it didn't previously exist, or overwrite it if it does exist.

```
// Retrieve reference to a blob named "myblob.txt".
let blockBlob = container.GetBlockBlobReference("myblob.txt")

// Create or overwrite the "myblob.txt" blob with contents from the local file.
do blockBlob.UploadFromFile(localFile)
```

List the blobs in a container

To list the blobs in a container, first get a container reference. You can then use the container's `ListBlobs` method to retrieve the blobs and/or directories within it. To access the rich set of properties and methods for a returned `IListBlobItem`, you must cast it to a `CloudBlockBlob`, `CloudPageBlob`, or `CloudBlobDirectory` object. If the type is unknown, you can use a type check to determine which to cast it to. The following code demonstrates how to retrieve and output the URI of each item in the `mydata` container:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, false) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %O" blob.Properties.Length blob.Uri

    | :? CloudPageBlob as pageBlob ->
        printfn "Page blob of length %d: %O" pageBlob.Properties.Length pageBlob.Uri

    | :? CloudBlobDirectory as directory ->
        printfn "Directory: %O" directory.Uri

    | _ ->
        printfn "Unknown blob type: %O" (item.GetType())
```

You can also name blobs with path information in their names. This creates a virtual directory structure that you can organize and traverse as you would a traditional file system. Note that the directory structure is virtual only - the only resources available in Blob storage are containers and blobs. However, the storage client library offers a `CloudBlobDirectory` object to refer to a virtual directory and simplify the process of working with blobs that are organized in this way.

For example, consider the following set of block blobs in a container named `photos`:

photo1.jpg
2015/architecture/description.txt
2015/architecture/photo3.jpg
2015/architecture/photo4.jpg
2016/architecture/photo5.jpg
2016/architecture/photo6.jpg
2016/architecture/description.txt
2016/photo7.jpg\

When you call `ListBlobs` on a container (as in the above sample), a hierarchical listing is returned. If it contains both `CloudBlobDirectory` and `CloudBlockBlob` objects, representing the directories and blobs in the container, respectively, then the resulting output looks similar to this:

```
Directory: https://<accountname>.blob.core.windows.net/photos/2015/
Directory: https://<accountname>.blob.core.windows.net/photos/2016/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Optionally, you can set the `UseFlatBlobListing` parameter of the `ListBlobs` method to `true`. In this case, every blob in the container is returned as a `CloudBlockBlob` object. The call to `ListBlobs` to return a flat listing looks like this:

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, true) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %O" blob.Properties.Length blob.Uri

    | _ ->
        printfn "Unexpected blob type: %O" (item.GetType())
```

and, depending on the current contents of your container, the results look like this:

```
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2015/architecture/description.txt
Block blob of length 314618: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo3.jpg
Block blob of length 522713: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo4.jpg
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2016/architecture/description.txt
Block blob of length 419048: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo5.jpg
Block blob of length 506388: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo6.jpg
Block blob of length 399751: https://<accountname>.blob.core.windows.net/photos/2016/photo7.jpg
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

Download blobs

To download blobs, first retrieve a blob reference and then call the `DownloadToStream` method. The following example uses the `DownloadToStream` method to transfer the blob contents to a stream object that you can then persist to a local file.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDownload = container.GetBlockBlobReference("myblob.txt")

// Save blob contents to a file.
do
    use fileStream = File.OpenWrite(__SOURCE_DIRECTORY__ + "/path/download.txt")
    blobToDownload.DownloadToStream(fileStream)
```

You can also use the `DownloadToStream` method to download the contents of a blob as a text string.

```
let text =
    use memoryStream = new MemoryStream()
    blobToDownload.DownloadToStream(memoryStream)
    Text.Encoding.UTF8.GetString(memoryStream.ToArray())
```

Delete blobs

To delete a blob, first get a blob reference and then call the `Delete` method on it.

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDelete = container.GetBlockBlobReference("myblob.txt")

// Delete the blob.
blobToDelete.Delete()
```

List blobs in pages asynchronously

If you are listing a large number of blobs, or you want to control the number of results you return in one listing operation, you can list blobs in pages of results. This example shows how to return results in pages asynchronously, so that execution is not blocked while waiting to return a large set of results.

This example shows a flat blob listing, but you can also perform a hierarchical listing, by setting the `useFlatBlobListing` parameter of the `ListBlobsSegmentedAsync` method to `false`.

The sample defines an asynchronous method, using an `async` block. The `let!` keyword suspends execution of the sample method until the listing task completes.

```
let ListBlobsSegmentedInFlatListing(container:CloudBlobContainer) =
    async {

        // List blobs to the console window, with paging.
        printfn "List blobs in pages:"

        // Call ListBlobsSegmentedAsync and enumerate the result segment
        // returned, while the continuation token is non-null.
        // When the continuation token is null, the last page has been
        // returned and execution can exit the loop.

        let rec loop continuationToken (i:int) =
            async {
                let! ct = Async.CancellationToken
                // This overload allows control of the page size. You can return
                // all remaining results by passing null for the maxResults
                // parameter, or by calling a different overload.
                let! resultSegment =
                    container.ListBlobsSegmentedAsync(
                        "", true, BlobListingDetails.All, Nullable 10,
                        continuationToken, null, null, ct)
                    |> Async.AwaitTask

                if (resultSegment.Results |> Seq.length > 0) then
                    printfn "Page %d:" i

                for blobItem in resultSegment.Results do
                    printfn "\t%O" blobItem.StorageUri.PrimaryUri

                printfn ""

                // Get the continuation token.
                let continuationToken = resultSegment.ContinuationToken
                if (continuationToken <> null) then
                    do! loop continuationToken (i+1)
            }

        do! loop null 1
    }
```

We can now use this asynchronous routine as follows. First you upload some dummy data (using the local file created earlier in this tutorial).

```
// Create some dummy data by uploading the same file over and over again
for i in 1 .. 100 do
    let blob  = container.GetBlockBlobReference("myblob" + string i + ".txt")
    use fileStream = System.IO.File.OpenRead(localFile)
    blob.UploadFromFile(localFile)
```

Now, call the routine. You use `Async.RunSynchronously` to force the execution of the asynchronous operation.

```
ListBlobsSegmentedInFlatListing container |> Async.RunSynchronously
```

Writing to an append blob

An append blob is optimized for append operations, such as logging. Like a block blob, an append blob is comprised of blocks, but when you add a new block to an append blob, it is always appended to the end of the blob. You cannot update or delete an existing block in an append blob. The block IDs for an append blob are not exposed as they are for a block blob.

Each block in an append blob can be a different size, up to a maximum of 4 MB, and an append blob can include a maximum of 50,000 blocks. The maximum size of an append blob is therefore slightly more than 195 GB (4 MB X 50,000 blocks).

The following example creates a new append blob and appends some data to it, simulating a simple logging operation.

```
// Get a reference to a container.
let appendContainer = blobClient.GetContainerReference("my-append-blobs")

// Create the container if it does not already exist.
appendContainer.CreateIfNotExists() |> ignore

// Get a reference to an append blob.
let appendBlob = appendContainer.GetAppendBlobReference("append-blob.log")

// Create the append blob. Note that if the blob already exists, the
// CreateOrReplace() method will overwrite it. You can check whether the
// blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateOrReplace()

let numBlocks = 10

// Generate an array of random bytes.
let rnd = new Random()
let bytes = Array.zeroCreate<byte>(numBlocks)
rnd.NextBytes(bytes)

// Simulate a logging operation by writing text data and byte data to the
// end of the append blob.
for i in 0 .. numBlocks - 1 do
    let msg = sprintf "Timestamp: %u \tLog Entry: %d\n" DateTime.UtcNow bytes.[i]
    appendBlob.AppendText(msg)

// Read the append blob to the console window.
let downloadedText = appendBlob.DownloadText()
printfn "%s" downloadedText
```

See Understanding Block Blobs, Page Blobs, and Append Blobs for more information about the differences between the three types of blobs.

Concurrent access

To support concurrent access to a blob from multiple clients or multiple process instances, you can use **ETags** or **leases**.

- **Etag** - provides a way to detect that the blob or container has been modified by another process
- **Lease** - provides a way to obtain exclusive, renewable, write or delete access to a blob for a period of time

For more information, see Managing Concurrency in Microsoft Azure Storage.

Naming containers

Every blob in Azure storage must reside in a container. The container forms part of the blob name. For example, `mydata` is the name of the container in these sample blob URIs:

- https://storagesample.blob.core.windows.net/mydata/blob1.txt
- https://storagesample.blob.core.windows.net/mydata/photos/myphoto.jpg

A container name must be a valid DNS name, conforming to the following naming rules:

1. Container names must start with a letter or number, and can contain only letters, numbers, and the dash (-) character.
2. Every dash (-) character must be immediately preceded and followed by a letter or number; consecutive dashes are not permitted in container names.
3. All letters in a container name must be lowercase.
4. Container names must be from 3 through 63 characters long.

Note that the name of a container must always be lowercase. If you include an upper-case letter in a container name, or otherwise violate the container naming rules, you may receive a 400 error (Bad Request).

Managing security for blobs

By default, Azure Storage keeps your data secure by limiting access to the account owner, who is in possession of the account access keys. When you need to share blob data in your storage account, it is important to do so without compromising the security of your account access keys. Additionally, you can encrypt blob data to ensure that it is secure going over the wire and in Azure Storage.

Controlling access to blob data

By default, the blob data in your storage account is accessible only to storage account owner. Authenticating requests against Blob storage requires the account access key by default. However, you might want to make certain blob data available to other users.

Encrypting blob data

Azure Storage supports encrypting blob data both at the client and on the server.

Next steps

Now that you've learned the basics of Blob storage, follow these links to learn more.

Tools

- F# AzureStorageTypeProvider
  An F# Type Provider which can be used to explore Blob, Table and Queue Azure Storage assets and easily apply CRUD operations on them.
- FSharp.Azure.Storage

An F# API for using Microsoft Azure Table Storage service

- Microsoft Azure Storage Explorer (MASE)
  A free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, OS X, and Linux.

Blob storage reference

- Azure Storage APIs for .NET
- Azure Storage Services REST API Reference

Related guides

- Getting Started with Azure Blob Storage in C#
- Transfer data with the AzCopy command-line utility on Windows
- Transfer data with the AzCopy command-line utility on Linux
- Configure Azure Storage connection strings
- Azure Storage Team Blog
- Quickstart: Use .NET to create a blob in object storage

Get started with Azure File storage using F#

7/30/2019 • 6 minutes to read • Edit Online

Azure File storage is a service that offers file shares in the cloud using the standard Server Message Block (SMB) Protocol. Both SMB 2.1 and SMB 3.0 are supported. With Azure File storage, you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. Applications running in Azure virtual machines or cloud services or from on-premises clients can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

For a conceptual overview of file storage, please see the .NET guide for file storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You'll also need your storage access key for this account.

Create an F# Script and Start F# Interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `files.fsx`, in your F# development environment.

Next, use a package manager such as Paket or NuGet to install the `WindowsAzure.Storage` package and reference `WindowsAzure.Storage.dll` in your script using a `#r` directive.

Add namespace declarations

Add the following `open` statements to the top of the `files.fsx` file:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.File // Namespace for File storage types
```

Get your connection string

You'll need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you'll enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

However, this is **not recommended** for real projects. Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. You can regenerate your key using the Azure Portal if you believe it may have been compromised.

For real applications, the best way to maintain your storage connection string is in a configuration file. To fetch the connection string from a configuration file, you can do this:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Using Azure Configuration Manager is optional. You can also use an API such as the .NET Framework's `ConfigurationManager` type.

Parse the connection string

To parse the connection string, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

This will return a `CloudStorageAccount`.

Create the File service client

The `CloudFileClient` type enables you to programmatically use files stored in File storage. Here's one way to create the service client:

```
let fileClient = storageAccount.CreateCloudFileClient()
```

Now you are ready to write code that reads data from and writes data to File storage.

Create a file share

This example shows how to create a file share if it does not already exist:

```
let share = fileClient.GetShareReference("myfiles")
share.CreateIfNotExists()
```

Create a root directory and a subdirectory

Here, you get the root directory and get a sub-directory of the root. You create both if they don't already exist.

```
let rootDir = share.GetRootDirectoryReference()
let subDir = rootDir.GetDirectoryReference("myLogs")
subDir.CreateIfNotExists()
```

Upload text as a file

This example shows how to upload text as a file.

```
let file = subDir.GetFileReference("log.txt")
file.UploadText("This is the content of the log file")
```

Download a file to a local copy of the file

Here you download the file just created, appending the contents to a local file.

```
file.DownloadToFile("log.txt", FileMode.Append)
```

Set the maximum size for a file share

The example below shows how to check the current usage for a share and how to set the quota for the share. `FetchAttributes` must be called to populate a share's `Properties`, and `SetProperties` to propagate local changes to Azure File storage.

```
// stats.Usage is current usage in GB
let stats = share.GetStats()
share.FetchAttributes()

// Set the quota to 10 GB plus current usage
share.Properties.Quota <- stats.Usage + 10 |> Nullable
share.SetProperties()

// Remove the quota
share.Properties.Quota <- Nullable()
share.SetProperties()
```

Generate a shared access signature for a file or file share

You can generate a shared access signature (SAS) for a file share or for an individual file. You can also create a shared access policy on a file share to manage shared access signatures. Creating a shared access policy is recommended, as it provides a means of revoking the SAS if it should be compromised.

Here, you create a shared access policy on a share, and then use that policy to provide the constraints for a SAS on a file in the share.

```
// Create a 24-hour read/write policy.
let policy =
    SharedAccessFilePolicy
       (SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable),
        Permissions = (SharedAccessFilePermissions.Read ||| SharedAccessFilePermissions.Write))


// Set the policy on the share.
let permissions = share.GetPermissions()
permissions.SharedAccessPolicies.Add("policyName", policy)
share.SetPermissions(permissions)

let sasToken = file.GetSharedAccessSignature(policy)
let sasUri = Uri(file.StorageUri.PrimaryUri.ToString() + sasToken)

let fileSas = CloudFile(sasUri)
fileSas.UploadText("This write operation is authenticated via SAS")
```

For more information about creating and using shared access signatures, see Using Shared Access Signatures (SAS) and Create and use a SAS with Blob storage.

Copy files

You can copy a file to another file or to a blob, or a blob to a file. If you are copying a blob to a file, or a file to a blob, you *must* use a shared access signature (SAS) to authenticate the source object, even if you are copying within the same storage account.

Copy a file to another file

Here, you copy a file to another file in the same share. Because this copy operation copies between files in the same storage account, you can use Shared Key authentication to perform the copy.

```
let destFile = subDir.GetFileReference("log_copy.txt")
destFile.StartCopy(file)
```

Copy a file to a blob

Here, you create a file and copy it to a blob within the same storage account. You create a SAS for the source file, which the service uses to authenticate access to the source file during the copy operation.

```
// Get a reference to the blob to which the file will be copied.
let blobClient = storageAccount.CreateCloudBlobClient()
let container = blobClient.GetContainerReference("myContainer")
container.CreateIfNotExists()
let destBlob = container.GetBlockBlobReference("log_blob.txt")

let filePolicy =
    SharedAccessFilePolicy
        (Permissions = SharedAccessFilePermissions.Read,
         SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable))

let fileSas2 = file.GetSharedAccessSignature(filePolicy)
let sasUri2 = Uri(file.StorageUri.PrimaryUri.ToString() + fileSas2)
destBlob.StartCopy(sasUri2)
```

You can copy a blob to a file in the same way. If the source object is a blob, then create a SAS to authenticate access to that blob during the copy operation.

Troubleshooting File storage using metrics

Azure Storage Analytics supports metrics for File storage. With metrics data, you can trace requests and diagnose issues.

You can enable metrics for File storage from the Azure Portal, or you can do it from F# like this:

```
open Microsoft.Azure.Storage.File.Protocol
open Microsoft.Azure.Storage.Shared.Protocol

let props =
    FileServiceProperties(
       (HourMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (14 |> Nullable),
            Version = "1.0"),
        MinuteMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (7 |> Nullable),
            Version = "1.0"))

fileClient.SetServiceProperties(props)
```

Next steps

See these links for more information about Azure File storage.

Conceptual articles and videos

- Azure Files Storage: a frictionless cloud SMB file system for Windows and Linux
- How to use Azure File Storage with Linux

Tooling support for File storage

- Using Azure PowerShell with Azure Storage
- How to use AzCopy with Microsoft Azure Storage
- Using the Azure CLI with Azure Storage

Reference

- Storage Client Library for .NET reference
- File Service REST API reference

Blog posts

- Azure File storage is now generally available
- Inside Azure File Storage
- Introducing Microsoft Azure File Service
- Persisting connections to Microsoft Azure Files

Get started with Azure Queue storage using F#

7/30/2019 • 6 minutes to read • Edit Online

Azure Queue storage provides cloud messaging between application components. In designing applications for scale, application components are often decoupled, so that they can scale independently. Queue storage delivers asynchronous messaging for communication between application components, whether they are running in the cloud, on the desktop, on an on-premises server, or on a mobile device. Queue storage also supports managing asynchronous tasks and building process work flows.

About this tutorial

This tutorial shows how to write F# code for some common tasks using Azure Queue storage. Tasks covered include creating and deleting queues and adding, reading, and deleting queue messages.

For a conceptual overview of queue storage, please see the .NET guide for queue storage.

Prerequisites

To use this guide, you must first create an Azure storage account. You'll also need your storage access key for this account.

Create an F# Script and Start F# Interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `queues.fsx`, in your F# development environment.

Next, use a package manager such as Paket or NuGet to install the `WindowsAzure.Storage` package and reference `WindowsAzure.Storage.dll` in your script using a `#r` directive.

Add namespace declarations

Add the following `open` statements to the top of the `queues.fsx` file:

```
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Queue // Namespace for Queue storage types
```

Get your connection string

You'll need an Azure Storage connection string for this tutorial. For more information about connection strings, see Configure Storage Connection Strings.

For the tutorial, you'll enter your connection string in your script, like this:

```
let storageConnString = "..." // fill this in from your storage account
```

However, this is **not recommended** for real projects. Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. You can regenerate your key using the Azure Portal if you believe it may have been compromised.

For real applications, the best way to maintain your storage connection string is in a configuration file. To fetch the connection string from a configuration file, you can do this:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Using Azure Configuration Manager is optional. You can also use an API such as the .NET Framework's `ConfigurationManager` type.

Parse the connection string

To parse the connection string, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

This will return a `CloudStorageAccount`.

Create the Queue service client

The `CloudQueueClient` class enables you to retrieve queues stored in Queue storage. Here's one way to create the service client:

```
let queueClient = storageAccount.CreateCloudQueueClient()
```

Now you are ready to write code that reads data from and writes data to Queue storage.

Create a queue

This example shows how to create a queue if it doesn't already exist:

```
// Retrieve a reference to a container.
let queue = queueClient.GetQueueReference("myqueue")

// Create the queue if it doesn't already exist
queue.CreateIfNotExists()
```

Insert a message into a queue

To insert a message into an existing queue, first create a new `CloudQueueMessage`. Next, call the `AddMessage` method. A `CloudQueueMessage` can be created from either a string (in UTF-8 format) or a `byte` array, like this:

```
// Create a message and add it to the queue.
let message = new CloudQueueMessage("Hello, World")
queue.AddMessage(message)
```

Peek at the next message

You can peek at the message in the front of a queue, without removing it from the queue, by calling the `PeekMessage` method.

```
// Peek at the next message.
let peekedMessage = queue.PeekMessage()
let msgAsString = peekedMessage.AsString
```

Get the next message for processing

You can retrieve the message at the front of a queue for processing by calling the `GetMessage` method.

```
// Get the next message. Successful processing must be indicated via DeleteMessage later.
let retrieved = queue.GetMessage()
```

You later indicate successful processing of the message by using `DeleteMessage`.

Change the contents of a queued message

You can change the contents of a retrieved message in-place in the queue. If the message represents a work task, you could use this feature to update the status of the work task. The following code updates the queue message with new contents, and sets the visibility timeout to extend another 60 seconds. This saves the state of work associated with the message, and gives the client another minute to continue working on the message. You could use this technique to track multi-step workflows on queue messages, without having to start over from the beginning if a processing step fails due to hardware or software failure. Typically, you would keep a retry count as well, and if the message is retried more than some number of times, you would delete it. This protects against a message that triggers an application error each time it is processed.

```
// Update the message contents and set a new timeout.
retrieved.SetMessageContent("Updated contents.")
queue.UpdateMessage(retrieved,
    TimeSpan.FromSeconds(60.0),
    MessageUpdateFields.Content ||| MessageUpdateFields.Visibility)
```

De-queue the next message

Your code de-queues a message from a queue in two steps. When you call `GetMessage`, you get the next message in a queue. A message returned from `GetMessage` becomes invisible to any other code reading messages from this queue. By default, this message stays invisible for 30 seconds. To finish removing the message from the queue, you must also call `DeleteMessage`. This two-step process of removing a message assures that if your code fails to process a message due to hardware or software failure, another instance of your code can get the same message and try again. Your code calls `DeleteMessage` right after the message has been processed.

```
// Process the message in less than 30 seconds, and then delete the message.
queue.DeleteMessage(retrieved)
```

Use Async workflows with common Queue storage APIs

This example shows how to use an async workflow with common Queue storage APIs.

```
async {
    let! exists = queue.CreateIfNotExistsAsync() |> Async.AwaitTask

    let! retrieved = queue.GetMessageAsync() |> Async.AwaitTask

    // ... process the message here ...

    // Now indicate successful processing:
    do! queue.DeleteMessageAsync(retrieved) |> Async.AwaitTask
}
```

Additional options for de-queuing messages

There are two ways you can customize message retrieval from a queue. First, you can get a batch of messages (up to 32). Second, you can set a longer or shorter invisibility timeout, allowing your code more or less time to fully process each message. The following code example uses `GetMessages` to get 20 messages in one call and then processes each message. It also sets the invisibility timeout to five minutes for each message. Note that the 5 minutes starts for all messages at the same time, so after 5 minutes have passed since the call to `GetMessages`, any messages which have not been deleted will become visible again.

```
for msg in queue.GetMessages(20, Nullable(TimeSpan.FromMinutes(5.))) do
        // Process the message here.
        queue.DeleteMessage(msg)
```

Get the queue length

You can get an estimate of the number of messages in a queue. The `FetchAttributes` method asks the Queue service to retrieve the queue attributes, including the message count. The `ApproximateMessageCount` property returns the last value retrieved by the `FetchAttributes` method, without calling the Queue service.

```
queue.FetchAttributes()
let count = queue.ApproximateMessageCount.GetValueOrDefault()
```

Delete a queue

To delete a queue and all the messages contained in it, call the `Delete` method on the queue object.

```
// Delete the queue.
queue.Delete()
```

Next steps

Now that you've learned the basics of Queue storage, follow these links to learn about more complex storage tasks.

- Azure Storage APIs for .NET
- Azure Storage Type Provider
- Azure Storage Team Blog
- Configure Azure Storage connection strings
- Azure Storage Services REST API Reference

Get started with Azure Table storage and the Azure Cosmos DB Table API using F#

7/30/2019 • 9 minutes to read • Edit Online

Azure Table storage is a service that stores structured NoSQL data in the cloud. Table storage is a key/attribute store with a schemaless design. Because Table storage is schemaless, it's easy to adapt your data as the needs of your application evolve. Access to data is fast and cost-effective for all kinds of applications. Table storage is typically significantly lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets, such as user data for web applications, address books, device information, and any other type of metadata that your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

Azure Cosmos DB provides the Table API for applications that are written for Azure Table storage and that require premium capabilities such as:

- Turnkey global distribution.
- Dedicated throughput worldwide.
- Single-digit millisecond latencies at the 99th percentile.
- Guaranteed high availability.
- Automatic secondary indexing.

Applications written for Azure Table storage can migrate to Azure Cosmos DB by using the Table API with no code changes and take advantage of premium capabilities. The Table API has client SDKs available for .NET, Java, Python, and Node.js.

For more information, see Introduction to Azure Cosmos DB Table API.

About this tutorial

This tutorial shows how to write F# code to do some common tasks using Azure Table storage or the Azure Cosmos DB Table API, including creating and deleting a table and inserting, updating, deleting, and querying table data.

Prerequisites

To use this guide, you must first create an Azure storage account or Azure Cosmos DB account.

Create an F# Script and Start F# Interactive

The samples in this article can be used in either an F# application or an F# script. To create an F# script, create a file with the `.fsx` extension, for example `tables.fsx`, in your F# development environment.

Next, use a package manager such as Paket or NuGet to install the `WindowsAzure.Storage` package and reference `WindowsAzure.Storage.dll` in your script using a `#r` directive. Do it again for `Microsoft.WindowsAzure.ConfigurationManager` in order to get the Microsoft.Azure namespace.

Add namespace declarations

Add the following `open` statements to the top of the `tables.fsx` file:

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Table // Namespace for Table storage types
```

Get your Azure Storage connection string

If you're connecting to Azure Storage Table service, you'll need your connection string for this tutorial. You can copy your connection string from the Azure portal. For more information about connection strings, see Configure Storage Connection Strings.

Get your Azure Cosmos DB connection string

If you're connecting to Azure Cosmos DB, you'll need your connection string for this tutorial. You can copy your connection string from the Azure portal. In the Azure portal, in your Cosmos DB account, go to **Settings** > **Connection String**, and click the **Copy** button to copy your Primary Connection String.

For the tutorial, enter your connection string in your script, like the following example:

```
let storageConnString = "..." // fill this in from your storage account
```

However, this is **not recommended** for real projects. Your storage account key is similar to the root password for your storage account. Always be careful to protect your storage account key. Avoid distributing it to other users, hard-coding it, or saving it in a plain-text file that is accessible to others. You can regenerate your key using the Azure Portal if you believe it may have been compromised.

For real applications, the best way to maintain your storage connection string is in a configuration file. To fetch the connection string from a configuration file, you can do this:

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

Using Azure Configuration Manager is optional. You can also use an API such as the .NET Framework's `ConfigurationManager` type.

Parse the connection string

To parse the connection string, use:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

This returns a `CloudStorageAccount`.

Create the Table service client

The `CloudTableClient` class enables you to retrieve tables and entities in Table storage. Here's one way to create the service client:

```
// Create the table client.
let tableClient = storageAccount.CreateCloudTableClient()
```

Now you are ready to write code that reads data from and writes data to Table storage.

Create a table

This example shows how to create a table if it does not already exist:

```
// Retrieve a reference to the table.
let table = tableClient.GetTableReference("people")

// Create the table if it doesn't exist.
table.CreateIfNotExists()
```

Add an entity to a table

An entity has to have a type that inherits from `TableEntity`. You can extend `TableEntity` in any way you like, but your type *must* have a parameter-less constructor. Only properties that have both `get` and `set` are stored in your Azure Table.

An entity's partition and row key uniquely identify the entity in the table. Entities with the same partition key can be queried faster than those with different partition keys, but using diverse partition keys allows for greater scalability of parallel operations.

Here's an example of a `Customer` that uses the `lastName` as the partition key and the `firstName` as the row key.

```
type Customer(firstName, lastName, email: string, phone: string) =
    inherit TableEntity(partitionKey=lastName, rowKey=firstName)
    new() = Customer(null, null, null, null)
    member val Email = email with get, set
    member val PhoneNumber = phone with get, set

let customer =
    Customer("Walter", "Harp", "Walter@contoso.com", "425-555-0101")
```

Now add `Customer` to the table. To do so, create a `TableOperation` that executes on the table. In this case, you create an `Insert` operation.

```
let insertOp = TableOperation.Insert(customer)
table.Execute(insertOp)
```

Insert a batch of entities

You can insert a batch of entities into a table using a single write operation. Batch operations allow you to combine operations into a single execution, but they have some restrictions:

- You can perform updates, deletes, and inserts in the same batch operation.
- A batch operation can include up to 100 entities.
- All entities in a batch operation must have the same partition key.
- While it is possible to perform a query in a batch operation, it must be the only operation in the batch.

Here's some code that combines two inserts into a batch operation:

```
let customer1 =
    Customer("Jeff", "Smith", "Jeff@contoso.com", "425-555-0102")

let customer2 =
    Customer("Ben", "Smith", "Ben@contoso.com", "425-555-0103")

let batchOp = TableBatchOperation()
batchOp.Insert(customer1)
batchOp.Insert(customer2)
table.ExecuteBatch(batchOp)
```

Retrieve all entities in a partition

To query a table for all entities in a partition, use a `TableQuery` object. Here, you filter for entities where "Smith" is the partition key.

```
let query =
    TableQuery<Customer>().Where(
        TableQuery.GenerateFilterCondition(
            "PartitionKey", QueryComparisons.Equal, "Smith"))

let result = table.ExecuteQuery(query)
```

You now print the results:

```
for customer in result do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

Retrieve a range of entities in a partition

If you don't want to query all the entities in a partition, you can specify a range by combining the partition key filter with a row key filter. Here, you use two filters to get all entities in the "Smith" partition where the row key (first name) starts with a letter earlier than "M" in the alphabet.

```
let range =
    TableQuery<Customer>().Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, "Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(
                "RowKey", QueryComparisons.LessThan, "M")))

let rangeResult = table.ExecuteQuery(range)
```

You now print the results:

```
for customer in rangeResult do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

Retrieve a single entity

You can write a query to retrieve a single, specific entity. Here, you use a `TableOperation` to specify the customer "Ben Smith". Instead of a collection, you get back a `Customer`. Specifying both the partition key and the row key in a query is the fastest way to retrieve a single entity from the Table service.

```
let retrieveOp = TableOperation.Retrieve<Customer>("Smith", "Ben")

let retrieveResult = table.Execute(retrieveOp)
```

You now print the results:

```
// Show the result
let retrieveCustomer = retrieveResult.Result :?> Customer
printfn "customer: %A %A" retrieveCustomer.RowKey retrieveCustomer.PartitionKey
```

Replace an entity

To update an entity, retrieve it from the Table service, modify the entity object, and then save the changes back to the Table service using a `Replace` operation. This causes the entity to be fully replaced on the server, unless the entity on the server has changed since it was retrieved, in which case the operation fails. This failure is to prevent your application from inadvertently overwriting changes from other sources.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0103"
    let replaceOp = TableOperation.Replace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeeded")
with e ->
    Console.WriteLine("Update failed")
```

Insert-or-replace an entity

Sometimes, you don't know whether an entity exists in the table. And if it does, the current values stored in it are no longer needed. You can use `InsertOrReplace` to create the entity, or replace it if it exists, regardless of its state.

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0104"
    let replaceOp = TableOperation.InsertOrReplace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeeded")
with e ->
    Console.WriteLine("Update failed")
```

Query a subset of entity properties

A table query can retrieve just a few properties from an entity instead of all of them. This technique, called projection, can improve query performance, especially for large entities. Here, you return only email addresses using `DynamicTableEntity` and `EntityResolver`. Note that projection is not supported on the local storage emulator, so this code runs only when you're using an account on the Table service.

```
// Define the query, and select only the Email property.
let projectionQ = TableQuery<DynamicTableEntity>().Select [|"Email"|]

// Define an entity resolver to work with the entity after retrieval.
let resolver = EntityResolver<string>(fun pk rk ts props etag ->
    if props.ContainsKey("Email") then
        props.["Email"].StringValue
    else
        null
    )

let resolvedResults = table.ExecuteQuery(projectionQ, resolver, null, null)
```

Retrieve entities in pages asynchronously

If you are reading a large number of entities, and you want to process them as they are retrieved rather than waiting for them all to return, you can use a segmented query. Here, you return results in pages by using an async workflow so that execution is not blocked while you're waiting for a large set of results to return.

```
let tableQ = TableQuery<Customer>()

let asyncQuery =
    let rec loop (cont: TableContinuationToken) = async {
        let! ct = Async.CancellationToken
        let! result = table.ExecuteQuerySegmentedAsync(tableQ, cont, ct) |> Async.AwaitTask

        // ...process the result here...

        // Continue to the next segment
        match result.ContinuationToken with
        | null -> ()
        | cont -> return! loop cont
    }
    loop null
```

You now execute this computation synchronously:

```
let asyncResults = asyncQuery |> Async.RunSynchronously
```

Delete an entity

You can delete an entity after you have retrieved it. As with updating an entity, this fails if the entity has changed since you retrieved it.

```
let deleteOp = TableOperation.Delete(customer)
table.Execute(deleteOp)
```

Delete a table

You can delete a table from a storage account. A table that has been deleted will be unavailable to be re-created for a period of time following the deletion.

```
table.DeleteIfExists()
```

Next steps

Now that you've learned the basics of Table storage, follow these links to learn about more complex storage tasks and the Azure Cosmos DB Table API.

- Introduction to Azure Cosmos DB Table API
- Storage Client Library for .NET reference
- Azure Storage Type Provider
- Azure Storage Team Blog
- Configuring Connection Strings
- Getting Started with Azure Table Storage in .NET

Package Management for F# Azure Dependencies

5/18/2019 • 2 minutes to read • Edit Online

Obtaining packages for Azure development is easy when you use a package manager. The two options are Paket and NuGet.

Using Paket

If you're using Paket as your dependency manager, you can use the `paket.exe` tool to add Azure dependencies. For example:

```
> paket add nuget WindowsAzure.Storage
```

Or, if you're using Mono for cross-platform .NET development:

```
> mono paket.exe add nuget WindowsAzure.Storage
```

This will add `WindowsAzure.Storage` to your set of package dependencies for the project in the current directory, modify the `paket.dependencies` file, and download the package. If you have previously set up dependencies, or are working with a project where dependencies have been set up by another developer, you can resolve and install dependencies locally like this:

```
> paket install
```

Or, for Mono development:

```
> mono paket.exe install
```

You can update all your package dependencies to the latest version like this:

```
> paket update
```

Or, for Mono development:

```
> mono paket.exe update
```

Using Nuget

If you're using NuGet as your dependency manager, you can use the `nuget.exe` tool to add Azure dependencies. For example:

```
> nuget install WindowsAzure.Storage -ExcludeVersion
```

Or, for Mono development:

```
> mono nuget.exe install WindowsAzure.Storage -ExcludeVersion
```

This will add `WindowsAzure.Storage` to your set of package dependencies for the project in the current directory, and download the package. If you have previously set up dependencies, or are working with a project where dependencies have been set up by another developer, you can resolve and install dependencies locally like this:

```
> nuget restore
```

Or, for Mono development:

```
> mono nuget.exe restore
```

You can update all your package dependencies to the latest version like this:

```
> nuget update
```

Or, for Mono development:

```
> mono nuget.exe update
```

Referencing Assemblies

In order to use your packages in your F# script, you need to reference the assemblies included in the packages using a `#r` directive. For example:

```
> #r "packages/WindowsAzure.Storage/lib/net40/Microsoft.WindowsAzure.Storage.dll"
```

As you can see, you'll need to specify the relative path to the DLL and the full DLL name, which may not be exactly the same as the package name. The path will include a framework version and possibly a package version number. To find all the installed assemblies, you can use something like this on a Windows command line:

```
> cd packages/WindowsAzure.Storage
> dir /s/b *.dll
```

Or in a Unix shell, something like this:

```
> find packages/WindowsAzure.Storage -name "*.dll"
```

This will give you the paths to the installed assemblies. From there, you can select the correct path for your framework version.

Visual Basic Guide

5/14/2019 • 2 minutes to read • Edit Online

Visual Basic is engineered for productively building type-safe and object-oriented applications. Visual Basic enables developers to target Windows, Web, and mobile devices. As with all languages targeting the Microsoft .NET Framework, programs written in Visual Basic benefit from security and language interoperability.

This generation of Visual Basic continues the tradition of giving you a fast and easy way to create .NET Framework-based applications.

If you don't already have Visual Basic, you can download a free version of Visual Studio that includes Visual Basic from the Visual Studio site.

In This Section

Getting Started
Helps you begin working by listing what is new and what is available in various editions of the product.

Programming Concepts
Presents the language concepts that are most useful to Visual Basic programmers.

Program Structure and Code Conventions
Contains documentation on the basic structure and code conventions of Visual Basic such as naming conventions, comments in code, and limitations within Visual Basic.

Visual Basic Language Features
Provides links to topics that introduce and discuss important features of Visual Basic, including LINQ and XML literals.

Visual Basic Reference
Contains the Visual Basic language and compiler information.

Developing Applications with Visual Basic
Discusses various aspects of development in Visual Basic, such as security, exception handling, and using the .NET Framework class library.

COM Interop
Explains the interoperability issues associated with creating and using component object model (COM) objects with Visual Basic.

Samples
Contains information about samples.

Walkthroughs
Provides links to step-by-step instructions for common scenarios.

Related Sections

Get Started Developing with Visual Studio
Provides links to topics that help you learn about the basics of Visual Studio.

.NET API Browser
Provides entry to the library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK.

What's new for Visual Basic

8/20/2019 • 11 minutes to read • Edit Online

This topic lists key feature names for each version of Visual Basic, with detailed descriptions of the new and enhanced features in the latest versions of the language.

Current version

Visual Basic 15.8 / Visual Studio 2017 Version 15.8 For new features, see Visual Basic 15.8

Previous versions

Visual Basic 15.5 / Visual Studio 2017 Version 15.5 For new features, see Visual Basic 15.5

Visual Basic 15.3 / Visual Studio 2017 Version 15.3 For new features, see Visual Basic 15.3

Visual Basic 2017 / Visual Studio 2017 For new features, see Visual Basic 2017

Visual Basic / Visual Studio 2015 For new features, see Visual Basic 14

Visual Basic / Visual Studio 2013 Technology previews of the .NET Compiler Platform ("Roslyn")

Visual Basic / Visual Studio 2012 `Async` and `await` keywords, iterators, caller info attributes

Visual Basic, Visual Studio 2010 Auto-implemented properties, collection initializers, implicit line continuation, dynamic, generic co/contra variance, global namespace access

Visual Basic / Visual Studio 2008 Language Integrated Query (LINQ), XML literals, local type inference, object initializers, anonymous types, extension methods, local `var` type inference, lambda expressions, `if` operator, partial methods, nullable value types

Visual Basic / Visual Studio 2005 The `My` type and helper types (access to app, computer, files system, network)

Visual Basic / Visual Studio .NET 2003 Bit-shift operators, loop variable declaration

Visual Basic / Visual Studio .NET 2002 The first release of Visual Basic .NET

Visual Basic 15.8

Optimized floating-point to integer conversion

In previous versions of Visual Basic, conversion of Double and Single values to integers offered relatively poor performance. Visual Basic 15.8 significantly enhances the performance of floating-point conversions to integers when you pass the value returned by any of the following methods to one of the intrinsic Visual Basic integer conversion functions (CByte, CShort, CInt, CLng, CSByte, CUShort, CUInt, CULng), or when the value returned by any of the following methods is implicitly cast to an integral type when Option Strict is set to `Off`:

- Conversion.Fix(Double)
- Conversion.Fix(Object)
- Conversion.Fix(Single)
- Conversion.Int(Double)
- Conversion.Int(Object)
- Conversion.Int(Single)

- Math.Ceiling(Double)
- Math.Floor(Double)
- Math.Round(Double)
- Math.Truncate(Double)

This optimization allows code to run faster -- up to twice as fast for code that does a large number of conversions to integer types. The following example illustrates some simple method calls that are affected by this optimization:

```
Dim s As Single = 173.7619
Dim d As Double = s

Dim i1 As Integer = CInt(Fix(s))              ' Result: 173
Dim b1 As Byte = CByte(Int(d))                ' Result: 173
Dim s1 AS Short = CShort(Math.Truncate(s))    ' Result: 173
Dim i2 As Integer = CInt(Math.Ceiling(d))     ' Result: 174
Dim i3 As Integer = CInt(Math.Round(s))       ' Result: 174
```

Note that this truncates rather than rounds floating-point values.

Visual Basic 15.5

Non-trailing named arguments

In Visual Basic 15.3 and earlier versions, when a method call included arguments both by position and by name, positional arguments had to precede named arguments. Starting with Visual Basic 15.5, positional and named arguments can appear in any order as long as all arguments up to the last positional argument are in the correct position. This is particularly useful when named arguments are used to make code more readable.

For example, the following method call has two positional arguments between a named argument. The named argument makes it clear that the value 19 represents an age.

```
StudentInfo.Display("Mary", age:=19, #9/21/1998#)
```

`Private Protected` member access modifier

This new keyword combination defines a member that is accessible by all members in its containing class as well as by types derived from the containing class, but only if they are also found in the containing assembly. Because structures cannot be inherited, `Private Protected` can only be applied to the members of a class.

Leading hex/binary/octal separator

Visual Basic 2017 added support for the underscore character (`_`) as a digit separator. Starting with Visual Basic 15.5, you can use the underscore character as a leading separator between the prefix and hexadecimal, binary, or octal digits. The following example uses a leading digit separator to define 3,271,948,384 as a hexadecimal number:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

Visual Basic 15.3

Named tuple inference

When you assign the value of tuple elements from variables, Visual Basic infers the name of tuple elements from the corresponding variable names; you do not have to explicitly name a tuple element. The following example uses inference to create a tuple with three named elements, `state`, `stateName`, and `capital`.

```
Dim state = "MI"
Dim stateName = "Michigan"
Dim capital = "Lansing"
Dim stateInfo = ( state, stateName, capital )
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.State}, Capital {stateInfo.capital}")
' The example displays the following output:
'      Michigan: 2-letter code: MI, Capital Lansing
```

Additional compiler switches

The Visual Basic command-line compiler now supports the **-refout** and **-refonly** compiler options to control the output of reference assemblies. **-refout** defines the output directory of the reference assembly, and **-refonly** specifies that only a reference assembly is to be output by compilation.

Visual Basic 2017

Tuples

Tuples are a lightweight data structure that most commonly is used to return multiple values from a single method call. Ordinarily, to return multiple values from a method, you have to do one of the following:

- Define a custom type (a `Class` or a `Structure`). This is a heavyweight solution.
- Define one or more `ByRef` parameters, in addition to returning a value from the method.

Visual Basic's support for tuples lets you quickly define a tuple, optionally assign semantic names to its values, and quickly retrieve its values. The following example wraps a call to the TryParse method and returns a tuple.

```
Imports System.Globalization

Public Module NumericLibrary
    Public Function ParseInteger(value As String) As (Success As Boolean, Number As Int32)
        Dim number As Integer
        Return (Int32.TryParse(value, NumberStyles.Any, CultureInfo.InvariantCulture, number), number)
    End Function
End Module
```

You can then call the method and handle the returned tuple with code like the following.

```
Dim numericString As String = "123,456"
Dim result = ParseInteger(numericString)
Console.WriteLine($"{If(result.Success, $"Success: {result.Number:N0}", "Failure")}")
Console.ReadLine()
'      Output: Success: 123,456
```

Binary literals and digit separators

You can define a binary literal by using the prefix `&B` or `&b`. In addition, you can use the underscore character, `_`, as a digit separator to enhance readability. The following example uses both features to assign a `Byte` value and to display it as a decimal, hexadecimal, and binary number.

```
Dim value As Byte = &B0110_1110
Console.WriteLine($"{NameOf(value)}  = {value} (hex: 0x{value:X2}) " +
                  $"(binary: {Convert.ToString(value, 2)})")
' The example displays the following output:
'      value  = 110 (hex: 0x6E) (binary: 1101110)
```

For more information, see the "Literal assignments" section of the Byte, Integer, Long, Short, SByte, UInteger, ULong, and UShort data types.

Support for C# reference return values

Starting with C# 7.0, C# supports reference return values. That is, when the calling method receives a value returned by reference, it can change the value of the reference. Visual Basic does not allow you to author methods with reference return values, but it does allow you to consume and modify the reference return values.

For example, the following `Sentence` class written in C# includes a `FindNext` method that finds the next word in a sentence that begins with a specified substring. The string is returned as a reference return value, and a `Boolean` variable passed by reference to the method indicates whether the search was successful. This means that the caller can not only read the returned value; he or she can also modify it, and that modification is reflected in the `Sentence` class.

```
using System;

public class Sentence
{
    private string[] words;
    private int currentSearchPointer;

    public Sentence(string sentence)
    {
        words = sentence.Split(' ');
        currentSearchPointer = -1;
    }

    public ref string FindNext(string startWithString, ref bool found)
    {
        for (int count = currentSearchPointer + 1; count < words.Length; count++)
        {
            if (words[count].StartsWith(startWithString))
            {
                currentSearchPointer = count;
                found = true;
                return ref words[currentSearchPointer];
            }
        }
        currentSearchPointer = -1;
        found = false;
        return ref words[0];
    }

    public string GetSentence()
    {
        string stringToReturn = null;
        foreach (var word in words)
            stringToReturn += $"{word} ";

        return stringToReturn.Trim();
    }
}
```

In its simplest form, you can modify the word found in the sentence by using code like the following. Note that you are not assigning a value to the method, but rather to the expression that the method returns, which is the reference return value.

```
Dim sentence As New Sentence("A time to see the world is now.")
Dim found = False
sentence.FindNext("A", found) = "A good"
Console.WriteLine(sentence.GetSentence())
' The example displays the following output:
'      A good time to see the world is now.
```

A problem with this code, though, is that if a match is not found, the method returns the first word. Since the example does not examine the value of the `Boolean` argument to determine whether a match is found, it modifies the first word if there is no match. The following example corrects this by replacing the first word with itself if there is no match.

```
Dim sentence As New Sentence("A time to see the world is now.")
Dim found = False
sentence.FindNext("A", found) = IIf(found, "A good", sentence.FindNext("B", found))
Console.WriteLine(sentence.GetSentence())
' The example displays the following output:
'      A good time to see the world is now.
```

A better solution is to use a helper method to which the reference return value is passed by reference. The helper method can then modify the argument passed to it by reference. The following example does that.

```
Module Example
   Public Sub Main()
      Dim sentence As New Sentence("A time to see the world is now.")
      Dim found = False
      Dim returns = RefHelper(sentence.FindNext("A", found), "A good", found)
      Console.WriteLine(sentence.GetSentence())
   End Sub

   Private Function RefHelper(ByRef stringFound As String, replacement As String, success As Boolean) _
                   As (originalString As String, found As Boolean)
      Dim originalString = stringFound
      If found Then stringFound = replacement
      Return (originalString, found)
   End Function
End Module
' The example displays the following output:
'      A good time to see the world is now.
```

For more information, see Reference Return Values.

Visual Basic 14

Nameof

You can get the unqualified string name of a type or member for use in an error message without hard coding a string. This allows your code to remain correct when refactoring. This feature is also useful for hooking up model-view-controller MVC links and firing property changed events.

String interpolation

You can use string interpolation expressions to construct strings. An interpolated string expression looks like a template string that contains expressions. An interpolated string is easier to understand with respect to arguments than Composite Formatting.

Null-conditional member access and indexing

You can test for null in a very light syntactic way before performing a member access (`?.`) or index (`?[]`) operation. These operators help you write less code to handle null checks, especially for descending into data structures. If the left operand or object reference is null, the operations returns null.

Multi-line string literals

String literals can contain newline sequences. You no longer need the old work around of using `<xml><![CDATA[...text with newlines...]]></xml>.Value`

Comments

You can put comments after implicit line continuations, inside initializer expressions, and among LINQ expression terms.

Smarter fully-qualified name resolution

Given code such as `Threading.Thread.Sleep(1000)`, Visual Basic used to look up the namespace "Threading", discover it was ambiguous between System.Threading and System.Windows.Threading, and then report an error. Visual Basic now considers both possible namespaces together. If you show the completion list, the Visual Studio editor lists members from both types in the completion list.

Year-first date literals

You can have date literals in yyyy-mm-dd format, `#2015-03-17 16:10 PM#`.

Readonly interface properties

You can implement readonly interface properties using a readwrite property. The interface guarantees minimum functionality, and it does not stop an implementing class from allowing the property to be set.

TypeOf <expr> IsNot <type>

For more readability of your code, you can now use `TypeOf` with `IsNot`.

#Disable Warning <ID> and #Enable Warning <ID>

You can disable and enable specific warnings for regions within a source file.

XML doc comment improvements

When writing doc comments, you get smart editor and build support for validating parameter names, proper handling of `crefs` (generics, operators, etc.), colorizing, and refactoring.

Partial module and interface definitions

In addition to classes and structs, you can declare partial modules and interfaces.

#Region directives inside method bodies

You can put #Region...#End Region delimiters anywhere in a file, inside functions, and even spanning across function bodies.

Overrides definitions are implicitly overloads

If you add the `Overrides` modifier to a definition, the compiler implicitly adds `Overloads` so that you can type less code in common cases.

CObj allowed in attributes arguments

The compiler used to give an error that CObj(...) was not a constant when used in attribute constructions.

Declaring and consuming ambiguous methods from different interfaces

Previously the following code yielded errors that prevented you from declaring `IMock` or from calling `GetDetails` (if these had been declared in C#):

```
Interface ICustomer
  Sub GetDetails(x As Integer)
End Interface

Interface ITime
  Sub GetDetails(x As String)
End Interface

Interface IMock : Inherits ICustomer, ITime
  Overloads Sub GetDetails(x As Char)
End Interface

Interface IMock2 : Inherits ICustomer, ITime
End Interface
```

Now the compiler will use normal overload resolution rules to choose the most appropriate `GetDetails` to call, and you can declare interface relationships in Visual Basic like those shown in the sample.

See also

- What's New in Visual Studio 2017

Get started with Visual Basic

5/23/2019 • 2 minutes to read • Edit Online

This section of the documentation helps you get started with Visual Basic application development.

Get started with Visual Basic and .NET Core

Build a Visual Basic Hello World application with .NET Core in Visual Studio 2017

Build a class library with Visual Basic and .NET Core in Visual Studio 2017

Additional information

- What's new for Visual Basic
  Lists new features in each of the versions of Visual Basic .NET.
- Visual Basic Breaking Changes in Visual Studio
  Lists changes in this release that might prevent an application from compiling or change its run-time behavior.
- Additional Resources for Visual Basic Programmers
  Provides a list of Web sites and newsgroups that can help you find answers to common problems.

See also

- Get Visual Basic
  Provides download links for Visual Studio versions that include Visual Basic support, including free versions.
- Visual Basic Fundamentals for Absolute Beginners
  Microsoft Virtual Academy course that teaches the fundamentals of Visual Basic programming.
- Object-Oriented Programming
  Provides links to pages that introduce object-oriented programming and describe how to create your own objects and how to use objects to simplify your coding.
- Samples
  Provides links to sample code in Visual Basic.
- Walkthroughs
  Provides a list of Help pages that demonstrate aspects of the Visual Basic language.
- Talk to Us
  Covers how to receive support and give feedback.
- Visual Studio
  Provides links into the Visual Studio documentation.
- C#
  Provides links into the documentation on application development with Visual C#.
- Visual C++
  Provides links into the Visual C++ documentation.

- Office and SharePoint Development
  Provides information about using Microsoft Office and Visual Studio as part of a business application.

Visual Basic Breaking Changes in Visual Studio

4/2/2019 • 2 minutes to read • Edit Online

For the latest documentation on Visual Studio 2017, see Visual Studio 2017 Documentation.

No changes in Visual Basic in Visual Studio 2015 will prevent an application that was created in Visual Basic in Visual Studio 2013 from compiling or change the run-time behavior of such an application.

See also

- Arrays
- LINQ
- Lambda Expressions
- For Each...Next Statement
- Getting Started
- When is a non-breaking language fix breaking?

Additional resources

8/7/2019 • 2 minutes to read • Edit Online

The following web sites provide guidance and can help you find answers to common problems.

Microsoft resources

On the web

URL	DESCRIPTION
Visual Basic .NET Language Design	Official repository on GitHub for Visual Basic .NET language design.
Microsoft Visual Basic Team Blog	Provides access to the Visual Basic team blog.

Code samples

URL	DESCRIPTION
Visual Basic documentation samples	Contains the samples used throughout the Visual Basic and .NET documentation.

Forums

URL	DESCRIPTION
Visual Basic Forums	Discusses general Visual Basic issues.

Videos and webcasts

URL	DESCRIPTION
Channel9	Provides continuous community through videos, Wikis, and forums.

Support

URL	DESCRIPTION
Microsoft Support	Provides access to Knowledge Base (KB) articles, downloads and updates, support webcasts, and other services.
Visual Studio Questions	Enables you to file bugs or provide suggestions to Microsoft about .NET and Visual Studio. You can also report a bug by selecting **Help** > **Send Feedback** > **Report a Problem** in Visual Studio.

Third-party resources

URL	DESCRIPTION
VBForums	Provides a forum to discuss Visual Basic, .NET, and more.
vbCity	A community site for people to learn and ask questions about Visual Basic and .NET.
Stack Overflow	Stack Overflow is a question and answer site for developers.

See also

- Get started with Visual Basic
- Talk to Us

Developing Applications with Visual Basic

3/5/2019 • 2 minutes to read • Edit Online

This section covers conceptual documentation for the Visual Basic language.

In This Section

Programming in Visual Basic

Covers a variety of programming subjects.

Development with My

Discusses a new feature called `My`, which provides access to information and default object instances that are related to an application and its run-time environment.

Accessing Data in Visual Basic Applications

Contains Help on accessing data in Visual Basic.

Creating and Using Components in Visual Basic

Defines the term *component* and discusses how and when to create components.

Windows Forms Application Basics

Provides information about creating Windows Forms applications by using Visual Studio.

Customizing Projects and Extending My

Describes how you can customize project templates to provide additional `My` objects.

Related Sections

Visual Basic Programming Guide

Walks through the essential elements of programming with Visual Basic.

Visual Basic Language Reference

Contains reference documentation for the Visual Basic language.

Programming in Visual Basic

4/2/2019 • 2 minutes to read • Edit Online

This section discusses programming tasks that you may want to learn more about as you create your Visual Basic application.

In this section

Accessing Computer Resources
Contains documentation on how to use the `My.Computer` object to access information about the computer on which an application runs and how to control the computer.

Logging Information from the Application
Contains documentation on how to log information from your application using the `My.Application.Log` and `My.Log` objects, and how to extend the application's logging capabilities.

Accessing User Data
Contains documentation on tasks that you can accomplish using the `My.User` object.

Accessing Application Forms
Contains documentation on accessing an application's forms by using the `My.Forms` and `My.Application` objects.

Accessing Application Web Services
Contains documentation on how to access the Web services referenced by the application using the `My.WebServices` object.

Accessing Application Settings
Contains documentation on accessing an application's settings using the `My.Settings` object.

Processing Drives, Directories, and Files
Contains documentation on how to access the file system using the `My.Computer.FileSystem` object.

See also

- Visual Basic Language Features
- Programming Concepts
- Collections
- Developing Applications with Visual Basic

Accessing computer resources (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

The `My.Computer` object is one of the three central objects in `My`, providing access to information and commonly used functionality. `My.Computer` provides methods, properties, and events for accessing the computer on which the application is running. Its objects include:

- Audio
- Clipboard (ClipboardProxy)
- Clock
- FileSystem
- Info
- Keyboard
- Mouse
- Network
- Ports
- Registry (RegistryProxy)

In this section

Playing Sounds
Lists tasks associated with `My.Computer.Audio`, such as playing a sound in the background.

Storing Data to and Reading from the Clipboard
Lists tasks associated with `My.Computer.Clipboard`, such as reading data from or writing data to the Clipboard.

Getting Information about the Computer
Lists tasks associated with `My.Computer.Info`, such as determining a computer's full name or IP addresses.

Accessing the Keyboard
Lists tasks associated with `My.Computer.Keyboard`, such as determining whether CAPS LOCK is on.

Accessing the Mouse
Lists tasks associated with `My.Computer.Mouse`, such as determining whether a mouse is present.

Performing Network Operations
Lists tasks associated with `My.Computer.Network`, such as uploading or downloading files.

Accessing the Computer's Ports
Lists tasks associated with `My.Computer.Ports`, such as showing available serial ports or sending strings to serial ports.

Reading from and Writing to the Registry
Lists tasks associated with `My.Computer.Registry`, such as reading data from or writing data to registry keys.

Logging Information from the Application (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

This section contains topics that cover how to log information from your application using the `My.Application.Log` or `My.Log` object, and how to extend the application's logging capabilities.

The `Log` object provides methods for writing information to the application's log listeners, and the `Log` object's advanced `TraceSource` property provides detailed configuration information. The `Log` object is configured by the application's configuration file.

The `My.Log` object is available only for ASP.NET applications. For client applications, use `My.Application.Log`. For more information, see Log.

Tasks

TO	SEE
Write event information to the application's logs.	How to: Write Log Messages
Write exception information to the application's logs.	How to: Log Exceptions
Write trace information to the application's logs when the application starts and shuts down.	How to: Log Messages When the Application Starts or Shuts Down
Configure `My.Application.Log` to write information to a text file.	How to: Write Event Information to a Text File
Configure `My.Application.Log` to write information to an event log.	How to: Write to an Application Event Log
Change where `My.Application.Log` writes information.	Walkthrough: Changing Where My.Application.Log Writes Information
Determine where `My.Application.Log` writes information.	Walkthrough: Determining Where My.Application.Log Writes Information
Create a custom log listener for `My.Application.Log`.	Walkthrough: Creating Custom Log Listeners
Filter the output of the `My.Application.Log` logs.	Walkthrough: Filtering My.Application.Log Output

See also

- Microsoft.VisualBasic.Logging.Log
- Working with Application Logs
- Troubleshooting: Log Listeners

Accessing User Data (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

This section contains topics dealing with the `My.User` object and tasks that you can accomplish with it.

The `My.User` object provides access to information about the logged-on user by returning an object that implements the IPrincipal interface.

Tasks

TO	SEE
Get the user's login name	Name
Get the user's domain name, if the application uses Windows authentication	CurrentPrincipal
Determine the user's role	IsInRole

See also

- User

Accessing Application Forms (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The `My.Forms` object provides an easy way to access an instance of each Windows Form declared in the application's project. You can also use properties of the `My.Application` object to access the application's splash screen and main form, and get a list of the application's open forms.

Tasks

The following table lists examples showing how to access an application's forms.

TO	SEE
Access one form from another form in an application.	My.Forms Object
Display the titles of all the application's open forms.	OpenForms
Update the splash screen with status information as the application starts.	SplashScreen

See also

- OpenForms
- SplashScreen
- My.Forms Object

Accessing Application Web Services (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The `My.WebServices` object provides an instance of each Web service referenced by the current project. Each instance is instantiated on demand. You can access these Web services through the properties of the `My.WebServices` object. The name of the property is the same as the name of the Web service that the property accesses. Any class that inherits from SoapHttpClientProtocol is a Web service.

Tasks

The following table lists possible ways to access Web services referenced by an application.

TO	SEE
Call a Web service	My.WebServices Object
Call a Web service asynchronously and handle an event when it completes	How to: Call a Web Service Asynchronously

See also

- My.WebServices Object

How to: Call a Web Service Asynchronously (Visual Basic)

3/5/2019 • 2 minutes to read • Edit Online

This example attaches a handler to a Web service's asynchronous handler event, so that it can retrieve the result of an asynchronous method call. This example used the DemoTemperatureService Web service at `http://www.xmethods.net`.

When you reference a Web service in your project in the Visual Studio Integrated Development Environment (IDE), it is added to the `My.WebServices` object, and the IDE generates a client proxy class to access a specified Web service

The proxy class allows you to call the Web service methods synchronously, where your application waits for the function to complete. In addition, the proxy creates additional members to help call the method asynchronously. For each Web service function, *NameOfWebServiceFunction*, the proxy creates a *NameOfWebServiceFunction* `Async` subroutine, a *NameOfWebServiceFunction* `Completed` event, and a *NameOfWebServiceFunction* `CompletedEventArgs` class. This example demonstrates how to use the asynchronous members to access the `getTemp` function of the DemoTemperatureService Web service.

> **NOTE**
>
> This code does not work in Web applications, because ASP.NET does not support the `My.WebServices` object.

To call a Web service asynchronously

1. Reference the DemoTemperatureService Web service at `http://www.xmethods.net`. The address is

   ```
   http://www.xmethods.net/sd/2001/DemoTemperatureService.wsdl
   ```

2. Add an event handler for the `getTempCompleted` event:

   ```
   Private Sub getTempCompletedHandler(ByVal sender As Object,
       ByVal e As net.xmethods.www.getTempCompletedEventArgs)

       MsgBox("Temperature: " & e.Result)
   End Sub
   ```

   > **NOTE**
   >
   > You cannot use the `Handles` statement to associate an event handler with the `My.WebServices` object's events.

3. Add a field to track if the event handler has been added to the `getTempCompleted` event:

   ```
   Private handlerAttached As Boolean = False
   ```

4. Add a method to add the event handler to the `getTempCompleted` event, if necessary, and to call the `getTempAsync` method:

```
Sub CallGetTempAsync(ByVal zipCode As Integer)
    If Not handlerAttached Then
        AddHandler My.WebServices.
            TemperatureService.getTempCompleted,
            AddressOf Me.TS_getTempCompleted
        handlerAttached = True
    End If
    My.WebServices.TemperatureService.getTempAsync(zipCode)
End Sub
```

To call the `getTemp` Web method asynchronously, call the `CallGetTempAsync` method. When the Web method finishes, its return value is passed to the `getTempCompletedHandler` event handler.

See also

- Accessing Application Web Services
- My.WebServices Object

Accessing application settings (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

This section contains topics describing the `My.Settings` object and the tasks it enables you to accomplish.

My.Settings

The properties of the `My.Settings` object provide access to your application's settings. To add or remove settings, use the **Settings** pane of the **Project Designer**.

The methods of the `My.Settings` object allow you to save the current user settings or revert the user settings to the last saved values.

Tasks

The following table lists examples showing how to access an application's forms.

TO	SEE
Update the value of a user setting	How to: Change User Settings in Visual Basic
Display application and user settings in a property grid	How to: Create Property Grids for User Settings in Visual Basic
Save updated user setting values	How to: Persist User Settings in Visual Basic
Determine the values of user settings	How to: Read Application Settings in Visual Basic

See also

- Managing Application Settings (.NET)
- My.Settings Object

2 minutes to read

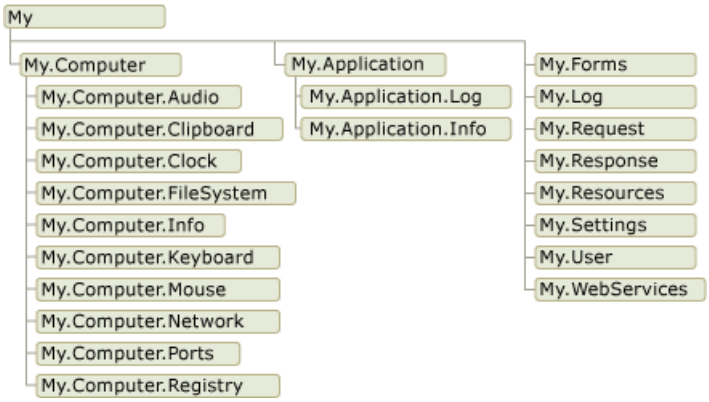
Development with My (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic provides new features for rapid application development that improve productivity and ease of use while delivering power. One of these features, called `My`, provides access to information and default object instances that are related to the application and its run-time environment. This information is organized in a format that is discoverable through IntelliSense and logically delineated according to use.

Top-level members of `My` are exposed as objects. Each object behaves similarly to a namespace or a class with `Shared` members, and it exposes a set of related members.

This table shows the top-level `My` objects and their relationship to each other.



In This Section

Performing Tasks with My.Application, My.Computer, and My.User
Describes the three central `My` objects, `My.Application`, `My.Computer`, and `My.User`, which provide access to information and functionality

Default Object Instances Provided by My.Forms and My.WebServices
Describes the `My.Forms` and `My.WebServices` objects, which provide access to forms, data sources, and XML Web services used by your application.

Rapid Application Development with My.Resources and My.Settings
Describes the `My.Resources` and `My.Settings` objects, which provide access to an application's resources and settings.

Overview of the Visual Basic Application Model
Describes the Visual Basic Application Startup/Shutdown model.

How My Depends on Project Type
Gives details on which `My` features are available in different project types.

See also

- ApplicationBase
- Computer
- User
- My.Forms Object
- My.WebServices Object

- How My Depends on Project Type

Performing Tasks with My.Application, My.Computer, and My.User (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The three central `My` objects that provide access to information and commonly used functionality are `My.Application` (ApplicationBase), `My.Computer` (Computer), and `My.User` (User). You can use these objects to access information that is related to the current application, the computer that the application is installed on, or the current user of the application, respectively.

My.Application, My.Computer, and My.User

The following examples demonstrate how information can be retrieved using `My`.

```
' Displays a message box that shows the full command line for the
' application.
Dim args As String = ""
For Each arg As String In My.Application.CommandLineArgs
    args &= arg & " "
Next
MsgBox(args)
```

```
' Gets a list of subfolders in a folder
My.Computer.FileSystem.GetDirectories(
  My.Computer.FileSystem.SpecialDirectories.MyDocuments, True, "*Logs*")
```

In addition to retrieving information, the members exposed through these three objects also allow you to execute methods related to that object. For instance, you can access a variety of methods to manipulate files or update the registry through `My.Computer`.

File I/O is significantly easier and faster with `My`, which includes a variety of methods and properties for manipulating files, directories, and drives. The TextFieldParser object allows you to read from large structured files that have delimited or fixed-width fields. This example opens the `TextFieldParser` `reader` and uses it to read from `C:\TestFolder1\test1.txt`.

```
Dim reader =
  My.Computer.FileSystem.OpenTextFieldParser("C:\TestFolder1\test1.txt")
reader.TextFieldType = Microsoft.VisualBasic.FileIO.FieldType.Delimited
reader.Delimiters = New String() {","}
Dim currentRow As String()
While Not reader.EndOfData
  Try
      currentRow = reader.ReadFields()
      Dim currentField As String
        For Each currentField In currentRow
            MsgBox(currentField)
        Next
        Catch ex As Microsoft.VisualBasic.FileIO.MalformedLineException
          MsgBox("Line " & ex.Message &
          "is not valid and will be skipped.")
    End Try
End While
```

`My.Application` allows you to change the culture for your application. The following example demonstrates how this method can be called.

```
' Changes the current culture for the application to Jamaican English.
My.Application.ChangeCulture("en-JM")
```

See also

- ApplicationBase
- Computer
- User
- How My Depends on Project Type

Default Object Instances Provided by My.Forms and My.WebServices (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The My.Forms and My.WebServices objects provide access to forms, data sources, and XML Web services used by your application. They do this by providing collections of *default instances* of each of these objects.

Default Instances

A default instance is an instance of the class that is provided by the runtime and does not need to be declared and instantiated using the `Dim` and `New` statements. The following example demonstrates how you might have declared and instantiated an instance of a Form class called `Form1`, and how you are now able to get a default instance of this Form class through `My.Forms`.

```
' The old method of declaration and instantiation
Dim myForm As New Form1
myForm.show()
```

```
' With My.Forms, you can directly call methods on the default
' instance()
My.Forms.Form1.Show()
```

The `My.Forms` object returns a collection of default instances for every `Form` class that exists in your project. Similarly, `My.WebServices` provides a default instance of the proxy class for every Web service that you have created a reference to in your application.

See also

- My.Forms Object
- My.WebServices Object
- How My Depends on Project Type

Rapid Application Development with My.Resources and My.Settings (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The `My.Resources` object provides access to the application's resources and allows you to dynamically retrieve resources for your application.

Retrieving Resources

A number of resources such as audio files, icons, images, and strings can be retrieved through the `My.Resources` object. For example, you can access the application's culture-specific resource files. The following example sets the icon of the form to the icon named `Form1Icon` stored in the application's resource file.

```
Sub SetFormIcon()
    Me.Icon = My.Resources.Form1Icon
End Sub
```

The `My.Resources` object exposes only global resources. It does not provide access to resource files associated with forms. You must access the form resources from the form.

Similarly, the `My.Settings` object provides access to the application's settings and allows you to dynamically store and retrieve property settings and other information for your application. For more information, see My.Resources Object and My.Settings Object.

See also

- My.Resources Object
- My.Settings Object
- Accessing Application Settings

Overview of the Visual Basic Application Model

4/28/2019 • 3 minutes to read • Edit Online

Visual Basic provides a well-defined model for controlling the behavior of Windows Forms applications: the Visual Basic Application model. This model includes events for handling the application's startup and shutdown, as well as events for catching unhandled exceptions. It also provides support for developing single-instance applications. The application model is extensible, so developers that need more control can customize its overridable methods.

Uses for the Application Model

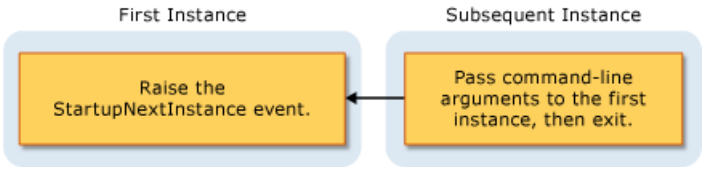
A typical application needs to perform tasks when it starts up and shuts down. For example, when it starts up, the application can display a splash screen, make database connections, load a saved state, and so on. When the application shuts down, it can close database connections, save the current state, and so on. In addition, the application can execute specific code when the application shuts down unexpectedly, such as during an unhandled exception.

The Visual Basic Application model makes it easy to create a *single-instance* application. A single-instance application differs from a normal application in that only one instance of the application can be running at a time. An attempt to launch another instance of a single-instance application results in the original instance being notified—by means of the `StartupNextInstance` event—that another launch attempt was made. The notification includes the subsequent instance's command-line arguments. The subsequent instance of the application is then closed before any initialization can occur.

A single-instance application starts and checks whether it is the first instance or a subsequent instance of the application:

- If it is the first instance, it starts as usual.
- Each subsequent attempt to start the application, while the first instance runs, results in very different behavior. The subsequent attempt notifies the first instance about the command-line arguments, and then immediately exits. The first instance handles the `StartupNextInstance` event to determine what the subsequent instance's command-line arguments were, and continues to run.

  This diagram shows how a subsequent instance signals the first instance:



By handling the `StartupNextInstance` event, you can control how your single-instance application behaves. For example, Microsoft Outlook typically runs as a single-instance application; when Outlook is running and you attempt to start Outlook again, focus shifts to the original instance but another instance does not open.

Events in the Application Model

The following events are found in the application model:

- **Application startup**. The application raises the Startup event when it starts. By handling this event, you can add code that initializes the application before the main form is loaded. The `Startup` event also provides for canceling execution of the application during that phase of the startup process, if desired.

You can configure the application to show a splash screen while the application startup code runs. By default, the application model suppresses the splash screen when either the `/nosplash` or `-nosplash` command-line argument is used.

- **Single-instance applications**. The StartupNextInstance event is raised when a subsequent instance of a single-instance application starts. The event passes the command-line arguments of the subsequent instance.
- **Unhandled exceptions**. If the application encounters an unhandled exception, it raises the UnhandledException event. Your handler for that event can examine the exception and determine whether to continue execution.

  The `UnhandledException` event is not raised in some circumstances. For more information, see UnhandledException.
- **Network-connectivity changes**. If the computer's network availability changes, the application raises the NetworkAvailabilityChanged event.

  The `NetworkAvailabilityChanged` event is not raised in some circumstances. For more information, see NetworkAvailabilityChanged.
- **Application shut down**. The application provides the Shutdown event to signal when it is about to shut down. In that event handler, you can make sure that the operations your application needs to perform—closing and saving, for example—are completed. You can configure your application to shut down when the main form closes, or to shut down only when all forms close.

Availability

By default, the Visual Basic Application model is available for Windows Forms projects. If you configure the application to use a different startup object, or start the application code with a custom `Sub Main`, then that object or class may need to provide an implementation of the WindowsFormsApplicationBase class to use the application model. For information about changing the startup object, see Application Page, Project Designer (Visual Basic).

See also

- WindowsFormsApplicationBase
- Startup
- StartupNextInstance
- UnhandledException
- Shutdown
- NetworkAvailabilityChanged
- WindowsFormsApplicationBase
- Extending the Visual Basic Application Model

How My Depends on Project Type (Visual Basic)

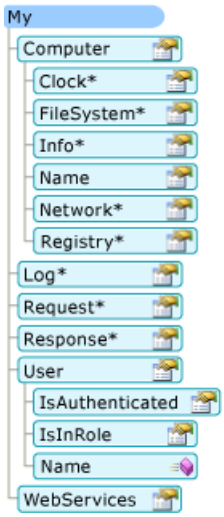
4/24/2019 • 2 minutes to read • Edit Online

`My` exposes only those objects required by a particular project type. For example, the `My.Forms` object is available in a Windows Forms application but not available in a console application. This topic describes which `My` objects are available in different project types.

My in Windows Applications and Web Sites

`My` exposes only objects that are useful in the current project type; it suppresses objects that are not applicable. For example, the following image shows the `My` object model in a Windows Forms project.

In a Web site project, `My` exposes objects that are relevant to a Web developer (such as the `My.Request` and `My.Response` objects) while suppressing objects that are not relevant (such as the `My.Forms` object). The following image shows the `My` object model in a Web site project:



Project Details

The following table shows which `My` objects are enabled by default for eight project types: Windows application, class Library, console application, Windows control library, Web control library, Windows service, empty, and Web site.

There are three versions of the `My.Application` object, two versions of the `My.Computer` object, and two versions of `My.User` object; details about these versions are given in the footnotes after the table.

MY OBJECT	WINDOWS APPLICATION	CLASS LIBRARY	CONSOLE APPLICATION	WINDOWS CONTROL LIBRARY	WEB CONTROL LIBRARY	WINDOWS SERVICE	EMPTY	WEB SITE
`My.Application`	**Yes** [1]	**Yes** [2]	**Yes** [3]	**Yes** [2]	No	**Yes** [3]	No	No
`My.Computer`	**Yes** [4]	**Yes** [4]	**Yes** [4]	**Yes** [4]	**Yes** [5]	**Yes** [4]	No	**Yes** [5]
`My.Forms`	**Yes**	No	No	**Yes**	No	No	No	No
`My.Log`	No	No	No	No	No	No	No	**Yes**
`My.Request`	No	No	No	No	No	No	No	**Yes**
`My.Resources`	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	No
`My.Response`	No	No	No	No	No	No	No	**Yes**
`My.Settings`	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	No
`My.User`	**Yes** [6]	**Yes** [6]	**Yes** [6]	**Yes** [6]	**Yes** [7]	**Yes** [6]	No	**Yes** [7]
`My.WebServices`	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No	No

[1] Windows Forms version of `My.Application`. Derives from the console version (see Note 3); adds support for interacting with the application's windows and provides the Visual Basic Application model.

[2] Library version of `My.Application`. Provides the basic functionality needed by an application: provides members for writing to the application log and accessing application information.

[3] Console version of `My.Application`. Derives from the library version (see Note 2), and adds additional members for accessing the application's command-line arguments and ClickOnce deployment information.

[4] Windows version of `My.Computer`. Derives from the Server version (see Note 5), and provides access to useful objects on a client machine, such as the keyboard, screen, and mouse.

[5] Server version of `My.Computer`. Provides basic information about the computer, such as the name, access to the clock, and so on.

[6] Windows version of `My.User`. This object is associated with the thread's current identity.

[7] Web version of `My.User`. This object is associated with the user identity of the application's current HTTP request.

See also

- ApplicationBase
- Computer

- Log
- User
- Customizing Which Objects are Available in My
- Conditional Compilation
- /define (Visual Basic)
- My.Forms Object
- My.Request Object
- My.Response Object
- My.WebServices Object

Accessing data in Visual Basic applications

9/7/2018 • 2 minutes to read • Edit Online

Visual Basic includes several new features to assist in developing applications that access data. Data-bound forms for Windows applications are created by dragging items from the Data Sources Window onto the form. You bind controls to data by dragging items from the **Data Sources Window** onto existing controls.

Related sections

Accessing Data in Visual Studio
Provides links to pages that discuss incorporating data access functionality into your applications.

Visual Studio data tools for .NET
Provides links to pages on creating applications that work with data, using Visual Studio.

LINQ
Provides links to topics that describe how to use LINQ with Visual Basic.

LINQ to SQL
Provides information about LINQ to SQL. Includes programming examples.

LINQ to SQL Tools in Visual Studio
Provides links to topics about how to create a LINQ to SQL object model in applications.

Work with datasets in n-tier applications
Provides links to topics about how to create multitiered data applications.

Add new connections
Provides links to pages on connecting your application to data with design-time tools and ADO.NET connection objects, using Visual Studio.

Dataset Tools in Visual Studio
Provides links to pages describing how to load data into datasets and how to execute SQL statements and stored procedures.

Bind controls to data in Visual Studio
Provides links to pages that explain how to display data on Windows Forms through data-bound controls.

Edit Data in Datasets
Provides links to pages describing how to manipulate the data in the data tables of a dataset.

Validate data in datasets
Provides links to pages describing how to add validation to a dataset during column and row changes.

Save data back to the database
Provides links to pages explaining how to send updated data from an application to the database.

ADO.NET
Describes the ADO.NET classes, which expose data-access services to the .NET Framework programmer.

Data in Office Solutions Contains links to pages that explain how data works in Office solutions, including information about schema-oriented programming, data caching, and server-side data access.

Creating and Using Components in Visual Basic

5/14/2019 • 2 minutes to read • Edit Online

A *component* is a class that implements the System.ComponentModel.IComponent interface or that derives directly or indirectly from a class that implements IComponent. A .NET Framework component is an object that is reusable, can interact with other objects, and provides control over external resources and design-time support.

An important feature of components is that they are designable, which means that a class that is a component can be used in the Visual Studio Integrated Development Environment. A component can be added to the Toolbox, dragged and dropped onto a form, and manipulated on a design surface. Notice that base design-time support for components is built into the .NET Framework; a component developer does not have to do any additional work to take advantage of the base design-time functionality.

A *control* is similar to a component, as both are designable. However, a control provides a user interface, while a component does not. A control must derive from one of the base control classes: Control or Control.

When to Create a Component

If your class will be used on a design surface (such as the Windows Forms or Web Forms Designer) but has no user interface, it should be a component and implement IComponent, or derive from a class that directly or indirectly implements IComponent.

The Component and MarshalByValueComponent classes are base implementations of the IComponent interface. The main difference between these classes is that the Component class is marshaled by reference, while IComponent is marshaled by value. The following list provides broad guidelines for implementers.

- If your component needs to be marshaled by reference, derive from Component.
- If your component needs to be marshaled by value, derive from MarshalByValueComponent.
- If your component cannot derive from one of the base implementations due to single inheritance, implement IComponent.

Component Classes

The System.ComponentModel namespace provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components.

The core component classes are:

- Component. A base implementation for the IComponent interface. This class enables object sharing between applications.
- MarshalByValueComponent. A base implementation for the IComponent interface.
- Container. The base implementation for the IContainer interface. This class encapsulates zero or more components.

Some of the classes used for component licensing are:

- License. The abstract base class for all licenses. A license is granted to a specific instance of a component.
- LicenseManager. Provides properties and methods to add a license to a component and to manage a

LicenseProvider.

- LicenseProvider. The abstract base class for implementing a license provider.
- LicenseProviderAttribute. Specifies the LicenseProvider class to use with a class.

Classes commonly used for describing and persisting components.

- TypeDescriptor. Provides information about the characteristics for a component, such as its attributes, properties, and events.
- EventDescriptor. Provides information about an event.
- PropertyDescriptor. Provides information about a property.

Related Sections

Troubleshooting Control and Component Authoring
Explains how to fix common problems.

See also

- How to: Access Design-Time Support in Windows Forms

Windows Forms Application Basics (Visual Basic)

7/1/2019 • 6 minutes to read • Edit Online

An important part of Visual Basic is the ability to create Windows Forms applications that run locally on users' computers. You can use Visual Studio to create the application and user interface using Windows Forms. A Windows Forms application is built on classes from the System.Windows.Forms namespace.

Designing Windows Forms Applications

You can create Windows Forms and Windows service applications with Visual Studio. For more information, see the following topics:

- Getting Started with Windows Forms. Provides information on how to create and program Windows Forms.
- Windows Forms Controls. Collection of topics detailing the use of Windows Forms controls.
- Windows Service Applications. Lists topics that explain how to create Windows services.

Building Rich, Interactive User Interfaces

Windows Forms is the smart-client component of the .NET Framework, a set of managed libraries that enable common application tasks such as reading and writing to the file system. Using a development environment like Visual Studio, you can create Windows Forms applications that display information, request input from users, and communicate with remote computers over a network.

In Windows Forms, a form is a visual surface on which you display information to the user. You commonly build Windows Forms applications by placing controls on forms and developing responses to user actions, such as mouse clicks or key presses. A *control* is a discrete user interface (UI) element that displays data or accepts data input.

Events

When a user does something to your form or one of its controls, it generates an event. Your application reacts to these events by using code, and processes the events when they occur. For more information, see Creating Event Handlers in Windows Forms.

Controls

Windows Forms contains a variety of controls that you can place on forms: controls that display text boxes, buttons, drop-down boxes, radio buttons, and even Web pages. For a list of all the controls you can use on a form, see Controls to Use on Windows Forms. If an existing control does not meet your needs, Windows Forms also supports creating your own custom controls using the UserControl class.

Windows Forms has rich UI controls that emulate features in high-end applications like Microsoft Office. Using the ToolStrip and MenuStrip control, you can create toolbars and menus that contain text and images, display submenus, and host other controls such as text boxes and combo boxes.

With the Visual Studio drag-and-drop forms designer, you can easily create Windows Forms applications: just select the controls with your cursor and place them where you want on the form. The designer provides tools such as grid lines and "snap lines" to take the hassle out of aligning controls. And whether you use Visual Studio or compile at the command line, you can use the FlowLayoutPanel, TableLayoutPanel and SplitContainer controls to create advanced form layouts with minimal time and effort.

Custom UI Elements

Finally, if you must create your own custom UI elements, the System.Drawing namespace contains all of the classes you need to render lines, circles, and other shapes directly on a form.

For step-by-step information about using these features, see the following Help topics.

TO	SEE
Create a new Windows Forms application with Visual Studio	Tutorial 1: Create a picture viewer
Use controls on forms	How to: Add Controls to Windows Forms
Create graphics with System.Drawing	Getting Started with Graphics Programming
Create custom controls	How to: Inherit from the UserControl Class

Displaying and Manipulating Data

Many applications must display data from a database, XML file, XML Web service, or other data source. Windows Forms provides a flexible control called the DataGridView control for rendering such tabular data in a traditional row and column format, so that every piece of data occupies its own cell. Using DataGridView you can customize the appearance of individual cells, lock arbitrary rows and columns in place, and display complex controls inside cells, among other features.

Connecting to data sources over a network is a simple task with Windows Forms smart clients. The BindingSource component, new with Windows Forms in Visual Studio 2005 and the .NET Framework 2.0, represents a connection to a data source, and exposes methods for binding data to controls, navigating to the previous and next records, editing records, and saving changes back to the original source. The BindingNavigator control provides a simple interface over the BindingSource component for users to navigate between records.

Data-Bound Controls

You can create data-bound controls easily using the Data Sources window, which displays data sources such as databases, Web services, and objects in your project. You can create data-bound controls by dragging items from this window onto forms in your project. You can also data-bind existing controls to data by dragging objects from the Data Sources window onto existing controls.

Settings

Another type of data binding you can manage in Windows Forms is settings. Most smart-client applications must retain some information about their run-time state, such as the last-known size of forms, and retain user-preference data, such as default locations for saved files. The application-settings feature addresses these requirements by providing an easy way to store both types of settings on the client computer. Once defined using either Visual Studio or a code editor, these settings are persisted as XML and automatically read back into memory at run time.

For step-by-step information about using these features, see the following Help topics.

TO	SEE
Use the BindingSource component	How to: Bind Windows Forms Controls with the BindingSource Component Using the Designer
Work with ADO.NET data sources	How to: Sort and Filter ADO.NET Data with the Windows Forms BindingSource Component

TO	SEE
Use the Data Sources window	Walkthrough: Displaying Data on a Windows Form

Deploying Applications to Client Computers

Once you have written your application, you must send it to your users so that they can install and run it on their own client computers. Using the ClickOnce technology, you can deploy your applications from within Visual Studio by using just a few clicks and provide users with a URL pointing to your application on the Web. ClickOnce manages all of the elements and dependencies in your application and ensures that the application is properly installed on the client computer.

ClickOnce applications can be configured to run only when the user is connected to the network, or to run both online and offline. When you specify that an application should support offline operation, ClickOnce adds a link to your application in the user's **Start** menu, so that the user can open it without using the URL.

When you update your application, you publish a new deployment manifest and a new copy of your application to your Web server. ClickOnce detects that there is an update available and upgrades the user's installation; no custom programming is required to update old assemblies.

For a full introduction to ClickOnce, see ClickOnce Security and Deployment. For step-by-step information about using these features, see the following Help topics:

TO	SEE
Deploy an application with ClickOnce	How to: Publish a ClickOnce Application using the Publish Wizard Walkthrough: Manually Deploying a ClickOnce Application
Update a ClickOnce deployment	How to: Manage Updates for a ClickOnce Application
Manage security with ClickOnce	How to: Enable ClickOnce Security Settings

Other Controls and Features

There are many other features in Windows Forms that make implementing common tasks fast and easy, such as support for creating dialog boxes, printing, adding Help and documentation, and localizing your application to multiple languages. In addition, Windows Forms relies on the robust security system of the .NET Framework, enabling you to release more secure applications to your customers.

For step-by-step information about using these features, see the following Help topics:

TO	SEE
Print the contents of a form	How to: Print Graphics in Windows Forms How to: Print a Multi-Page Text File in Windows Forms
Learn more about Windows Forms security	Security in Windows Forms Overview

See also

- WindowsFormsApplicationBase

- Windows Forms Overview
- My.Forms Object

Customizing Projects and Extending My with Visual Basic

4/3/2019 • 2 minutes to read • Edit Online

You can customize project templates to provide additional `My` objects. This makes it easy for other developers to find and use your objects.

In This Section

Extending the My Namespace in Visual Basic
Describes how to add custom members and values to the `My` namespace in Visual Basic.

Packaging and Deploying Custom My Extensions
Describes how to publish custom `My` namespace extensions by using Visual Studio templates.

Extending the Visual Basic Application Model
Describes how to specify your own extensions to the application model by overriding members of the WindowsFormsApplicationBase class.

Customizing Which Objects are Available in My
Describes how to control which `My` objects are enabled by setting your project's _MYTYPE conditional-compilation constant.

Related Sections

Development with My
Describes which `My` objects are available in different project types by default.

Overview of the Visual Basic Application Model
Describes Visual Basic's model for controlling the behavior of Windows Forms applications.

How My Depends on Project Type
Describes which `My` objects are available in different project types by default.

Conditional Compilation
Discusses how the compiler uses conditional-compilation to select particular sections of code to compile and exclude other sections.

ApplicationBase
Describes the `My` object that provides properties, methods, and events related to the current application.

See also

- Developing Applications with Visual Basic

Extending the My Namespace in Visual Basic

4/28/2019 • 8 minutes to read • Edit Online

The `My` namespace in Visual Basic exposes properties and methods that enable you to easily take advantage of the power of the .NET Framework. The `My` namespace simplifies common programming problems, often reducing a difficult task to a single line of code. Additionally, the `My` namespace is fully extensible so that you can customize the behavior of `My` and add new services to its hierarchy to adapt to specific application needs. This topic discusses both how to customize existing members of the `My` namespace and how to add your own custom classes to the `My` namespace.

Topic Contents

- Customizing Existing My Namespace Members
- Adding Members to My Objects
- Adding Custom Objects to the My Namespace
- Adding Members to the My Namespace
- Adding Events to Custom My Objects
- Design Guidelines
- Designing Class Libraries for My
- Packaging and Deploying Extensions

Customizing Existing My Namespace Members

The `My` namespace in Visual Basic exposes frequently used information about your application, your computer, and more. For a complete list of the objects in the `My` namespace, see My Reference. You may have to customize existing members of the `My` namespace so that they better match the needs of your application. Any property of an object in the `My` namespace that is not read-only can be set to a custom value.

For example, assume that you frequently use the `My.User` object to access the current security context for the user running your application. However, your company uses a custom user object to expose additional information and capabilities for users within the company. In this scenario, you can replace the default value of the `My.User.CurrentPrincipal` property with an instance of your own custom principal object, as shown in the following example.

```
My.User.CurrentPrincipal = CustomPrincipal
```

Setting the `CurrentPrincipal` property on the `My.User` object changes the identity under which the application runs. The `My.User` object, in turn, returns information about the newly specified user.

Adding Members to My Objects

The types returned from `My.Application` and `My.Computer` are defined as `Partial` classes. Therefore, you can extend the `My.Application` and `My.Computer` objects by creating a `Partial` class named `MyApplication` or `MyComputer`. The class cannot be a `Private` class. If you specify the class as part of the `My` namespace, you can add properties and methods that will be included with the `My.Application` or `My.Computer` objects.

For example, the following example adds a property named `DnsServerIPAddresses` to the `My.Computer` object.

```
Imports System.Net.NetworkInformation

Namespace My

  Partial Class MyComputer
    Friend ReadOnly Property DnsServerIPAddresses() As IPAddressCollection
      Get
        Dim dnsAddressList As IPAddressCollection = Nothing

        For Each adapter In System.Net.NetworkInformation.
          NetworkInterface.GetAllNetworkInterfaces()

          Dim adapterProperties = adapter.GetIPProperties()
          Dim dnsServers As IPAddressCollection = adapterProperties.DnsAddresses
          If dnsAddressList Is Nothing Then
            dnsAddressList = dnsServers
          Else
            dnsAddressList.Union(dnsServers)
          End If
        Next adapter

        Return dnsAddressList
      End Get
    End Property
  End Class

End Namespace
```

Adding Custom Objects to the My Namespace

Although the `My` namespace provides solutions for many common programming tasks, you may encounter tasks that the `My` namespace does not address. For example, your application might access custom directory services for user data, or your application might use assemblies that are not installed by default with Visual Basic. You can extend the `My` namespace to include custom solutions to common tasks that are specific to your environment. The `My` namespace can easily be extended to add new members to meet growing application needs. Additionally, you can deploy your `My` namespace extensions to other developers as a Visual Basic template.

Adding Members to the My Namespace

Because `My` is a namespace like any other namespace, you can add top-level properties to it by just adding a module and specifying a `Namespace` of `My`. Annotate the module with the `HideModuleName` attribute as shown in the following example. The `HideModuleName` attribute ensures that IntelliSense will not display the module name when it displays the members of the `My` namespace.

```
Namespace My
  <HideModuleName()>
  Module MyCustomModule

  End Module
End Namespace
```

To add members to the `My` namespace, add properties as needed to the module. For each property added to the `My` namespace, add a private field of type `ThreadSafeObjectProvider(Of T)`, where the type is the type returned by your custom property. This field is used to create thread-safe object instances to be returned by the property by calling the `GetInstance` method. As a result, each thread that is accessing the extended property receives its own instance of the returned type. The following example adds a property named `SampleExtension` that is of type `SampleExtension` to the `My` namespace:

```
Namespace My
  <HideModuleName()>
  Module MyCustomExtensions
    Private _extension As New ThreadSafeObjectProvider(Of SampleExtension)
    Friend ReadOnly Property SampleExtension() As SampleExtension
      Get
        Return _extension.GetInstance()
      End Get
    End Property
  End Module
End Namespace
```

Adding Events to Custom My Objects

You can use the `My.Application` object to expose events for your custom `My` objects by extending the `MyApplication` partial class in the `My` namespace. For Windows-based projects, you can double-click the **My Project** node in for your project in **Solution Explorer**. In the Visual Basic **Project Designer**, click the `Application` tab and then click the `View Application Events` button. A new file that is named ApplicationEvents.vb will be created. It contains the following code for extending the `MyApplication` class.

```
Namespace My
  Partial Friend Class MyApplication
  End Class
End Namespace
```

You can add event handlers for your custom `My` objects by adding custom event handlers to the `MyApplication` class. Custom events enable you to add code that will execute when an event handler is added, removed, or the event is raised. Note that the `AddHandler` code for a custom event runs only if code is added by a user to handle the event. For example, consider that the `SampleExtension` object from the previous section has a `Load` event that you want to add a custom event handler for. The following code example shows a custom event handler named `SampleExtensionLoad` that will be invoked when the `My.SampleExtension.Load` event occurs. When code is added to handle the new `My.SampleExtensionLoad` event, the `AddHandler` part of this custom event code is executed. The `MyApplication_SampleExtensionLoad` method is included in the code example to show an example of an event handler that handles the `My.SampleExtensionLoad` event. Note that the `SampleExtensionLoad` event will be available when you select the **My Application Events** option in the left drop-down list above the Code Editor when you are editing the ApplicationEvents.vb file.

```
Namespace My

  Partial Friend Class MyApplication

    ' Custom event handler for Load event.
    Private _sampleExtensionHandlers As EventHandler

    Public Custom Event SampleExtensionLoad As EventHandler
      AddHandler(ByVal value As EventHandler)
        ' Warning: This code is not thread-safe. Do not call
        ' this code from multiple concurrent threads.
        If _sampleExtensionHandlers Is Nothing Then
          AddHandler My.SampleExtension.Load, AddressOf OnSampleExtensionLoad
        End If
        _sampleExtensionHandlers =
            System.Delegate.Combine(_sampleExtensionHandlers, value)
      End AddHandler
      RemoveHandler(ByVal value As EventHandler)
        _sampleExtensionHandlers =
          System.Delegate.Remove(_sampleExtensionHandlers, value)
      End RemoveHandler
      RaiseEvent(ByVal sender As Object, ByVal e As EventArgs)
        If _sampleExtensionHandlers IsNot Nothing Then
          _sampleExtensionHandlers.Invoke(sender, e)
        End If
      End RaiseEvent
    End Event

    ' Method called by custom event handler to raise user-defined
    ' event handlers.
    <Global.System.ComponentModel.EditorBrowsable(
         Global.System.ComponentModel.EditorBrowsableState.Advanced)>
      Protected Overridable Sub OnSampleExtensionLoad(
                ByVal sender As Object, ByVal e As EventArgs)
      RaiseEvent SampleExtensionLoad(sender, e)
    End Sub

    ' Event handler to call My.SampleExtensionLoad event.
    Private Sub MyApplication_SampleExtensionLoad(
        ByVal sender As Object, ByVal e As System.EventArgs
        ) Handles Me.SampleExtensionLoad

    End Sub
  End Class
End Namespace
```

Design Guidelines

When you develop extensions to the `My` namespace, use the following guidelines to help minimize the maintenance costs of your extension components.

- **Include only the extension logic.** The logic included in the `My` namespace extension should include only the code that is needed to expose the required functionality in the `My` namespace. Because your extension will reside in user projects as source code, updating the extension component incurs a high maintenance cost and should be avoided if possible.
- **Minimize project assumptions.** When you create your extensions of the `My` namespace, do not assume a set of references, project-level imports, or specific compiler settings (for example, `Option Strict` off). Instead, minimize dependencies and fully qualify all type references by using the `Global` keyword. Also, ensure that the extension compiles with `Option Strict` on to minimize errors in the extension.
- **Isolate the extension code.** Placing the code in a single file makes your extension easily deployable as a

Visual Studio item template. For more information, see "Packaging and Deploying Extensions" later in this topic. Placing all the `My` namespace extension code in a single file or a separate folder in a project will also help users locate the `My` namespace extension.

Designing Class Libraries for My

As is the case with most object models, some design patterns work well in the `My` namespace and others do not. When designing an extension to the `My` namespace, consider the following principles:

- **Stateless methods.** Methods in the `My` namespace should provide a complete solution to a specific task. Ensure that the parameter values that are passed to the method provide all the input required to complete the particular task. Avoid creating methods that rely on prior state, such as open connections to resources.
- **Global instances.** The only state that is maintained in the `My` namespace is global to the project. For example, `My.Application.Info` encapsulates state that is shared throughout the application.
- **Simple parameter types.** Keep things simple by avoiding complex parameter types. Instead, create methods that either take no parameter input or that take simple input types such as strings, primitive types, and so on.
- **Factory methods.** Some types are necessarily difficult to instantiate. Providing factory methods as extensions to the `My` namespace enables you to more easily discover and consume types that fall into this category. An example of a factory method that works well is `My.Computer.FileSystem.OpenTextFileReader`. There are several stream types available in the .NET Framework. By specifying text files specifically, the `OpenTextFileReader` helps the user understand which stream to use.

These guidelines do not preclude general design principles for class libraries. Rather, they are recommendations that are optimized for developers who are using Visual Basic and the `My` namespace. For general design principles for creating class libraries, see Framework Design Guidelines.

Packaging and Deploying Extensions

You can include `My` namespace extensions in a Visual Studio project template, or you can package your extensions and deploy them as a Visual Studio item template. When you package your `My` namespace extensions as a Visual Studio item template, you can take advantage of additional capabilities provided by Visual Basic. These capabilities enable you to include an extension when a project references a particular assembly, or enable users to explicitly add your `My` namespace extension by using the **My Extensions** page of the Visual Basic Project Designer.

For details about how to deploy `My` namespace extensions, see Packaging and Deploying Custom My Extensions.

See also

- Packaging and Deploying Custom My Extensions
- Extending the Visual Basic Application Model
- Customizing Which Objects are Available in My
- My Extensions Page, Project Designer
- Application Page, Project Designer (Visual Basic)
- Partial

Package and deploy custom My extensions (Visual Basic)

8/19/2018 • 4 minutes to read • Edit Online

Visual Basic provides an easy way for you to deploy your custom `My` namespace extensions by using Visual Studio templates. If you are creating a project template for which your `My` extensions are an integral part of the new project type, you can just include your custom `My` extension code with the project when you export the template. For more information about exporting project templates, see How to: Create Project Templates.

If your custom `My` extension is in a single code file, you can export the file as an item template that users can add to any type of Visual Basic project. You can then customize the item template to enable additional capabilities and behavior for your custom `My` extension in a Visual Basic project. Those capabilities include the following:

- Allowing users to manage your custom `My` extension from the **My Extensions** page of the Visual Basic Project Designer.
- Automatically adding your custom `My` extension when a reference to a specified assembly is added to a project.
- Hiding the `My` extension item template in the **Add Item** dialog box so that it is not included in the list of project items.

This topic discusses how to package a custom `My` extension as a hidden item template that can be managed from the **My Extensions** page of the Visual Basic Project Designer. The custom `My` extension can also be added automatically when a reference to a specified assembly is added to a project.

Create a My namespace extension

The first step in creating a deployment package for a custom `My` extension is to create the extension as a single code file. For details and guidance about how to create a custom `My` extension, see Extending the My Namespace in Visual Basic.

Export a My namespace extension as an item template

After you have a code file that includes your `My` namespace extension, you can export the code file as a Visual Studio item template. For instructions on how to export a file as a Visual Studio item template, see How to: Create Item Templates.

> **NOTE**
>
> If your `My` namespace extension has a dependency on a particular assembly, you can customize your item template to automatically install your `My` namespace extension when a reference to that assembly is added. As a result, you will want to exclude that assembly reference when you export the code file as a Visual Studio item template.

Customize the item template

You can enable your item template to be managed from the **My Extensions** page of the Visual Basic Project Designer. You can also enable the item template to be added automatically when a reference to a specified assembly is added to a project. To enable these customizations, you will add a new file, called the CustomData file,

to your template, and then add a new element to the XML in your .vstemplate file.

Add the CustomData file

The CustomData file is a text file that has a file name extension of .CustomData (the file name can be set to any value meaningful to your template) and that contains XML. The XML in the CustomData file instructs Visual Basic to include your `My` extension when users use the **My Extensions** page of the Visual Basic Project Designer. You can optionally add the < `AssemblyFullName>` attribute to your CustomData file XML. This instructs Visual Basic to automatically install your custom `My` extension when a reference to a particular assembly is added to the project. You can use any text editor or XML editor to create the CustomData file, and then add it to your item template's compressed folder (.zip file).

For example, the following XML shows the contents of a CustomData file that will add the template item to the My Extensions folder of a Visual Basic project when a reference to the Microsoft.VisualBasic.PowerPacks.Vs.dll assembly is added to the project.

```
<VBMyExtensionTemplate
    ID="Microsoft.VisualBasic.Samples.MyExtensions.MyPrinterInfo"
    Version="1.0.0.0"
    AssemblyFullName="Microsoft.VisualBasic.PowerPacks.vs"
/>
```

The CustomData file contains a < `VBMyExtensionTemplate>` element that has attributes as listed in the following table.

ATTRIBUTE	DESCRIPTION
`ID`	Required. A unique identifier for the extension. If the extension that has this ID has already been added to the project, the user will not be prompted to add it again.
`Version`	Required. A version number for the item template.
`AssemblyFullName`	Optional. An assembly name. When a reference to this assembly is added to the project, the user will be prompted to add the `My` extension from this item template.

Add the <CustomDataSignature> element to the .vstemplate file

To identify your Visual Studio item template as a `My` namespace extension, you must also modify the .vstemplate file for your item template. You must add a `<CustomDataSignature>` element to the `<TemplateData>` element. The `<CustomDataSignature>` element must contain the text `Microsoft.VisualBasic.MyExtension`, as shown in the following example.

```
<CustomDataSignature>Microsoft.VisualBasic.MyExtension</CustomDataSignature>
```

You cannot modify files in a compressed folder (.zip file) directly. You must copy the .vstemplate file from the compressed folder, modify it, and then replace the .vstemplate file in the compressed folder with your updated copy.

The following example shows the contents of a .vstemplate file that has the `<CustomDataSignature>` element added.

```
<VSTemplate Version="2.0.0" xmlns="http://schemas.microsoft.com/developer/vstemplate/2005" Type="Item">
  <TemplateData>
    <DefaultName>MyCustomExtensionModule.vb</DefaultName>
    <Name>MyPrinterInfo</Name>
    <Description>Custom My Extensions Item Template</Description>
    <ProjectType>VisualBasic</ProjectType>
    <SortOrder>10</SortOrder>
    <Icon>__TemplateIcon.ico</Icon>
    <CustomDataSignature      >Microsoft.VisualBasic.MyExtension</CustomDataSignature>
  </TemplateData>
  <TemplateContent>
    <References />
    <ProjectItem SubType="Code"
                 TargetFileName="$fileinputname$.vb"
                 ReplaceParameters="true"
     >MyCustomExtensionModule.vb</ProjectItem>
  </TemplateContent>
</VSTemplate>
```

Install the template

To install the template, you can copy the compressed folder (*.zip* file) to the Visual Basic item templates folder. By default, user item templates are located in *%USERPROFILE%\Documents\Visual Studio <Version>\Templates\ItemTemplates\Visual Basic*. Alternatively, you can publish the template as a Visual Studio Installer (*.vsi*) file.

See also

- Extending the My Namespace in Visual Basic
- Extending the Visual Basic Application Model
- Customizing Which Objects are Available in My
- My Extensions Page, Project Designer

Extending the Visual Basic Application Model

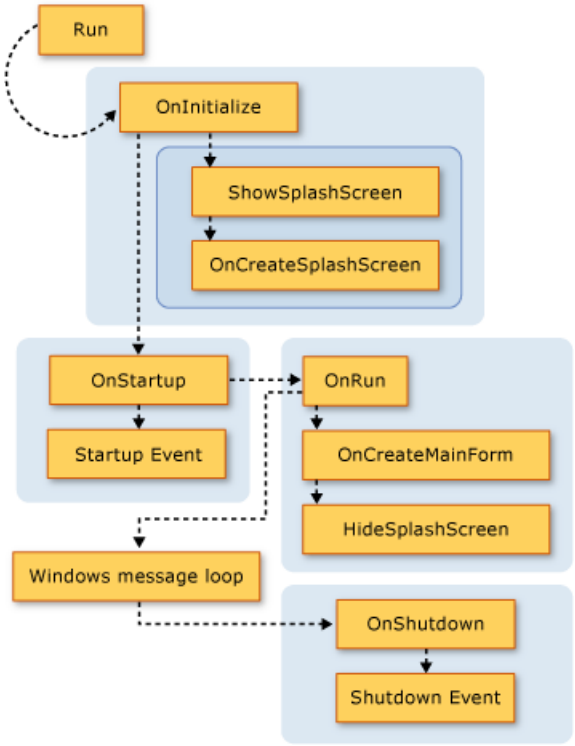
5/9/2019 • 4 minutes to read • Edit Online

You can add functionality to the application model by overriding the `Overridable` members of the WindowsFormsApplicationBase class. This technique allows you to customize the behavior of the application model and add calls to your own methods as the application starts up and shuts down.

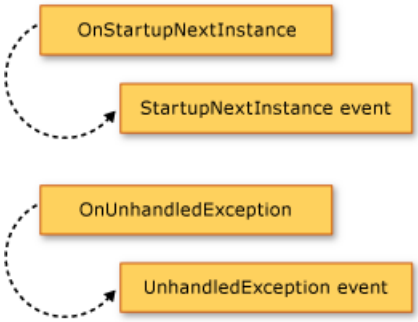
Visual Overview of the Application Model

This section visually presents the sequence of function calls in the Visual Basic Application Model. The next section describes the purpose of each function in detail.

The following graphic shows the application model call sequence in a normal Visual Basic Windows Forms application. The sequence starts when the `Sub Main` procedure calls the Run method.



The Visual Basic Application Model also provides the StartupNextInstance and UnhandledException events. The following graphics show the mechanism for raising these events.



Overriding the Base Methods

The Run method defines the order in which the `Application` methods run. By default, the `Sub Main` procedure for

a Windows Forms application calls the Run method.

If the application is a normal application (multiple-instance application), or the first instance of a single-instance application, the Run method executes the `Overridable` methods in the following order:

1. OnInitialize. By default, this method sets the visual styles, text display styles, and current principal for the main application thread (if the application uses Windows authentication), and calls `ShowSplashScreen` if neither `/nosplash` nor `-nosplash` is used as a command-line argument.

   The application startup sequence is canceled if this function returns `False`. This can be useful if there are circumstances in which the application should not run.

   The OnInitialize method calls the following methods:

   a. ShowSplashScreen. Determines if the application has a splash screen defined and if it does, displays the splash screen on a separate thread.

      The ShowSplashScreen method contains the code that displays the splash screen for at least the number of milliseconds specified by the MinimumSplashScreenDisplayTime property. To use this functionality, you must add the splash screen to your application using the **Project Designer** (which sets the `My.Application.MinimumSplashScreenDisplayTime` property to two seconds), or set the `My.Application.MinimumSplashScreenDisplayTime` property in a method that overrides the OnInitialize or OnCreateSplashScreen method. For more information, see MinimumSplashScreenDisplayTime.

   b. OnCreateSplashScreen. Allows a designer to emit code that initializes the splash screen.

      By default, this method does nothing. If you select a splash screen for your application in the Visual Basic **Project Designer**, the designer overrides the OnCreateSplashScreen method with a method that sets the SplashScreen property to a new instance of the splash-screen form.

2. OnStartup. Provides an extensibility point for raising the `Startup` event. The application startup sequence stops if this function returns `False`.

   By default, this method raises the Startup event. If the event handler sets the Cancel property of the event argument to `True`, the method returns `False` to cancel the application startup.

3. OnRun. Provides the starting point for when the main application is ready to start running, after the initialization is done.

   By default, before it enters the Windows Forms message loop, this method calls the `OnCreateMainForm` (to create the application's main form) and `HideSplashScreen` (to close the splash screen) methods:

   a. OnCreateMainForm. Provides a way for a designer to emit code that initializes the main form.

      By default, this method does nothing. However, when you select a main form for your application in the Visual Basic **Project Designer**, the designer overrides the OnCreateMainForm method with a method that sets the MainForm property to a new instance of the main form.

   b. HideSplashScreen. If application has a splash screen defined and it is open, this method closes the splash screen.

      By default, this method closes the splash screen.

4. OnStartupNextInstance. Provides a way to customize how a single-instance application behaves when another instance of the application starts.

   By default, this method raises the StartupNextInstance event.

5. OnShutdown. Provides an extensibility point for raising the `Shutdown` event. This method does not run if an unhandled exception occurs in the main application.

By default, this method raises the Shutdown event.

6. OnUnhandledException. Executed if an unhandled exception occurs in any of the above listed methods.

   By default, this method raises the UnhandledException event as long as a debugger is not attached and the application is handling the `UnhandledException` event.

If the application is a single-instance application, and the application is already running, the subsequent instance of the application calls the OnStartupNextInstance method on the original instance of the application, and then exits.

The OnStartupNextInstance(StartupNextInstanceEventArgs) constructor calls the UseCompatibleTextRendering property to determine which text rendering engine to use for the application's forms. By default, the UseCompatibleTextRendering property returns `False`, indicating that the GDI text rendering engine be used, which is the default in Visual Basic 2005 and later versions. You can override the UseCompatibleTextRendering property to return `True`, which indicates that the GDI+ text rendering engine be used, which is the default in Visual Basic .NET 2002 and Visual Basic .NET 2003.

Configuring the Application

As a part of the Visual Basic Application model, the WindowsFormsApplicationBase class provides protected properties that configure the application. These properties should be set in the constructor of the implementing class.

In a default Windows Forms project, the **Project Designer** creates code to set the properties with the designer settings. The properties are used only when the application is starting; setting them after the application starts has no effect.

PROPERTY	DETERMINES	SETTING IN THE APPLICATION PANE OF THE PROJECT DESIGNER
IsSingleInstance	Whether the application runs as a single-instance or multiple-instance application.	**Make single instance application** check box
EnableVisualStyles	If the application will use visual styles that match Windows XP.	**Enable XP visual styles** check box
SaveMySettingsOnExit	If application automatically saves application's user-settings changes when the application exits.	**Save My.Settings on Shutdown** check box
ShutdownStyle	What causes the application to terminate, such as when the startup form closes or when the last form closes.	**Shutdown mode** list

See also

- ApplicationBase
- Startup
- StartupNextInstance
- UnhandledException
- Shutdown
- NetworkAvailabilityChanged

- Overview of the Visual Basic Application Model
- Application Page, Project Designer (Visual Basic)

Customizing Which Objects are Available in My (Visual Basic)

4/3/2019 • 2 minutes to read • Edit Online

This topic describes how you can control which `My` objects are enabled by setting your project's `_MYTYPE` conditional-compilation constant. The Visual Studio Integrated Development Environment (IDE) keeps the `_MYTYPE` conditional-compilation constant for a project in sync with the project's type.

Predefined _MYTYPE Values

You must use the `/define` compiler option to set the `_MYTYPE` conditional-compilation constant. When specifying your own value for the `_MYTYPE` constant, you must enclose the string value in backslash/quotation mark (\") sequences. For example, you could use:

```
/define:_MYTYPE=\"WindowsForms\"
```

This table shows what the `_MYTYPE` conditional-compilation constant is set to for several project types.

PROJECT TYPE	_MYTYPE VALUE
Class Library	"Windows"
Console Application	"Console"
Web	"Web"
Web Control Library	"WebControl"
Windows Application	"WindowsForms"
Windows Application, when starting with custom `Sub Main`	"WindowsFormsWithCustomSubMain"
Windows Control Library	"Windows"
Windows Service	"Console"
Empty	"Empty"

NOTE

All conditional-compilation string comparisons are case-sensitive, regardless of how the `Option Compare` statement is set.

Dependent _MY Compilation Constants

The `_MYTYPE` conditional-compilation constant, in turn, controls the values of several other `_MY` compilation constants:

_MYTYPE	_MYAPPLICATIONTYPE	_MYCOMPUTERTYPE	_MYFORMS	_MYUSERTYPE	_MYWEBSERVICES
"Console"	"Console"	"Windows"	Undefined	"Windows"	TRUE
"Custom"	Undefined	Undefined	Undefined	Undefined	Undefined
"Empty"	Undefined	Undefined	Undefined	Undefined	Undefined
"Web"	Undefined	"Web"	FALSE	"Web"	FALSE
"WebControl"	Undefined	"Web"	FALSE	"Web"	TRUE
"Windows" or ""	"Windows"	"Windows"	Undefined	"Windows"	TRUE
"WindowsForms"	"WindowsForms"	"Windows"	TRUE	"Windows"	TRUE
"WindowsFormsWithCustomSubMain"	"Console"	"Windows"	TRUE	"Windows"	TRUE

By default, undefined conditional-compilation constants resolve to `FALSE`. You can specify values for the undefined constants when compiling your project to override the default behavior.

NOTE

When `_MYTYPE` is set to "Custom", the project contains the `My` namespace, but it contains no objects. However, setting `_MYTYPE` to "Empty" prevents the compiler from adding the `My` namespace and its objects.

This table describes the effects of the predefined values of the `_MY` compilation constants.

CONSTANT	MEANING
`_MYAPPLICATIONTYPE`	Enables `My.Application`, if the constant is "Console," Windows," or "WindowsForms": - The "Console" version derives from ConsoleApplicationBase. and has fewer members than the "Windows" version. - The "Windows" version derives from ApplicationBase.and has fewer members than the "WindowsForms" version. - The "WindowsForms" version of `My.Application` derives from WindowsFormsApplicationBase. If the `TARGET` constant is defined to be "winexe", then the class includes a `Sub Main` method.
`_MYCOMPUTERTYPE`	Enables `My.Computer`, if the constant is "Web" or "Windows": - The "Web" version derives from ServerComputer, and has fewer members than the "Windows" version. - The "Windows" version of `My.Computer` derives from Computer.
`_MYFORMS`	Enables `My.Forms`, if the constant is `TRUE`.

CONSTANT	MEANING
`_MYUSERTYPE`	Enables `My.User`, if the constant is "Web" or "Windows": - The "Web" version of `My.User` is associated with the user identity of the current HTTP request. - The "Windows" version of `My.User` is associated with the thread's current principal.
`_MYWEBSERVICES`	Enables `My.WebServices`, if the constant is `TRUE`.
`_MYTYPE`	Enables `My.Log`, `My.Request`, and `My.Response`, if the constant is "Web".

See also

- ApplicationBase
- Computer
- Log
- User
- How My Depends on Project Type
- Conditional Compilation
- /define (Visual Basic)
- My.Forms Object
- My.Request Object
- My.Response Object
- My.WebServices Object

Programming Concepts (Visual Basic)

3/8/2019 • 2 minutes to read • Edit Online

This section explains programming concepts in the Visual Basic language.

In this section

TITLE	DESCRIPTION
Asynchronous Programming with Async and Await (Visual Basic)	Describes how to write asynchronous solutions by using Async and Await keywords. Includes a walkthrough.
Attributes overview (Visual Basic)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (Visual Basic)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (Visual Basic)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (Visual Basic)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (Visual Basic)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (Visual Basic)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (Visual Basic)	Discusses the powerful query capabilities in the language syntax of Visual Basic, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (Visual Basic)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
Reflection (Visual Basic)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (Visual Basic)	Describes key concepts in binary, XML, and SOAP serialization.

Related sections

Assemblies in .NET	Describes how to create and use assemblies.
Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

How to: Create Unsigned Friend Assemblies (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

This example shows how to use friend assemblies with assemblies that are unsigned.

To create an assembly and a friend assembly

1. Open a command prompt.
2. Create a Visual Basic file named `friend_signed_A.` that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare friend_signed_B as a friend assembly.

```
' friend_unsigned_A.vb
' Compile with:
' vbc -target:library friend_unsigned_A.vb
Imports System.Runtime.CompilerServices
Imports System

<Assembly: InternalsVisibleTo("friend_unsigned_B")>

' Friend type.
Friend Class Class1
    Public Sub Test()
        Console.WriteLine("Class1.Test")
    End Sub
End Class

' Public type with Friend member.
Public Class Class2
    Friend Sub Test()
        Console.WriteLine("Class2.Test")
    End Sub
End Class
```

3. Compile and sign friend_signed_A by using the following command.

```
vbc -target:library friend_unsigned_A.vb
```

4. Create a Visual Basic file named `friend_unsigned_B` that contains the following code. Because friend_unsigned_A specifies friend_unsigned_B as a friend assembly, the code in friend_unsigned_B can access `Friend` types and members from friend_unsigned_A.

```
' friend_unsigned_B.vb
' Compile with:
' vbc -r:friend_unsigned_A.dll friend_unsigned_B.vb
Module Module1
    Sub Main()
        ' Access a Friend type.
        Dim inst1 As New Class1()
        inst1.Test()

        Dim inst2 As New Class2()
        ' Access a Friend member of a public type.
        inst2.Test()

        System.Console.ReadLine()
    End Sub
End Module
```

5. Compile friend_signed_B by using the following command.

```
vbc -r:friend_unsigned_A.dll friend_unsigned_B.vb
```

   The name of the assembly that is generated by the compiler must match the friend assembly name that is passed to the InternalsVisibleToAttribute attribute. You can explicitly set the assembly by using the `/out` compiler option.

6. Run the friend_signed_B.exe file.

   The program displays two strings: "Class1.Test" and "Class2.Test".

.NET Framework Security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `Friend` types and members.

See also

- InternalsVisibleToAttribute
- Assemblies in .NET
- Friend Assemblies
- How to: Create Signed Friend Assemblies (Visual Basic)
- Programming Guide Concepts

How to: Create Signed Friend Assemblies (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

This example shows how to use friend assemblies with assemblies that have strong names. Both assemblies must be strong named. Although both assemblies in this example use the same keys, you could use different keys for two assemblies.

To create a signed assembly and a friend assembly

1. Open a command prompt.
2. Use the following sequence of commands with the Strong Name tool to generate a keyfile and to display its public key. For more information, see Sn.exe (Strong Name Tool)).

   a. Generate a strong-name key for this example and store it in the file FriendAssemblies.snk:

   ```
   sn -k FriendAssemblies.snk
   ```

   b. Extract the public key from FriendAssemblies.snk and put it into FriendAssemblies.publickey:

   ```
   sn -p FriendAssemblies.snk FriendAssemblies.publickey
   ```

   c. Display the public key stored in the file FriendAssemblies.publickey:

   ```
   sn -tp FriendAssemblies.publickey
   ```

3. Create a Visual Basic file named `friend_signed_A` that contains the following code. The code uses the InternalsVisibleToAttribute attribute to declare friend_signed_B as a friend assembly.

   The Strong Name tool generates a new public key every time it runs. Therefore, you must replace the public key in the following code with the public key you just generated, as shown in the following example.

   ```
   ' friend_signed_A.vb
   ' Compile with:
   ' Vbc -target:library -keyfile:FriendAssemblies.snk friend_signed_A.vb
   Imports System.Runtime.CompilerServices

   <Assembly: InternalsVisibleTo("friend_signed_B,
   PublicKey=0024000004800000940000000602000000240000525341310004000001000100e3aedce99b7e10823920206f8e46cd
   5558b4ec7345bd1a5b201ffe71660625dcb8f9a08687d881c8f65a0dcf042f81475d2e88f3e3e273c8311ee40f952db306c02fbf
   c5d8bc6ee1e924e6ec8fe8c01932e0648a0d3e5695134af3bb7fab370d3012d083fa6b83179dd3d031053f72fc1f7da8459140b0
   af5afc4d2804deccb6")>
   Public Class Class1
       Public Sub Test()
           System.Console.WriteLine("Class1.Test")
           System.Console.ReadLine()
       End Sub
   End Class
   ```

4. Compile and sign friend_signed_A by using the following command.

   ```
   Vbc -target:library -keyfile:FriendAssemblies.snk friend_signed_A.vb
   ```

5. Create a Visual Basic file that is named `friend_signed_B` and contains the following code. Because

friend_signed_A specifies friend_signed_B as a friend assembly, the code in friend_signed_B can access `Friend` types and members from friend_signed_A. The file contains the following code.

```
' friend_signed_B.vb
' Compile with:
' Vbc -keyfile:FriendAssemblies.snk -r:friend_signed_A.dll friend_signed_B.vb
Module Sample
    Public Sub Main()
        Dim inst As New Class1
        inst.Test()
    End Sub
End Module
```

6. Compile and sign friend_signed_B by using the following command.

```
vbc -keyfile:FriendAssemblies.snk -r:friend_signed_A.dll friend_signed_B.vb
```

   The name of the assembly generated by the compiler must match the friend assembly name passed to the InternalsVisibleToAttribute attribute. You can explicitly set the assembly by using the `-out` compiler option. For more information, see -out (Visual Basic).

7. Run the friend_signed_B.exe file.

   The program displays the string "Class1.Test".

.NET Framework Security

There are similarities between the InternalsVisibleToAttribute attribute and the StrongNameIdentityPermission class. The main difference is that StrongNameIdentityPermission can demand security permissions to run a particular section of code, whereas the InternalsVisibleToAttribute attribute controls the visibility of `Friend` types and members.

See also

- InternalsVisibleToAttribute
- Assemblies in .NET
- Friend Assemblies
- How to: Create Unsigned Friend Assemblies (Visual Basic)
- -keyfile
- Sn.exe (Strong Name Tool))
- Creating and Using Strong-Named Assemblies
- Programming Concepts

How to: Create and Use Assemblies Using the Command Line (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

An assembly, or a dynamic linking library (DLL), is linked to your program at run time. To demonstrate building and using a DLL, consider the following scenario:

- `MathLibrary.DLL`: The library file that contains the methods to be called at run time. In this example, the DLL contains two methods, `Add` and `Multiply`.
- `Add`: The source file that contains the method `Add`. It returns the sum of its parameters. The class `AddClass` that contains the method `Add` is a member of the namespace `UtilityMethods`.
- `Mult`: The source code that contains the method `Multiply`. It returns the product of its parameters. The class `MultiplyClass` that contains the method `Multiply` is also a member of the namespace `UtilityMethods`.
- `TestCode`: The file that contains the `Main` method. It uses the methods in the DLL file to calculate the sum and the product of the run-time arguments.

Example

```
' File: Add.vb
Namespace UtilityMethods
    Public Class AddClass
        Public Shared Function Add(ByVal i As Long, ByVal j As Long) As Long
            Return i + j
        End Function
    End Class
End Namespace
```

```
' File: Mult.vb
Namespace UtilityMethods
    Public Class MultiplyClass
        Public Shared Function Multiply(ByVal x As Long, ByVal y As Long) As Long
            Return x * y
        End Function
    End Class
End Namespace
```

```
' File: TestCode.vb

Imports UtilityMethods

Module Test

    Sub Main(ByVal args As String())

        System.Console.WriteLine("Calling methods from MathLibrary.DLL:")

        If args.Length <> 2 Then
            System.Console.WriteLine("Usage: TestCode <num1> <num2>")
            Return
        End If

        Dim num1 As Long = Long.Parse(args(0))
        Dim num2 As Long = Long.Parse(args(1))

        Dim sum As Long = AddClass.Add(num1, num2)
        Dim product As Long = MultiplyClass.Multiply(num1, num2)

        System.Console.WriteLine("{0} + {1} = {2}", num1, num2, sum)
        System.Console.WriteLine("{0} * {1} = {2}", num1, num2, product)

    End Sub

End Module

' Output (assuming 1234 and 5678 are entered as command-line arguments):
' Calling methods from MathLibrary.DLL:
' 1234 + 5678 = 6912
' 1234 * 5678 = 7006652
```

This file contains the algorithm that uses the DLL methods, `Add` and `Multiply`. It starts with parsing the arguments entered from the command line, `num1` and `num2`. Then it calculates the sum by using the `Add` method on the `AddClass` class, and the product by using the `Multiply` method on the `MultiplyClass` class.

Notice that the `Imports` statement at the beginning of the file enables you to use the unqualified class names to reference the DLL methods at compile time, as follows:

```
MultiplyClass.Multiply(num1, num2)
```

Otherwise, you have to use the fully qualified names, as follows:

```
UtilityMethods.MultiplyClass.Multiply(num1, num2)
```

Execution

To run the program, enter the name of the EXE file, followed by two numbers, as follows:

```
TestCode 1234 5678
```

See also

- Programming Concepts
- Assemblies in .NET
- Creating a Class to Hold DLL Functions

How to: Determine If a File Is an Assembly (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

A file is an assembly if and only if it is managed, and contains an assembly entry in its metadata. For more information on assemblies and metadata, see the topic Assembly Manifest.

How to manually determine if a file is an assembly

1. Start the Ildasm.exe (IL Disassembler).
2. Load the file you wish to test.
3. If **ILDASM** reports that the file is not a portable executable (PE) file, then it is not an assembly. For more information, see the topic How to: View Assembly Contents.

How to programmatically determine if a file is an assembly

1. Call the GetAssemblyName method, passing the full file path and name of the file you are testing.
2. If a BadImageFormatException exception is thrown, the file is not an assembly.

Example

This example tests a DLL to see if it is an assembly.

```
Module Module1
    Sub Main()
        Try
            Dim testAssembly As Reflection.AssemblyName =

Reflection.AssemblyName.GetAssemblyName("C:\Windows\Microsoft.NET\Framework\v3.5\System.Net.dll")
            Console.WriteLine("Yes, the file is an Assembly.")
        Catch ex As System.IO.FileNotFoundException
            Console.WriteLine("The file cannot be found.")
        Catch ex As System.BadImageFormatException
            Console.WriteLine("The file is not an Assembly.")
        Catch ex As System.IO.FileLoadException
            Console.WriteLine("The Assembly has already been loaded.")
        End Try
        Console.ReadLine()
    End Sub
End Module
' Output (with .NET Framework 3.5 installed):
'        Yes, the file is an Assembly.
```

The GetAssemblyName method loads the test file, and then releases it once the information is read.

See also

- AssemblyName
- Programming Concepts
- Assemblies and the Global Assembly Cache (Visual Basic)

How to: Load and Unload Assemblies (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

The assemblies referenced by your program will automatically be loaded at build time, but it is also possible to load specific assemblies into the current application domain at runtime. For more information, see How to: Load Assemblies into an Application Domain.

There is no way to unload an individual assembly without unloading all of the application domains that contain it. Even if the assembly goes out of scope, the actual assembly file will remain loaded until all application domains that contain it are unloaded.

If you want to unload some assemblies but not others, consider creating a new application domain, executing the code inside that domain, and then unloading that application domain. For more information, see How to: Unload an Application Domain.

To load an assembly into an application domain

1. Use one of the several load methods contained in the classes AppDomain and System.Reflection. For more information, see How to: Load Assemblies into an Application Domain.

To unload an application domain

1. There is no way to unload an individual assembly without unloading all of the application domains that contain it. Use the `Unload` method from AppDomain to unload the application domains. For more information, see How to: Unload an Application Domain.

See also

- Programming Concepts
- Assemblies in .NET
- How to: Load Assemblies into an Application Domain

How to: Share an Assembly with Other Applications (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

Assemblies can be private or shared: by default, most simple programs consist of a private assembly because they are not intended to be used by other applications.

In order to share an assembly with other applications, it must be placed in the Global Assembly Cache (GAC).

Sharing an assembly

1. Create your assembly. For more information, see Creating Assemblies.
2. Assign a strong name to your assembly. For more information, see How to: Sign an Assembly with a Strong Name.
3. Assign version information to your assembly. For more information, see Assembly Versioning.
4. Add your assembly to the Global Assembly Cache. For more information, see How to: Install an Assembly into the Global Assembly Cache.
5. Access the types contained in the assembly from the other applications. For more information, see How to: Reference a Strong-Named Assembly.

See also

- Programming Concepts
- Assemblies in .NET
- Programming with Assemblies

Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (Visual Basic)

7/10/2019 • 8 minutes to read • Edit Online

If you embed type information from a strong-named managed assembly, you can loosely couple types in an application to achieve version independence. That is, your program can be written to use types from multiple versions of a managed library without having to be recompiled for each version.

Type embedding is frequently used with COM interop, such as an application that uses automation objects from Microsoft Office. Embedding type information enables the same build of a program to work with different versions of Microsoft Office on different computers. However, you can also use type embedding with a fully managed solution.

Type information can be embedded from an assembly that has the following characteristics:

- The assembly exposes at least one public interface.
- The embedded interfaces are annotated with a `ComImport` attribute and a `Guid` attribute (and a unique GUID).
- The assembly is annotated with the `ImportedFromTypeLib` attribute or the `PrimaryInteropAssembly` attribute, and an assembly-level `Guid` attribute. (By default, Visual Basic project templates include an assembly-level `Guid` attribute.)

After you have specified the public interfaces that can be embedded, you can create runtime classes that implement those interfaces. A client program can then embed the type information for those interfaces at design time by referencing the assembly that contains the public interfaces and setting the `Embed Interop Types` property of the reference to `True`. This is equivalent to using the command line compiler and referencing the assembly by using the `/link` compiler option. The client program can then load instances of your runtime objects typed as those interfaces. If you create a new version of your strong-named runtime assembly, the client program does not have to be recompiled with the updated runtime assembly. Instead, the client program continues to use whichever version of the runtime assembly is available to it, using the embedded type information for the public interfaces.

Because the primary function of type embedding is to support embedding of type information from COM interop assemblies, the following limitations apply when you embed type information in a fully managed solution:

- Only attributes specific to COM interop are embedded; other attributes are ignored.
- If a type uses generic parameters and the type of the generic parameter is an embedded type, that type cannot be used across an assembly boundary. Examples of crossing an assembly boundary include calling a method from another assembly or a deriving a type from a type defined in another assembly.
- Constants are not embedded.
- The System.Collections.Generic.Dictionary<TKey,TValue> class does not support an embedded type as a key. You can implement your own dictionary type to support an embedded type as a key.

In this walkthrough, you will do the following:

- Create a strong-named assembly that has a public interface that contains type information that can be embedded.
- Create a strong-named runtime assembly that implements that public interface.

- Create a client program that embeds the type information from the public interface and creates an instance of the class from the runtime assembly.
- Modify and rebuild the runtime assembly.
- Run the client program to see that the new version of the runtime assembly is being used without having to recompile the client program.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Creating an Interface

To create the type equivalence interface project

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Class Library** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceInterface`, and then click **OK**. The new project is created.
3. In **Solution Explorer**, right-click the Class1.vb file and click **Rename**. Rename the file to `ISampleInterface.vb` and press ENTER. Renaming the file will also rename the class to `ISampleInterface`. This class will represent the public interface for the class.
4. Right-click the TypeEquivalenceInterface project and click **Properties**. Click the **Compile** tab. Set the output path to a valid location on your development computer, such as `C:\TypeEquivalenceSample`. This location will also be used in a later step in this walkthrough.
5. While still editing the project properties, click the **Signing** tab. Select the **Sign the assembly** option. In the **Choose a strong name key file** list, click **<New...>**. In the **Key file name** box, type `key.snk`. Clear the **Protect my key file with a password** check box. Click **OK**.
6. Open the ISampleInterface.vb file. Add the following code to the ISampleInterface class file to create the ISampleInterface interface.

```
Imports System.Runtime.InteropServices

<ComImport()>
<Guid("8DA56996-A151-4136-B474-32784559F6DF")>
Public Interface ISampleInterface
    Sub GetUserInput()
    ReadOnly Property UserInput As String
End Interface
```

7. On the **Tools** menu, click **Create Guid**. In the **Create GUID** dialog box, click **Registry Format** and then click **Copy**. Click **Exit**.
8. In the `Guid` attribute, delete the sample GUID and paste in the GUID that you copied from the **Create GUID** dialog box. Remove the braces ({}) from the copied GUID.
9. On the **Project** menu, click **Show All Files**.
10. In **Solution Explorer**, expand the **My Project** folder. Double-click the AssemblyInfo.vb. Add the following attribute to the file.

```
<Assembly: ImportedFromTypeLib("")>
```

Save the file.

11. Save the project.
12. Right-click the TypeEquivalenceInterface project and click **Build**. The class library .dll file is compiled and saved to the specified build output path (for example, C:\TypeEquivalenceSample).

Creating a Runtime Class

To create the type equivalence runtime project

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Class Library** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceRuntime`, and then click **OK**. The new project is created.
3. In **Solution Explorer**, right-click the Class1.vb file and click **Rename**. Rename the file to `SampleClass.vb` and press ENTER. Renaming the file also renames the class to `SampleClass`. This class will implement the `ISampleInterface` interface.
4. Right-click the TypeEquivalenceRuntime project and click **Properties**. Click the **Compile** tab. Set the output path to the same location you used in the TypeEquivalenceInterface project, for example, `C:\TypeEquivalenceSample`.
5. While still editing the project properties, click the **Signing** tab. Select the **Sign the assembly** option. In the **Choose a strong name key file** list, click **<New...>**. In the **Key file name** box, type `key.snk`. Clear the **Protect my key file with a password** check box. Click **OK**.
6. Right-click the TypeEquivalenceRuntime project and click **Add Reference**. Click the **Browse** tab and browse to the output path folder. Select the TypeEquivalenceInterface.dll file and click **OK**.
7. On the **Project** menu, click **Show All Files**.
8. In **Solution Explorer**, expand the **References** folder. Select the TypeEquivalenceInterface reference. In the Properties window for the TypeEquivalenceInterface reference, set the **Specific Version** property to **False**.
9. Add the following code to the SampleClass class file to create the SampleClass class.

```
Imports TypeEquivalenceInterface

Public Class SampleClass
    Implements ISampleInterface

    Private p_UserInput As String
    Public ReadOnly Property UserInput() As String Implements ISampleInterface.UserInput
        Get
            Return p_UserInput
        End Get
    End Property

    Public Sub GetUserInput() Implements ISampleInterface.GetUserInput
        Console.WriteLine("Please enter a value:")
        p_UserInput = Console.ReadLine()
    End Sub
End Class
```

10. Save the project.

11. Right-click the TypeEquivalenceRuntime project and click **Build**. The class library .dll file is compiled and saved to the specified build output path (for example, C:\TypeEquivalenceSample).

Creating a Client Project

To create the type equivalence client project

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `TypeEquivalenceClient`, and then click **OK**. The new project is created.
3. Right-click the TypeEquivalenceClient project and click **Properties**. Click the **Compile** tab. Set the output path to the same location you used in the TypeEquivalenceInterface project, for example, `C:\TypeEquivalenceSample`.
4. Right-click the TypeEquivalenceClient project and click **Add Reference**. Click the **Browse** tab and browse to the output path folder. Select the TypeEquivalenceInterface.dll file (not the TypeEquivalenceRuntime.dll) and click **OK**.
5. On the **Project** menu, click **Show All Files**.
6. In **Solution Explorer**, expand the **References** folder. Select the TypeEquivalenceInterface reference. In the Properties window for the TypeEquivalenceInterface reference, set the **Embed Interop Types** property to **True**.
7. Add the following code to the Module1.vb file to create the client program.

```
Imports TypeEquivalenceInterface
Imports System.Reflection

Module Module1

    Sub Main()
        Dim sampleAssembly = Assembly.Load("TypeEquivalenceRuntime")
        Dim sampleClass As ISampleInterface = CType( _
            sampleAssembly.CreateInstance("TypeEquivalenceRuntime.SampleClass"), ISampleInterface)
        sampleClass.GetUserInput()
        Console.WriteLine(sampleClass.UserInput)
        Console.WriteLine(sampleAssembly.GetName().Version)
        Console.ReadLine()
    End Sub

End Module
```

8. Press CTRL+F5 to build and run the program.

Modifying the Interface

To modify the interface

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, right-click the TypeEquivalenceInterface project, and then click **Properties**. Click the **Application** tab. Click the **Assembly Information** button. Change the **Assembly Version** and **File Version** values to `2.0.0.0`.
3. Open the ISampleInterface.vb file. Add the following line of code to the ISampleInterface interface.

```
Function GetDate() As Date
```

Save the file.

4. Save the project.
5. Right-click the TypeEquivalenceInterface project and click **Build**. A new version of the class library .dll file is compiled and saved in the specified build output path (for example, C:\TypeEquivalenceSample).

Modifying the Runtime Class

To modify the runtime class

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, right-click the TypeEquivalenceRuntime project and click **Properties**. Click the **Application** tab. Click the **Assembly Information** button. Change the **Assembly Version** and **File Version** values to `2.0.0.0`.
3. Open the SampleClass.vb file. Add the following lines of code to the SampleClass class.

```
Public Function GetDate() As DateTime Implements ISampleInterface.GetDate
    Return Now
End Function
```

Save the file.

4. Save the project.
5. Right-click the TypeEquivalenceRuntime project and click **Build**. An updated version of the class library .dll file is compiled and saved in the previously specified build output path (for example, C:\TypeEquivalenceSample).
6. In File Explorer, open the output path folder (for example, C:\TypeEquivalenceSample). Double-click the TypeEquivalenceClient.exe to run the program. The program will reflect the new version of the TypeEquivalenceRuntime assembly without having been recompiled.

See also

- /link (Visual Basic)
- Programming Concepts
- Programming with Assemblies
- Assemblies in .NET

Asynchronous Programming with Async and Await (Visual Basic)

7/2/2019 • 16 minutes to read • Edit Online

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

Visual Studio 2012 introduced a simplified approach, async programming, that leverages asynchronous support in the .NET Framework 4.5 and higher as well as in the Windows Runtime. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

This topic provides an overview of when and how to use async programming and includes links to support topics that contain details and examples.

Async Improves Responsiveness

Asynchrony is essential for activities that are potentially blocking, such as when your application accesses the web. Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

The following table shows typical areas where asynchronous programming improves responsiveness. The listed APIs from the .NET Framework 4.5 and the Windows Runtime contain methods that support async programming.

APPLICATION AREA	SUPPORTING APIS THAT CONTAIN ASYNC METHODS
Web access	HttpClient, SyndicationClient
Working with files	StorageFile, StreamWriter, StreamReader, XmlReader
Working with images	MediaCapture, BitmapEncoder, BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it has failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

Async Methods Are Easier to Write

The Async and Await keywords in Visual Basic are the heart of async programming. By using those two keywords, you can use resources in the .NET Framework or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using `Async` and `Await` are referred to as async methods.

The following example shows an async method. Almost everything in the code should look completely familiar to you. The comments call out the features that you add to create the asynchrony.

You can find a complete Windows Presentation Foundation (WPF) example file at the end of this topic, and you can download the sample from Async Sample: Example from "Asynchronous Programming with Async and Await".

```
' Three things to note in the signature:
'  - The method has an Async modifier.
'  - The return type is Task or Task(Of T). (See "Return Types" section.)
'    Here, it is Task(Of Integer) because the return statement returns an integer.
'  - The method name ends in "Async."
Async Function AccessTheWebAsync() As Task(Of Integer)

    ' You need to add a reference to System.Net.Http to declare client.
    Dim client As HttpClient = New HttpClient()

    ' GetStringAsync returns a Task(Of String). That means that when you await the
    ' task you'll get a string (urlContents).
    Dim getStringTask As Task(Of String) = client.GetStringAsync("https://msdn.microsoft.com")

    ' You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork()

    ' The Await operator suspends AccessTheWebAsync.
    '  - AccessTheWebAsync can't continue until getStringTask is complete.
    '  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    '  - Control resumes here when getStringTask is complete.
    '  - The Await operator then retrieves the string result from getStringTask.
    Dim urlContents As String = Await getStringTask

    ' The return statement specifies an integer result.
    ' Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    Return urlContents.Length
End Function
```

If `AccessTheWebAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

```
Dim urlContents As String = Await client.GetStringAsync()
```

The following characteristics summarize what makes the previous example an async method.

- The method signature includes an `Async` modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - Task<TResult> if your method has a return statement in which the operand has type TResult.
  - Task if your method has no return statement or has a return statement with no operand.
  - Sub if you're writing an async event handler.

  For more information, see "Return Types and Parameters" later in this topic.

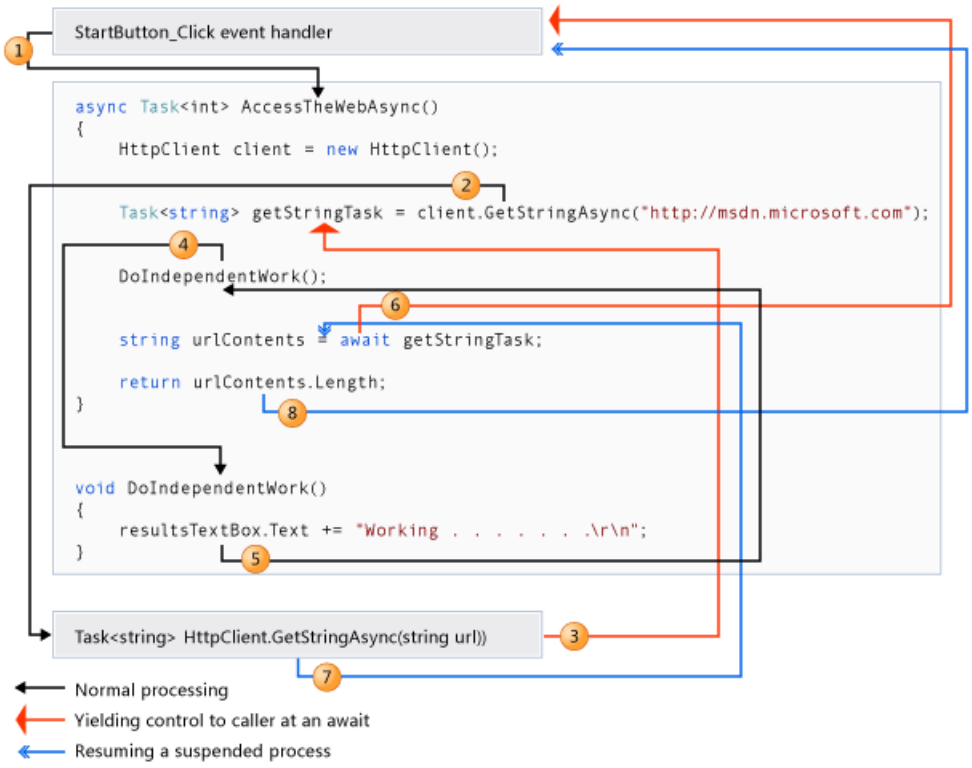
- The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.

In async methods, you use the provided keywords and types to indicate what you want to do, and the compiler does the rest, including keeping track of what must happen when control returns to an await point in a suspended method. Some routine processes, such as loops and exception handling, can be difficult to handle in traditional asynchronous code. In an async method, you write these elements much as you would in a synchronous solution, and the problem is solved.

For more information about asynchrony in previous versions of the .NET Framework, see TPL and Traditional .NET Framework Asynchronous Programming.

What Happens in an Async Method

The most important thing to understand in asynchronous programming is how the control flow moves from method to method. The following diagram leads you through the process.



The numbers in the diagram correspond to the following steps.

1. An event handler calls and awaits the `AccessTheWebAsync` async method.

2. `AccessTheWebAsync` creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.

3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `AccessTheWebAsync`.

   `GetStringAsync` returns a Task<TResult> where TResult is a string, and `AccessTheWebAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.

4. Because `getStringTask` hasn't been awaited yet, `AccessTheWebAsync` can continue with other work that doesn't depend on the final result from `GetStringAsync`. That work is represented by a call to the synchronous method `DoIndependentWork`.

5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.

6. `AccessTheWebAsync` has run out of work that it can do without a result from `getStringTask`. `AccessTheWebAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

   Therefore, `AccessTheWebAsync` uses an await operator to suspend its progress and to yield control to the method that called `AccessTheWebAsync`. `AccessTheWebAsync` returns a `Task<int>` (`Task(Of Integer)` in Visual Basic) to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

   > **NOTE**
   > If `GetStringAsync` (and therefore `getStringTask`) is complete before `AccessTheWebAsync` awaits it, control remains in `AccessTheWebAsync`. The expense of suspending and then returning to `AccessTheWebAsync` would be wasted if the called asynchronous process (`getStringTask`) has already completed and AccessTheWebSync doesn't have to wait for the final result.

   Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from `AccessTheWebAsync` before awaiting that result, or the caller might await immediately. The event handler is waiting for `AccessTheWebAsync`, and `AccessTheWebAsync` is waiting for `GetStringAsync`.

7. `GetStringAsync` completes and produces a string result. The string result isn't returned by the call to `GetStringAsync` in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, `getStringTask`. The await operator retrieves the result from `getStringTask`. The assignment statement assigns the retrieved result to `urlContents`.

8. When `AccessTheWebAsync` has the string result, the method can calculate the length of the string. Then the work of `AccessTheWebAsync` is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result.

If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

For more information about control flow, see Control Flow in Async Programs (Visual Basic).

API Async Methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. The .NET Framework 4.5 or higher contains many members that work with `Async` and `Await`. You can recognize these members by the "Async" suffix that's attached to the member name and a return type of Task or Task<TResult>. For example, the `System.IO.Stream` class contains methods such as CopyToAsync, ReadAsync, and WriteAsync alongside the synchronous methods CopyTo, Read, and Write.

The Windows Runtime also contains many methods that you can use with `Async` and `Await` in Windows apps. For more information and example methods, see Call asynchronous APIs in C# or Visual Basic, Asynchronous

programming (Windows Runtime apps), and WhenAny: Bridging between the .NET Framework and the Windows Runtime.

Threads

Async methods are intended to be non-blocking operations. An `Await` expression in an async method doesn't block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The `Async` and `Await` keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use Task.Run to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better than BackgroundWorker for I/O-bound operations because the code is simpler and you don't have to guard against race conditions. In combination with Task.Run, async programming is better than BackgroundWorker for CPU-bound operations because async programming separates the coordination details of running your code from the work that `Task.Run` transfers to the threadpool.

Async and Await

If you specify that a method is an async method by using an Async modifier, you enable the following two capabilities.

- The marked async method can use Await to designate suspension points. The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

  The suspension of an async method at an `Await` expression doesn't constitute an exit from the method, and `Finally` blocks don't run.

- The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an `Await` operator, but the absence of `Await` expressions doesn't cause a compiler error. If an async method doesn't use an `Await` operator to mark a suspension point, the method executes as a synchronous method does, despite the `Async` modifier. The compiler issues a warning for such methods.

`Async` and `Await` are contextual keywords. For more information and examples, see the following topics:

- Async
- Await Operator

Return Types and Parameters

In .NET Framework programming, an async method typically returns a Task or a Task<TResult>. Inside an async method, an `Await` operator is applied to a task that's returned from a call to another async method.

You specify Task<TResult> as the return type if the method contains a Return statement that specifies an operand of type `TResult`.

You use `Task` as the return type if the method has no return statement or has a return statement that doesn't return an operand.

The following example shows how you declare and call a method that returns a Task<TResult> or a Task.

```
' Signature specifies Task(Of Integer)
Async Function TaskOfTResult_MethodAsync() As Task(Of Integer)

    Dim hours As Integer
    ' . . .
    ' Return statement specifies an integer result.
    Return hours
End Function

' Calls to TaskOfTResult_MethodAsync
Dim returnedTaskTResult As Task(Of Integer) = TaskOfTResult_MethodAsync()
Dim intResult As Integer = Await returnedTaskTResult
' or, in a single statement
Dim intResult As Integer = Await TaskOfTResult_MethodAsync()

' Signature specifies Task
Async Function Task_MethodAsync() As Task

    ' . . .
    ' The method has no return statement.
End Function

' Calls to Task_MethodAsync
Task returnedTask = Task_MethodAsync()
Await returnedTask
' or, in a single statement
Await Task_MethodAsync()
```

Each returned task represents ongoing work. A task encapsulates information about the state of the asynchronous process and, eventually, either the final result from the process or the exception that the process raises if it doesn't succeed.

An async method can also be a `Sub` method. This return type is used primarily to define event handlers, where a return type is required. Async event handlers often serve as the starting point for async programs.

An async method that's a `Sub` procedure can't be awaited, and the caller can't catch any exceptions that the method throws.

An async method can't declare ByRef parameters, but the method can call methods that have such parameters.

For more information and examples, see Async Return Types (Visual Basic). For more information about how to catch exceptions in async methods, see Try...Catch...Finally Statement.

Asynchronous APIs in Windows Runtime programming have one of the following return types, which are similar to tasks:

- IAsyncOperation<TResult>, which corresponds to Task<TResult>
- IAsyncAction, which corresponds to Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult, TProgress>

For more information and an example, see Call asynchronous APIs in C# or Visual Basic.

Naming Convention

By convention, you append "Async" to the names of methods that have an `Async` modifier.

You can ignore the convention where an event, base class, or interface contract suggests a different name. For

example, you shouldn't rename common event handlers, such as `Button1_Click`.

Related Topics and Samples (Visual Studio)

TITLE	DESCRIPTION	SAMPLE
Walkthrough: Accessing the Web by Using Async and Await (Visual Basic)	Shows how to convert a synchronous WPF solution to an asynchronous WPF solution. The application downloads a series of websites.	Async Sample: Accessing the Web Walkthrough
How to: Extend the Async Walkthrough by Using Task.WhenAll (Visual Basic)	Adds Task.WhenAll to the previous walkthrough. The use of `WhenAll` starts all the downloads at the same time.	
How to: Make Multiple Web Requests in Parallel by Using Async and Await (Visual Basic)	Demonstrates how to start several tasks at the same time.	Async Sample: Make Multiple Web Requests in Parallel
Async Return Types (Visual Basic)	Illustrates the types that async methods can return and explains when each type is appropriate.	
Control Flow in Async Programs (Visual Basic)	Traces in detail the flow of control through a succession of await expressions in an asynchronous program.	Async Sample: Control Flow in Async Programs
Fine-Tuning Your Async Application (Visual Basic)	Shows how to add the following functionality to your async solution: - Cancel an Async Task or a List of Tasks (Visual Basic) - Cancel Async Tasks after a Period of Time (Visual Basic) - Cancel Remaining Async Tasks after One Is Complete (Visual Basic) - Start Multiple Async Tasks and Process Them As They Complete (Visual Basic)	Async Sample: Fine Tuning Your Application
Handling Reentrancy in Async Apps (Visual Basic)	Shows how to handle cases in which an active asynchronous operation is restarted while it's running.	
WhenAny: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use WhenAny with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask and WhenAny)
Async Cancellation: Bridging between the .NET Framework and the Windows Runtime	Shows how to bridge between Task types in the .NET Framework and IAsyncOperations in the Windows Runtime so that you can use CancellationTokenSource with a Windows Runtime method.	Async Sample: Bridging between .NET and Windows Runtime (AsTask & Cancellation)

TITLE	DESCRIPTION	SAMPLE
Using Async for File Access (Visual Basic)	Lists and demonstrates the benefits of using async and await to access files.	
Task-based Asynchronous Pattern (TAP)	Describes a new pattern for asynchrony in the .NET Framework. The pattern is based on the Task and Task<TResult> types.	
Async Videos on Channel 9	Provides links to a variety of videos about async programming.	

Complete Example

The following code is the MainWindow.xaml.vb file from the Windows Presentation Foundation (WPF) application that this topic discusses. You can download the sample from Async Sample: Example from "Asynchronous Programming with Async and Await".

```
' Add an Imports statement and a reference for System.Net.Http
Imports System.Net.Http

Class MainWindow

    ' Mark the event handler with async so you can use Await in it.
    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

        ' Call and await separately.
        'Task<int> getLengthTask = AccessTheWebAsync();
        '' You can do independent work here.
        'int contentLength = await getLengthTask;

        Dim contentLength As Integer = Await AccessTheWebAsync()

        ResultsTextBox.Text &=
            String.Format(vbCrLf & "Length of the downloaded string: {0}." & vbCrLf, contentLength)
    End Sub

    ' Three things to note in the signature:
    '  - The method has an Async modifier.
    '  - The return type is Task or Task(Of T). (See "Return Types" section.)
    '    Here, it is Task(Of Integer) because the return statement returns an integer.
    '  - The method name ends in "Async."
    Async Function AccessTheWebAsync() As Task(Of Integer)

        ' You need to add a reference to System.Net.Http to declare client.
        Dim client As HttpClient = New HttpClient()

        ' GetStringAsync returns a Task(Of String). That means that when you await the
        ' task you'll get a string (urlContents).
        Dim getStringTask As Task(Of String) = client.GetStringAsync("https://msdn.microsoft.com")

        ' You can do work here that doesn't rely on the string from GetStringAsync.
        DoIndependentWork()

        ' The Await operator suspends AccessTheWebAsync.
        '  - AccessTheWebAsync can't continue until getStringTask is complete.
        '  - Meanwhile, control returns to the caller of AccessTheWebAsync.
        '  - Control resumes here when getStringTask is complete.
        '  - The Await operator then retrieves the string result from getStringTask.
        Dim urlContents As String = Await getStringTask

        ' The return statement specifies an integer result.
        ' Any methods that are awaiting AccessTheWebAsync retrieve the length value.
        Return urlContents.Length
    End Function

    Sub DoIndependentWork()
        ResultsTextBox.Text &= "Working . . . . . ." & vbCrLf
    End Sub
End Class

' Sample Output:

' Working . . . . . .

' Length of the downloaded string: 41763.
```

See also

- Await Operator
- Async

Attributes overview (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see Reflection (Visual Basic).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see, Creating Custom Attributes (Visual Basic).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see Accessing Attributes by Using Reflection (Visual Basic).

Using Attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In Visual Basic, an attribute is enclosed in angle brackets (< >). It must appear immediately before the element to which it is applied, on the same line.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

```
<System.Serializable()> Public Class SampleClass
    ' Objects of this type can be serialized.
End Class
```

A method with the attribute DllImportAttribute is declared like this:

```
Imports System.Runtime.InteropServices
```

```
<System.Runtime.InteropServices.DllImport("user32.dll")>
Sub SampleMethod()
End Sub
```

More than one attribute can be placed on a declaration:

```
Imports System.Runtime.InteropServices
```

```
Sub MethodA(<[In](), Out()> ByVal x As Double)
End Sub
Sub MethodB(<Out(), [In]()> ByVal x As Double)
End Sub
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

```
<Conditional("DEBUG"), Conditional("TEST1")>
Sub TraceMethod()
End Sub
```

> **NOTE**
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Framework.

Attribute Parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted; named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
<DllImport("user32.dll")>
<DllImport("user32.dll", SetLastError:=False, ExactSpelling:=False)>
<DllImport("user32.dll", ExactSpelling:=False, SetLastError:=False)>
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Refer to the individual attribute's documentation for information on default parameter values.

Attribute Targets

The *target* of an attribute is the entity to which the attribute applies. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
<target : attribute-list>
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
`assembly`	Entire assembly
`module`	Current assembly module (which is different from a Visual Basic Module)

The following example shows how to apply attributes to assemblies and modules. For more information, see

Common Attributes (Visual Basic).

```
Imports System.Reflection
<Assembly: AssemblyTitleAttribute("Production assembly 4"),
Module: CLSCompliant(True)>
```

Common Uses for Attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related Sections

For more information, see:

- Creating Custom Attributes (Visual Basic)
- Accessing Attributes by Using Reflection (Visual Basic)
- How to: Create a C/C++ Union by Using Attributes (Visual Basic)
- Common Attributes (Visual Basic)
- Caller Information (Visual Basic)

See also

- Visual Basic Programming Guide
- Reflection (Visual Basic)
- Attributes

Caller Information (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. This is the file path at compile time.	`String`
CallerLineNumberAttribute	Line number in the source file at which the method is called.	`Integer`
CallerMemberNameAttribute	Method or property name of the caller. See Member Names later in this topic.	`String`

Example

The following example shows how to use Caller Info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

```
Private Sub DoProcessing()
    TraceMessage("Something happened.")
End Sub

Public Sub TraceMessage(message As String,
        <System.Runtime.CompilerServices.CallerMemberName> Optional memberName As String = Nothing,
        <System.Runtime.CompilerServices.CallerFilePath> Optional sourcefilePath As String = Nothing,
        <System.Runtime.CompilerServices.CallerLineNumber()> Optional sourceLineNumber As Integer = 0)

    System.Diagnostics.Trace.WriteLine("message: " & message)
    System.Diagnostics.Trace.WriteLine("member name: " & memberName)
    System.Diagnostics.Trace.WriteLine("source file path: " & sourcefilePath)
    System.Diagnostics.Trace.WriteLine("source line number: " & sourceLineNumber)
End Sub

' Sample output:
'   message: Something happened.
'   member name: DoProcessing
'   source file path: C:\Users\username\Documents\Visual Studio
2012\Projects\CallerInfoVB\CallerInfoVB\Form1.vb
'   source line number: 15
```

Remarks

You must specify an explicit default value for each optional parameter. You can't apply Caller Info attributes to parameters that aren't specified as optional.

The Caller Info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member Names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the member to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- Attributes (Visual Basic)
- Common Attributes (Visual Basic)
- Optional Parameters
- Programming Concepts (Visual Basic)

Collections (Visual Basic)

4/26/2019 • 14 minutes to read • Edit Online

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly-typed objects. For information about arrays, see Arrays.

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

> **NOTE**
> For the examples in this topic, include Imports statements for the `System.Collections.Generic` and `System.Linq` namespaces.

Using a Simple Collection

The examples in this section use the generic List<T> class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a For Each...Next statement.

```
' Create a list of strings.
Dim salmons As New List(Of String)
salmons.Add("chinook")
salmons.Add("coho")
salmons.Add("pink")
salmons.Add("sockeye")

' Iterate through the list.
For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see Collection Initializers.

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink", "sockeye"}

For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'Output: chinook coho pink sockeye
```

You can use a For...Next statement instead of a `For Each` statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using `For...Next` instead of `For Each`.

```
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink", "sockeye"}

For index = 0 To salmons.Count - 1
    Console.Write(salmons(index) & " ")
Next
'Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

```
' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink", "sockeye"}

' Remove an element in the list by specifying
' the object.
salmons.Remove("coho")

For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a `For Each` statement, a For...Next statement that iterates in descending order is used. This is because the RemoveAt method causes elements after a removed element to have a lower index value.

```
Dim numbers As New List(Of Integer) From
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

' Remove odd numbers.
For index As Integer = numbers.Count - 1 To 0 Step -1
    If numbers(index) Mod 2 = 1 Then
        ' Remove the element by specifying
        ' the zero-based index in the list.
        numbers.RemoveAt(index)
    End If
Next

' Iterate through the list.
' A lambda expression is placed in the ForEach method
' of the List(T) object.
numbers.ForEach(
    Sub(number) Console.Write(number & " "))
' Output: 0 2 4 6 8
```

For the type of elements in the List<T>, you can also define your own class. In the following example, the `Galaxy` class that is used by the List<T> is defined in the code.

```
Private Sub IterateThroughList()
    Dim theGalaxies As New List(Of Galaxy) From
        {
            New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400},
            New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25},
            New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0},
            New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        }

    For Each theGalaxy In theGalaxies
        With theGalaxy
            Console.WriteLine(.Name & "  " & .MegaLightYears)
        End With
    Next

    ' Output:
    '  Tadpole  400
    '  Pinwheel  25
    '  Milky Way  0
    '  Andromeda  3
End Sub

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class
```

Kinds of Collections

Many common collections are provided by the .NET Framework. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- System.Collections.Generic classes
- System.Collections.Concurrent classes
- System.Collections classes

- Visual Basic `Collection` class

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the System.Collections.Generic namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the System.Collections.Generic namespace:

CLASS	DESCRIPTION
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see Commonly Used Collection Types, Selecting a Collection Class, and System.Collections.Generic.

System.Collections.Concurrent Classes

In the .NET Framework 4 or newer, the collections in the System.Collections.Concurrent namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the System.Collections.Concurrent namespace should be used instead of the corresponding types in the System.Collections.Generic and System.Collections namespaces whenever multiple threads are accessing the collection concurrently. For more information, see Thread-Safe Collections and System.Collections.Concurrent.

Some classes included in the System.Collections.Concurrent namespace are BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T>, and ConcurrentStack<T>.

System.Collections Classes

The classes in the System.Collections namespace do not store elements as specifically typed objects, but as objects of type `Object`.

Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the legacy types in the `System.Collections` namespace.

The following table lists some of the frequently used classes in the `System.Collections` namespace:

CLASS	DESCRIPTION
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.

CLASS	DESCRIPTION
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

The System.Collections.Specialized namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Visual Basic Collection Class

You can use the Visual Basic Collection class to access a collection item by using either a numeric index or a `String` key. You can add items to a collection object either with or without specifying a key. If you add an item without a key, you must use its numeric index to access it.

The Visual Basic `Collection` class stores all its elements as type `Object`, so you can add an item of any data type. There is no safeguard against inappropriate data types being added.

When you use the Visual Basic `Collection` class, the first item in a collection has an index of 1. This differs from the .NET Framework collection classes, for which the starting index is 0.

Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the Visual Basic `Collection` class.

For more information, see Collection.

Implementing a Collection of Key/Value Pairs

The Dictionary<TKey,TValue> generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the `Dictionary` class is implemented as a hash table.

The following example creates a `Dictionary` collection and iterates through the dictionary by using a `For Each` statement.

```
Private Sub IterateThroughDictionary()
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    For Each kvp As KeyValuePair(Of String, Element) In elements
        Dim theElement As Element = kvp.Value

        Console.WriteLine("key: " & kvp.Key)
        With theElement
            Console.WriteLine("values: " & .Symbol & " " &
                .Name & " " & .AtomicNumber)
        End With
    Next
End Sub

Private Function BuildDictionary() As Dictionary(Of String, Element)
    Dim elements As New Dictionary(Of String, Element)

    AddToDictionary(elements, "K", "Potassium", 19)
    AddToDictionary(elements, "Ca", "Calcium", 20)
    AddToDictionary(elements, "Sc", "Scandium", 21)
    AddToDictionary(elements, "Ti", "Titanium", 22)

    Return elements
End Function

Private Sub AddToDictionary(ByVal elements As Dictionary(Of String, Element),
ByVal symbol As String, ByVal name As String, ByVal atomicNumber As Integer)
    Dim theElement As New Element

    theElement.Symbol = symbol
    theElement.Name = name
    theElement.AtomicNumber = atomicNumber

    elements.Add(Key:=theElement.Symbol, value:=theElement)
End Sub

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class
```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```
Private Function BuildDictionary2() As Dictionary(Of String, Element)
    Return New Dictionary(Of String, Element) From
        {
            {"K", New Element With
                {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
            {"Ca", New Element With
                {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
            {"Sc", New Element With
                {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
            {"Ti", New Element With
                {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
        }
End Function
```

The following example uses the ContainsKey method and the Item[TKey] property of `Dictionary` to quickly find an item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements(symbol)` code in Visual Basic.

```
Private Sub FindInDictionary(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    If elements.ContainsKey(symbol) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Dim theElement = elements(symbol)
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

The following example instead uses the TryGetValue method quickly find an item by key.

```
Private Sub FindInDictionary2(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    Dim theElement As Element = Nothing
    If elements.TryGetValue(symbol, theElement) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see Getting Started with LINQ in Visual Basic.

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```
Private Sub ShowLINQ()
    Dim elements As List(Of Element) = BuildList()

    ' LINQ Query.
    Dim subset = From theElement In elements
                 Where theElement.AtomicNumber < 22
                 Order By theElement.Name

    For Each theElement In subset
        Console.WriteLine(theElement.Name & " " & theElement.AtomicNumber)
    Next

    ' Output:
    '  Calcium 20
    '  Potassium 19
    '  Scandium 21
End Sub

Private Function BuildList() As List(Of Element)
    Return New List(Of Element) From
        {
            {New Element With
                {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
            {New Element With
                {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
            {New Element With
                {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
            {New Element With
                {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
        }
End Function

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class
```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a List<T>. The `Car` class implements the IComparable<T> interface, which requires that the CompareTo method be implemented.

Each call to the CompareTo method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the Sort method of the List<T> causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```
Public Sub ListCars()

    ' Create some new cars.
    Dim cars As New List(Of Car) From
    {
        New Car With {.Name = "car1", .Color = "blue", .Speed = 20},
        New Car With {.Name = "car2", .Color = "red", .Speed = 50},
        New Car With {.Name = "car3", .Color = "green", .Speed = 10},
        New Car With {.Name = "car4", .Color = "blue", .Speed = 50},
```

```
        New Car With {.Name = "car5", .Color = "blue", .Speed = 30},
        New Car With {.Name = "car6", .Color = "red", .Speed = 60},
        New Car With {.Name = "car7", .Color = "green", .Speed = 50}
    }

    ' Sort the cars by color alphabetically, and then by speed
    ' in descending order.
    cars.Sort()

    ' View all of the cars.
    For Each thisCar As Car In cars
        Console.Write(thisCar.Color.PadRight(5) & " ")
        Console.Write(thisCar.Speed.ToString & " ")
        Console.Write(thisCar.Name)
        Console.WriteLine()
    Next

    ' Output:
    '  blue  50 car4
    '  blue  30 car5
    '  blue  20 car1
    '  green 50 car7
    '  green 10 car3
    '  red   60 car6
    '  red   50 car2
End Sub

Public Class Car
    Implements IComparable(Of Car)

    Public Property Name As String
    Public Property Speed As Integer
    Public Property Color As String

    Public Function CompareTo(ByVal other As Car) As Integer _
        Implements System.IComparable(Of Car).CompareTo
        ' A call to this method makes a single comparison that is
        ' used for sorting.

        ' Determine the relative order of the objects being compared.
        ' Sort by color alphabetically, and then by speed in
        ' descending order.

        ' Compare the colors.
        Dim compare As Integer
        compare = String.Compare(Me.Color, other.Color, True)

        ' If the colors are the same, compare the speeds.
        If compare = 0 Then
            compare = Me.Speed.CompareTo(other.Speed)

            ' Use descending order for speed.
            compare = -compare
        End If

        Return compare
    End Function
End Class
```

Defining a Custom Collection

You can define a collection by implementing the IEnumerable<T> or IEnumerable interface. For additional information, see Enumerating a Collection.

Although you can define a custom collection, it is usually better to instead use the collections that are included in the .NET Framework, which are described in Kinds of Collections earlier in this topic.

The following example defines a custom collection class named `AllColors`. This class implements the IEnumerable interface, which requires that the GetEnumerator method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the IEnumerator interface, which requires that the Current property, MoveNext method, and Reset method be implemented.

```
Public Sub ListColors()
    Dim colors As New AllColors()

    For Each theColor As Color In colors
        Console.Write(theColor.Name & " ")
    Next
    Console.WriteLine()
    ' Output: red blue green
End Sub

' Collection class.
Public Class AllColors
    Implements System.Collections.IEnumerable

    Private _colors() As Color =
    {
        New Color With {.Name = "red"},
        New Color With {.Name = "blue"},
        New Color With {.Name = "green"}
    }

    Public Function GetEnumerator() As System.Collections.IEnumerator _
        Implements System.Collections.IEnumerable.GetEnumerator

        Return New ColorEnumerator(_colors)

        ' Instead of creating a custom enumerator, you could
        ' use the GetEnumerator of the array.
        'Return _colors.GetEnumerator
    End Function

    ' Custom enumerator.
    Private Class ColorEnumerator
        Implements System.Collections.IEnumerator

        Private _colors() As Color
        Private _position As Integer = -1

        Public Sub New(ByVal colors() As Color)
            _colors = colors
        End Sub

        Public ReadOnly Property Current() As Object _
            Implements System.Collections.IEnumerator.Current
            Get
                Return _colors(_position)
            End Get
        End Property

        Public Function MoveNext() As Boolean _
            Implements System.Collections.IEnumerator.MoveNext
            _position += 1
            Return (_position < _colors.Length)
        End Function

        Public Sub Reset() Implements System.Collections.IEnumerator.Reset
            _position = -1
        End Sub
    End Class
End Class

' Element class.
Public Class Color
    Public Property Name As String
End Class
```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a Yield statement to return each element of the collection one at a time.

You call an iterator by using a For Each…Next statement. Each iteration of the `For Each` loop calls the iterator. When a `Yield` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see Iterators (Visual Basic).

The following example uses an iterator method. The iterator method has a `Yield` statement that is inside a For…Next loop. In the `ListEvenNumbers` method, each iteration of the `For Each` statement body creates a call to the iterator method, which proceeds to the next `Yield` statement.

```
Public Sub ListEvenNumbers()
    For Each number As Integer In EvenSequence(5, 18)
        Console.Write(number & " ")
    Next
    Console.WriteLine()
    ' Output: 6 8 10 12 14 16 18
End Sub

Private Iterator Function EvenSequence(
ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As IEnumerable(Of Integer)

' Yield even numbers in the range.
    For number = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

See also

- Collection Initializers
- Programming Concepts (Visual Basic)
- Option Strict Statement
- LINQ to Objects (Visual Basic)
- Parallel LINQ (PLINQ)
- Collections and Data Structures
- Selecting a Collection Class
- Comparisons and Sorts Within Collections
- When to Use Generic Collections

Covariance and Contravariance (Visual Basic)

10/4/2018 • 3 minutes to read • Edit Online

In Visual Basic, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
' Assignment compatibility.
Dim str As String = "test"
' An object of a more derived type is assigned to an object of a less derived type.
Dim obj As Object = str

' Covariance.
Dim strings As IEnumerable(Of String) = New List(Of String)()
' An object that is instantiated with a more derived type argument
' is assigned to an object instantiated with a less derived type argument.
' Assignment compatibility is preserved.
Dim objects As IEnumerable(Of Object) = strings

' Contravariance.
' Assume that there is the following method in the class:
' Shared Sub SetObject(ByVal o As Object)
' End Sub
Dim actObject As Action(Of Object) = AddressOf SetObject

' An object that is instantiated with a less derived type argument
' is assigned to an object instantiated with a more derived type argument.
' Assignment compatibility is reversed.
Dim actString As Action(Of String) = actObject
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
Dim array() As Object = New String(10) {}
' The following statement produces a run-time exception.
' array(0) = 10
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see Variance in Delegates (Visual Basic) and Using Variance in Delegates (Visual Basic).

The following code example shows covariance and contravariance support for method groups.

```
Shared Function GetObject() As Object
    Return Nothing
End Function

Shared Sub SetObject(ByVal obj As Object)
End Sub

Shared Function GetString() As String
    Return ""
End Function

Shared Sub SetString(ByVal str As String)

End Sub

Shared Sub Test()
    ' Covariance. A delegate specifies a return type as object,
    ' but you can assign a method that returns a string.
    Dim del As Func(Of Object) = AddressOf GetString

    ' Contravariance. A delegate specifies a parameter type as string,
    ' but you can assign a method that takes an object.
    Dim del2 As Action(Of String) = AddressOf SetObject
End Sub
```

In .NET Framework 4 or later, Visual Basic supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see Variance in Generic Interfaces (Visual Basic) and Variance in Delegates (Visual Basic).

The following code example shows implicit reference conversion for generic interfaces.

```
Dim strings As IEnumerable(Of String) = New List(Of String)
Dim objects As IEnumerable(Of Object) = strings
```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. Visual Basic enables you to create your own variant interfaces and delegates. For more information, see Creating Variant Generic Interfaces (Visual Basic) and Variance in Delegates (Visual Basic).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (Visual Basic)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
Creating Variant Generic Interfaces (Visual Basic)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (Visual Basic)	Shows how covariance and contravariance support in the IEnumerable<T> and IComparable<T> interfaces can help you reuse code.
Variance in Delegates (Visual Basic)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.

TITLE	DESCRIPTION
Using Variance in Delegates (Visual Basic)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (Visual Basic)	Shows how covariance and contravariance support in the `Func` and `Action` delegates can help you reuse code.

Expression Trees (Visual Basic)

4/2/2019 • 4 minutes to read • Edit Online

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to: Use Expression Trees to Build Dynamic Queries (Visual Basic).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The Visual Basic compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in Visual Basic, see Lambda Expressions.

The following code examples demonstrate how to have the Visual Basic compiler create an expression tree that represents the lambda expression `Function(num) num < 5`.

```
Dim lambda As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the Expression class. This class contains static factory methods that create expression tree nodes of specific types, for example, ParameterExpression, which represents a variable or parameter, or MethodCallExpression, which represents a method call. ParameterExpression, MethodCallExpression, and the other expression-specific types are also defined in the System.Linq.Expressions namespace. These types derive from the abstract type Expression.

The following code example demonstrates how to create an expression tree that represents the lambda expression `Function(num) num < 5` by using the API.

```
' Import the following namespace to your project: System.Linq.Expressions

' Manually build the expression tree for the lambda expression num => num < 5.
Dim numParam As ParameterExpression = Expression.Parameter(GetType(Integer), "num")
Dim five As ConstantExpression = Expression.Constant(5, GetType(Integer))
Dim numLessThanFive As BinaryExpression = Expression.LessThan(numParam, five)
Dim lambda1 As Expression(Of Func(Of Integer, Boolean)) =
  Expression.Lambda(Of Func(Of Integer, Boolean))(
        numLessThanFive,
        New ParameterExpression() {numParam})
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the Visual Basic compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
' Creating a parameter expression.
Dim value As ParameterExpression =
    Expression.Parameter(GetType(Integer), "value")

' Creating an expression to hold a local variable.
Dim result As ParameterExpression =
    Expression.Parameter(GetType(Integer), "result")

' Creating a label to jump to from a loop.
Dim label As LabelTarget = Expression.Label(GetType(Integer))

' Creating a method body.
Dim block As BlockExpression = Expression.Block(
    New ParameterExpression() {result},
    Expression.Assign(result, Expression.Constant(1)),
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(value, Expression.Constant(1)),
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            Expression.Break(label, result)
        ),
        label
    )
)

' Compile an expression tree and return a delegate.
Dim factorial As Integer =
    Expression.Lambda(Of Func(Of Integer, Integer))(block, value).Compile()(5)

Console.WriteLine(factorial)
' Prints 120.
```

For more information, see Generating Dynamic Methods with Expression Trees in Visual Studio 2010, which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression `Function(num) num < 5` can be decomposed into its parts.

```
' Import the following namespace to your project: System.Linq.Expressions

' Create an expression tree.
Dim exprTree As Expression(Of Func(Of Integer, Boolean)) = Function(num) num < 5

' Decompose the expression tree.
Dim param As ParameterExpression = exprTree.Parameters(0)
Dim operation As BinaryExpression = exprTree.Body
Dim left As ParameterExpression = operation.Left
Dim right As ConstantExpression = operation.Right

Console.WriteLine(String.Format("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value))

' This code produces the following output:
'
' Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see How to: Modify Expression Trees (Visual Basic).

Compiling Expression Trees

The Expression<TDelegate> type provides the Compile method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
' Creating an expression tree.
Dim expr As Expression(Of Func(Of Integer, Boolean)) =
    Function(num) num < 5

' Compiling the expression tree into a delegate.
Dim result As Func(Of Integer, Boolean) = expr.Compile()

' Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4))

' Prints True.

' You can also use simplified syntax
' to compile and run an expression tree.
' The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4))

' Also prints True.
```

For more information, see How to: Execute Expression Trees (Visual Basic).

See also

- System.Linq.Expressions
- How to: Execute Expression Trees (Visual Basic)
- How to: Modify Expression Trees (Visual Basic)
- Lambda Expressions

- Dynamic Language Runtime Overview
- Programming Concepts (Visual Basic)

Iterators (Visual Basic)

8/22/2019 • 10 minutes to read • Edit Online

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the Yield statement to return each element one at a time. When a `Yield` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a For Each…Next statement, or by using a LINQ query.

In the following example, the first iteration of the `For Each` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `Yield` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `Yield` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
Sub Main()
    For Each number As Integer In SomeNumbers()
        Console.Write(number & " ")
    Next
    ' Output: 3 5 8
    Console.ReadKey()
End Sub

Private Iterator Function SomeNumbers() As System.Collections.IEnumerable
    Yield 3
    Yield 5
    Yield 8
End Function
```

The return type of an iterator method or `get` accessor can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

You can use an `Exit Function` or `Return` statement to end the iteration.

A Visual Basic iterator function or `get` accessor declaration includes an Iterator modifier.

Iterators were introduced in Visual Basic in Visual Studio 2012.

In this topic

- Simple Iterator
- Creating a Collection Class
- Try Blocks
- Anonymous Methods
- Using Iterators with a Generic List
- Syntax Information
- Technical Implementation

- Use of Iterators

> **NOTE**
>
> For all examples in the topic except the Simple Iterator example, include Imports statements for the `System.Collections` and `System.Collections.Generic` namespaces.

Simple Iterator

The following example has a single `Yield` statement that is inside a For...Next loop. In `Main`, each iteration of the `For Each` statement body creates a call to the iterator function, which proceeds to the next `Yield` statement.

```
Sub Main()
    For Each number As Integer In EvenSequence(5, 18)
        Console.Write(number & " ")
    Next
    ' Output: 6 8 10 12 14 16 18
    Console.ReadKey()
End Sub

Private Iterator Function EvenSequence(
ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number As Integer = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the IEnumerable interface, which requires a GetEnumerator method. The compiler implicitly calls the `GetEnumerator` method, which returns an IEnumerator.

The `GetEnumerator` method returns each string one at a time by using the `Yield` statement, and an `Iterator` modifier is in the function declaration.

```
Sub Main()
    Dim days As New DaysOfTheWeek()
    For Each day As String In days
        Console.Write(day & " ")
    Next
    ' Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey()
End Sub

Private Class DaysOfTheWeek
    Implements IEnumerable

    Public days =
        New String() {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}

    Public Iterator Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        ' Yield each day of the week.
        For i As Integer = 0 To days.Length - 1
            Yield days(i)
        Next
    End Function
End Class
```

The following example creates a `Zoo` class that contains a collection of animals.

The `For Each` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `For Each` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.

```
Sub Main()
    Dim theZoo As New Zoo()

    theZoo.AddMammal("Whale")
    theZoo.AddMammal("Rhinoceros")
    theZoo.AddBird("Penguin")
    theZoo.AddBird("Warbler")

    For Each name As String In theZoo
        Console.Write(name & " ")
    Next
    Console.WriteLine()
    ' Output: Whale Rhinoceros Penguin Warbler

    For Each name As String In theZoo.Birds
        Console.Write(name & " ")
    Next
    Console.WriteLine()
    ' Output: Penguin Warbler

    For Each name As String In theZoo.Mammals
        Console.Write(name & " ")
    Next
    Console.WriteLine()
    ' Output: Whale Rhinoceros

    Console.ReadKey()
End Sub

Public Class Zoo
    Implements IEnumerable

    ' Private members.
    Private animals As New List(Of Animal)
```

```
    ' Public methods.
    Public Sub AddMammal(ByVal name As String)
        animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Mammal})
    End Sub

    Public Sub AddBird(ByVal name As String)
        animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Bird})
    End Sub

    Public Iterator Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        For Each theAnimal As Animal In animals
            Yield theAnimal.Name
        Next
    End Function

    ' Public members.
    Public ReadOnly Property Mammals As IEnumerable
        Get
            Return AnimalsForType(Animal.TypeEnum.Mammal)
        End Get
    End Property

    Public ReadOnly Property Birds As IEnumerable
        Get
            Return AnimalsForType(Animal.TypeEnum.Bird)
        End Get
    End Property

    ' Private methods.
    Private Iterator Function AnimalsForType( _
    ByVal type As Animal.TypeEnum) As IEnumerable
        For Each theAnimal As Animal In animals
            If (theAnimal.Type = type) Then
                Yield theAnimal.Name
            End If
        Next
    End Function

    ' Private class.
    Private Class Animal
        Public Enum TypeEnum
            Bird
            Mammal
        End Enum

        Public Property Name As String
        Public Property Type As TypeEnum
    End Class
End Class
```

Try Blocks

Visual Basic allows a `Yield` statement in the `Try` block of a Try...Catch...Finally Statement. A `Try` block that has a `Yield` statement can have `Catch` blocks, and can have a `Finally` block.

The following example includes `Try`, `Catch`, and `Finally` blocks in an iterator function. The `Finally` block in the iterator function executes before the `For Each` iteration finishes.

```
Sub Main()
    For Each number As Integer In Test()
        Console.WriteLine(number)
    Next
    Console.WriteLine("For Each is done.")

    ' Output:
    '  3
    '  4
    '  Something happened. Yields are done.
    '  Finally is called.
    '  For Each is done.
    Console.ReadKey()
End Sub

Private Iterator Function Test() As IEnumerable(Of Integer)
    Try
        Yield 3
        Yield 4
        Throw New Exception("Something happened. Yields are done.")
        Yield 5
        Yield 6
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        Console.WriteLine("Finally is called.")
    End Try
End Function
```

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (instead of the iterator method) throws an exception, a `Catch` block in the iterator function is not executed, but a `Finally` block in the iterator function is executed. A `Catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Anonymous Methods

In Visual Basic, an anonymous function can be an iterator function. The following example illustrates this.

```
Dim iterateSequence = Iterator Function() _
                      As IEnumerable(Of Integer)
                          Yield 1
                          Yield 2
                      End Function

For Each number As Integer In iterateSequence()
    Console.Write(number & " ")
Next
' Output: 1 2
Console.ReadKey()
```

The following example has a non-iterator method that validates the arguments. The method returns the result of an anonymous iterator that describes the collection elements.

```
Sub Main()
    For Each number As Integer In GetSequence(5, 10)
        Console.Write(number & " ")
    Next
    ' Output: 5 6 7 8 9 10
    Console.ReadKey()
End Sub

Public Function GetSequence(ByVal low As Integer, ByVal high As Integer) _
As IEnumerable
    ' Validate the arguments.
    If low < 1 Then
        Throw New ArgumentException("low is too low")
    End If
    If high > 140 Then
        Throw New ArgumentException("high is too high")
    End If

    ' Return an anonymous iterator function.
    Dim iterateSequence = Iterator Function() As IEnumerable
                              For index = low To high
                                  Yield index
                              Next
                          End Function
    Return iterateSequence()
End Function
```

If validation is instead inside the iterator function, the validation cannot be performed until the start of the first iteration of the `For Each` body.

Using Iterators with a Generic List

In the following example, the `Stack(Of T)` generic class implements the IEnumerable<T> generic interface. The `Push` method assigns values to an array of type `T`. The GetEnumerator method returns the array values by using the `Yield` statement.

In addition to the generic GetEnumerator method, the non-generic GetEnumerator method must also be implemented. This is because IEnumerable<T> inherits from IEnumerable. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property declaration includes the `Iterator` keyword.

```
Sub Main()
    Dim theStack As New Stack(Of Integer)

    ' Add items to the stack.
    For number As Integer = 0 To 9
        theStack.Push(number)
    Next

    ' Retrieve items from the stack.
    ' For Each is allowed because theStack implements
    ' IEnumerable(Of Integer).
    For Each number As Integer In theStack
        Console.Write("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 9 8 7 6 5 4 3 2 1 0

    ' For Each is allowed, because theStack.TopToBottom
```

```
    ' For Each is allowed, because theStack.TopToBottom
    ' returns IEnumerable(Of Integer).
    For Each number As Integer In theStack.TopToBottom
        Console.Write("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 9 8 7 6 5 4 3 2 1 0

    For Each number As Integer In theStack.BottomToTop
        Console.Write("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 0 1 2 3 4 5 6 7 8 9

    For Each number As Integer In theStack.TopN(7)
        Console.Write("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 9 8 7 6 5 4 3

    Console.ReadKey()
End Sub

Public Class Stack(Of T)
    Implements IEnumerable(Of T)

    Private values As T() = New T(99) {}
    Private top As Integer = 0

    Public Sub Push(ByVal t As T)
        values(top) = t
        top = top + 1
    End Sub

    Public Function Pop() As T
        top = top - 1
        Return values(top)
    End Function

    ' This function implements the GetEnumerator method. It allows
    ' an instance of the class to be used in a For Each statement.
    Public Iterator Function GetEnumerator() As IEnumerator(Of T) _
        Implements IEnumerable(Of T).GetEnumerator

        For index As Integer = top - 1 To 0 Step -1
            Yield values(index)
        Next
    End Function

    Public Iterator Function GetEnumerator1() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        Yield GetEnumerator()
    End Function

    Public ReadOnly Property TopToBottom() As IEnumerable(Of T)
        Get
            Return Me
        End Get
    End Property

    Public ReadOnly Iterator Property BottomToTop As IEnumerable(Of T)
        Get
            For index As Integer = 0 To top - 1
                Yield values(index)
            Next
        End Get
    End Property

    Public Iterator Function TopN(ByVal itemsFromTop As Integer)
```

```
    Public Iterator Function TopN(ByVal itemsFromTop As Integer) _
        As IEnumerable(Of T)

        ' Return less than itemsFromTop if necessary.
        Dim startIndex As Integer =
            If(itemsFromTop >= top, 0, top - itemsFromTop)

        For index As Integer = top - 1 To startIndex Step -1
            Yield values(index)
        Next
    End Function
End Class
```

Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static destructor.

An implicit conversion must exist from the expression type in the `Yield` statement to the return type of the iterator.

In Visual Basic, an iterator method cannot have any `ByRef` parameters.

In Visual Basic, "Yield" is not a reserved word and has special meaning only when it is used in an `Iterator` method or `get` accessor.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `For Each...Next` loop in the client code continues.

To see what the compiler does, you can use the Ildasm.exe tool to view the Microsoft intermediate language code that is generated for an iterator method.

When you create an iterator for a class or struct, you do not have to implement the whole IEnumerator interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the IEnumerator or IEnumerator<T> interface.

On each successive iteration of the `For Each…Next` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `Yield` statement. It then continues to the next `Yield` statement until the end of the iterator body is reached, or until an `Exit Function` or `Return` statement is encountered.

Iterators do not support the IEnumerator.Reset method. To re-iterate from the start, you must obtain a new iterator.

For additional information, see the Visual Basic Language Specification.

Use of Iterators

Iterators enable you to maintain the simplicity of a `For Each` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `For Each` loop iteration.
- Avoid fully loading a large list before the first iteration of a `For Each` loop. An example is a paged fetch to load a batch of table rows. Another example is the EnumerateFiles method, which implements iterators within the .NET Framework.

- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See also

- System.Collections.Generic
- IEnumerable<T>
- For Each...Next Statement
- Yield Statement
- Iterator

Language-Integrated Query (LINQ) (Visual Basic)

5/4/2018 • 2 minutes to read • Edit Online

LINQ is a set of features that extends powerful query capabilities to the language syntax of Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store. The .NET Framework includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

In This Section

Introduction to LINQ (Visual Basic)

Provides a general introduction to the kinds of applications that you can write and the kinds of problems that you can solve with LINQ queries.

Getting Started with LINQ in Visual Basic

Describes the basic facts you should know in order to understand the Visual Basic documentation and samples.

Visual Studio IDE and Tools Support for LINQ (Visual Basic)

Describes Visual Studio's Object Relational Designer, debugger support for queries, and other IDE features related to LINQ.

Standard Query Operators Overview (Visual Basic)

Provides an introduction to the standard query operators. It also provides links to topics that have more information about each type of query operation.

LINQ to Objects (Visual Basic)

Includes links to topics that explain how to use LINQ to Objects to access in-memory data structures,

LINQ to XML (Visual Basic)

Includes links to topics that explain how to use LINQ to XML, which provides the in-memory document modification capabilities of the Document Object Model (DOM), and supports LINQ query expressions.

LINQ to ADO.NET (Portal Page)

Provides an entry point for documentation about LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet enables you to build richer query capabilities into DataSet by using the same query functionality that is available for other data sources. LINQ to SQL provides a run-time infrastructure for managing relational data as objects. LINQ to Entities enables developers to write queries against the Entity Framework conceptual model by using C#.

Enabling a Data Source for LINQ Querying

Provides an introduction to custom LINQ providers, LINQ expression trees, and other ways to extend LINQ.

Object-oriented programming (Visual Basic)

11/16/2018 • 10 minutes to read • Edit Online

Visual Basic provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object.

Inheritance describes the ability to create new classes based on an existing class.

Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

This section describes the following concepts:

- Classes and objects
  - Class members
    - Properties and fields
    - Methods
    - Constructors
    - Destructors
    - Events
    - Nested classes
  - Access modifiers and access levels
    - Instantiating classes
    - Shared classes and members
    - Anonymous types
- Inheritance
  - Overriding members
- Interfaces
- Generics
- Delegates

Classes and objects

The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable *instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

To define a class:

```
Class SampleClass
End Class
```

Visual Basic also provides a light version of classes called *structures* that are useful when you need to create large array of objects and do not want to consume too much memory for that.

To define a structure:

```
Structure SampleStructure
End Structure
```

For more information, see:

- Class Statement
- Structure Statement

Class members

Each class can have different *class members* that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects.

Properties and fields

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.

To define a field:

```
Class SampleClass
    Public SampleField As String
End Class
```

Properties have get and set procedures, which provide more control on how values are set or returned.

Visual Basic allows you either to create a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures.

To define an auto-implemented property:

```
Class SampleClass
    Public Property SampleProperty as String
End Class
```

If you need to perform some additional operations for reading and writing the property value, define a field for storing the property value and provide the basic logic for storing and retrieving it:

```
Class SampleClass
    Private m_Sample As String
    Public Property Sample() As String
        Get
            ' Return the value stored in the field.
            Return m_Sample
        End Get
        Set(ByVal Value As String)
            ' Store the value in the field.
            m_Sample = Value
        End Set
    End Property
End Class
```

Most properties have methods or procedures to both set and get the property value. However, you can create read-only or write-only properties to restrict them from being modified or read. In Visual Basic you can use `ReadOnly` and `WriteOnly` keywords. However, auto-implemented properties cannot be read-only or write-only.

For more information, see:

- Property Statement
- Get Statement
- Set Statement
- ReadOnly
- WriteOnly

Methods

A *method* is an action that an object can perform.

> **NOTE**
> In Visual Basic, there are two ways to create a method: the `Sub` statement is used if the method does not return a value; the `Function` statement is used if a method returns a value.

To define a method of a class:

```
Class SampleClass
    Public Function SampleFunc(ByVal SampleParam As String)
        ' Add code here
    End Function
End Class
```

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

To overload a method:

```
Overloads Sub Display(ByVal theChar As Char)
    ' Add code that displays Char data.
End Sub
Overloads Sub Display(ByVal theInteger As Integer)
    ' Add code that displays Integer data.
End Sub
```

In most cases you declare a method within a class definition. However, Visual Basic also supports *extension methods* that allow you to add methods to an existing class outside the actual definition of the class.

For more information, see:

- Function Statement
- Sub Statement
- Overloads
- Extension Methods

Constructors

Constructors are class methods that are executed automatically when an object of a given type is created. Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for any other method.

To define a constructor for a class:

```
Class SampleClass
    Sub New(ByVal s As String)
        // Add code here.
    End Sub
End Class
```

For more information, see: Object Lifetime: How Objects Are Created and Destroyed.

Destructors

Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates. There can be only one destructor for a class.

For more information about destructors and garbage collection in the .NET Framework, see Garbage Collection.

Events

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* and the classes that receive (or handle) the event are called *subscribers*. For more information about events, how they are raised and handled, see Events.

- To declare events, use the Event Statement.
- To raise events, use the RaiseEvent Statement.
- To specify event handlers using a declarative way, use the WithEvents statement and the Handles clause.
- To be able to dynamically add, remove, and change the event handler associated with an event, use the AddHandler Statement and RemoveHandler Statement together with the AddressOf Operator.

Nested classes

A class defined within another class is called *nested*. By default, the nested class is private.

```
Class Container
    Class Nested
    ' Add code here.
    End Class
End Class
```

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

```
Dim nestedInstance As Container.Nested = New Container.Nested()
```

Access modifiers and access levels

All classes and class members can specify what access level they provide to other classes by using *access modifiers*.

The following access modifiers are available:

VISUAL BASIC MODIFIER	DEFINITION
Public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.

VISUAL BASIC MODIFIER	DEFINITION
Private	The type or member can only be accessed by code in the same class.
Protected	The type or member can only be accessed by code in the same class or in a derived class.
Friend	The type or member can be accessed by any code in the same assembly, but not from another assembly.
`Protected Friend`	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

For more information, see Access levels in Visual Basic.

Instantiating classes

To create an object, you need to instantiate a class, or create a class instance.

```
Dim sampleObject as New SampleClass()
```

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
' Set a property value.
sampleObject.SampleProperty = "Sample String"
' Call a method.
sampleObject.SampleMethod()
```

To assign values to properties during the class instantiation process, use object initializers:

```
Dim sampleObject = New SampleClass With
    {.FirstProperty = "A", .SecondProperty = "B"}
```

For more information, see:

- New Operator
- Object Initializers: Named and Anonymous Types

Shared classes and members

A shared member of the class is a property, procedure, or field that is shared by all instances of a class.

To define a shared member:

```
Class SampleClass
    Public Shared SampleString As String = "Sample String"
End Class
```

To access the shared member, use the name of the class without creating an object of this class:

```
MsgBox(SampleClass.SampleString)
```

Shared modules in Visual Basic have shared members only and cannot be instantiated. Shared members also cannot access non-shared properties, fields or methods

For more information, see:

- Shared
- Module Statement

Anonymous types

Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

```
' sampleObject is an instance of a simple anonymous type.
Dim sampleObject =
    New With {Key .FirstProperty = "A", .SecondProperty = "B"}
```

For more information, see: Anonymous Types.

Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in Visual Basic implicitly inherit from the Object class that supports .NET class hierarchy and provides low-level services to all classes.

> **NOTE**
> Visual Basic doesn't support multiple inheritance. That is, you can specify only one base class for a derived class.

To inherit from a base class:

```
Class DerivedClass
    Inherits BaseClass
End Class
```

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

To specify that a class cannot be used as a base class:

```
NotInheritable Class SampleClass
End Class
```

To specify that a class can be used as a base class only and cannot be instantiated:

```
MustInherit Class BaseClass
End Class
```

For more information, see:

- Inherits Statement
- NotInheritable
- MustInherit

Overriding members

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

VISUAL BASIC MODIFIER	DEFINITION
Overridable	Allows a class member to be overridden in a derived class.
Overrides	Overrides a virtual (overridable) member defined in the base class.
NotOverridable	Prevents a member from being overridden in an inheriting class.
MustOverride	Requires that a class member to be overridden in the derived class.
Shadows	Hides a member inherited from a base class

Interfaces

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface:

```
Public Interface ISampleInterface
    Sub DoSomething()
End Interface
```

To implement an interface in a class:

```
Class SampleClass
    Implements ISampleInterface
    Sub DoSomething
        ' Method implementation.
    End Sub
End Class
```

For more information, see:

- Interfaces
- Interface Statement
- Implements Statement

Generics

Classes, structures, interfaces and methods in .NET can include *type parameters* that define types of objects that they can store or use. The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.

To define a generic class:

```
Class SampleGeneric(Of T)
    Public Field As T
End Class
```

To create an instance of a generic class:

```
Dim sampleObject As New SampleGeneric(Of String)
sampleObject.Field = "Sample string"
```

For more information, see:

- Generics
- Generic Types in Visual Basic

Delegates

A *delegate* is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

> **NOTE**
> Event handlers are nothing more than methods that are invoked through delegates. For more information about using delegates in event handling, see Events.

To create a delegate:

```
Delegate Sub SampleDelegate(ByVal str As String)
```

To create a reference to a method that matches the signature specified by the delegate:

```
Class SampleClass
    ' Method that matches the SampleDelegate signature.
    Sub SampleSub(ByVal str As String)
        ' Add code here.
    End Sub
    ' Method that instantiates the delegate.
    Sub SampleDelegateSub()
        Dim sd As SampleDelegate = AddressOf SampleSub
        sd("Sample string")
    End Sub
End Class
```

For more information, see:

- Delegates
- Delegate Statement
- AddressOf Operator

See also

- Visual Basic Programming Guide

Reflection (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Reflection provides objects (of type Type) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the static method `GetType` - inherited by all types from the `Object` base class - to obtain the type of a variable:

```
' Using GetType to obtain type information:
Dim i As Integer = 42
Dim type As System.Type = i.GetType()
System.Console.WriteLine(type)
```

The output is:

```
System.Int32
```

The following example uses reflection to obtain the full name of the loaded assembly.

```
' Using Reflection to get information from an Assembly:
Dim info As System.Reflection.Assembly = GetType(System.Int32).Assembly
System.Console.WriteLine(info)
```

The output is:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Reflection Overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the topic Dynamically Loading and Using Types.

Related Sections

For more information:

- Reflection
- Viewing Type Information
- Reflection and Generic Types

- System.Reflection.Emit
- Retrieving Information Stored in Attributes

See also

- Visual Basic Programming Guide
- Assemblies in the Common Language Runtime

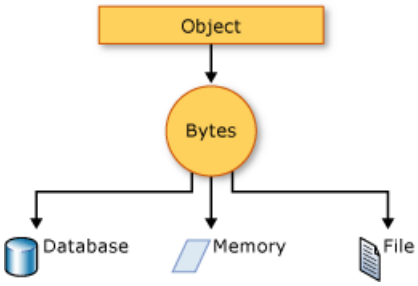
Serialization (Visual Basic)

3/25/2019 • 3 minutes to read • Edit Online

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How Serialization Works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for Serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Making an Object Serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a Formatter. System.Runtime.Serialization contains the classes necessary for serializing and deserializing objects.

Apply the SerializableAttribute attribute to a type to indicate that instances of this type can be serialized. A SerializationException exception is thrown if you attempt to serialize but the type does not have the SerializableAttribute attribute.

If you do not want a field within your class to be serializable, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

Binary and XML Serialization

Either binary or XML serialization can be used. In binary serialization, all members, even those that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, as well as greater flexibility of object sharing and usage for interoperability purposes.

Binary Serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

XML Serialization

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. System.Xml.Serialization contains the classes necessary for serializing and deserializing XML.

You can apply attributes to classes and class members in order to control the way the XmlSerializer serializes or deserializes an instance of the class.

Basic and Custom Serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

Basic Serialization

The only requirement in basic serialization is that the object has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems, in which case custom serialization may be preferable. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

Custom Serialization

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface.

If you want your object to be deserialized in a custom manner as well, you must use a custom constructor.

Designer Serialization

Designer serialization is a special form of serialization that involves the kind of object persistence usually associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

Walkthrough: Persisting an Object in Visual Studio (Visual Basic)
Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

How to: Read Object Data from an XML File (Visual Basic)
Shows how to read object data that was previously written to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (Visual Basic)
Shows how to write the object from a class to an XML file using the XmlSerializer class.

Program Structure and Code Conventions (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

This section introduces the typical Visual Basic program structure, provides a simple Visual Basic program, "Hello, World", and discusses Visual Basic code conventions. Code conventions are suggestions that focus not on a program's logic but on its physical structure and appearance. Following them makes your code easier to read, understand, and maintain. Code conventions can include, among others:

- Standardized formats for labeling and commenting code.
- Guidelines for spacing, formatting, and indenting code.
- Naming conventions for objects, variables, and procedures.

The following topics present a set of programming guidelines for Visual Basic programs, along with examples of good usage.

In This Section

Structure of a Visual Basic Program
Provides an overview of the elements that make up a Visual Basic program.

Main Procedure in Visual Basic
Discusses the procedure that serves as the starting point and overall control for your application.

References and the Imports Statement
Discusses how to reference objects in other assemblies.

Namespaces in Visual Basic
Describes how namespaces organize objects within assemblies.

Visual Basic Naming Conventions
Includes general guidelines for naming procedures, constants, variables, arguments, and objects.

Visual Basic Coding Conventions
Reviews the guidelines used in developing the samples in this documentation.

Conditional Compilation
Describes how to compile particular blocks of code selectively while directing the compiler to ignore others.

How to: Break and Combine Statements in Code
Shows how to divide long statements into multiple lines and combine short statements on one line.

How to: Collapse and Hide Sections of Code
Shows how to collapse and hide sections of code in the Visual Basic code editor.

How to: Label Statements
Shows how to mark a line of code to identify it for use with statements such as `On Error Goto`.

Special Characters in Code
Shows how and where to use non-numeric and non-alphabetic characters.

Comments in Code

Discusses how to add descriptive comments to your code.

Keywords as Element Names in Code

Describes how to use brackets (`[]`) to delimit variable names that are also Visual Basic keywords.

Me, My, MyBase, and MyClass

Describes various ways to refer to elements of a Visual Basic program.

Visual Basic Limitations

Discusses the removal of known coding limits within Visual Basic.

Related Sections

Typographic and Code Conventions

Provides standard coding conventions for Visual Basic.

Writing Code

Describes features that make it easier for you to write and manage your code.

Structure of a Visual Basic Program

4/28/2019 • 3 minutes to read • Edit Online

A Visual Basic program is built up from standard building blocks. A *solution* comprises one or more projects. A *project* in turn can contain one or more assemblies. Each *assembly* is compiled from one or more source files. A *source file* provides the definition and implementation of classes, structures, modules, and interfaces, which ultimately contain all your code.

For more information about these building blocks of a Visual Basic program, see Solutions and Projects and Assemblies in .NET.

File-Level Programming Elements

When you start a project or file and open the code editor, you see some code already in place and in the correct order. Any code that you write should follow the following sequence:

1. `Option` statements
2. `Imports` statements
3. `Namespace` statements and namespace-level elements

If you enter statements in a different order, compilation errors can result.

A program can also contain conditional compilation statements. You can intersperse these in the source file among the statements of the preceding sequence.

Option Statements

`Option` statements establish ground rules for subsequent code, helping prevent syntax and logic errors. The Option Explicit Statement ensures that all variables are declared and spelled correctly, which reduces debugging time. The Option Strict Statement helps to minimize logic errors and data loss that can occur when you work between variables of different data types. The Option Compare Statement specifies the way strings are compared to each other, based on either their `Binary` or `Text` values.

Imports Statements

You can include an Imports Statement (.NET Namespace and Type) to import names defined outside your project. An `Imports` statement allows your code to refer to classes and other types defined within the imported namespace, without having to qualify them. You can use as many `Imports` statements as appropriate. For more information, see References and the Imports Statement.

Namespace Statements

Namespaces help you organize and classify your programming elements for ease of grouping and accessing. You use the Namespace Statement to classify the following statements within a particular namespace. For more information, see Namespaces in Visual Basic.

Conditional Compilation Statements

Conditional compilation statements can appear almost anywhere in your source file. They cause parts of your code to be included or excluded at compile time depending on certain conditions. You can also use them for debugging your application, because conditional code runs in debugging mode only. For more information, see Conditional Compilation.

Namespace-Level Programming Elements

Classes, structures, and modules contain all the code in your source file. They are *namespace-level* elements, which can appear within a namespace or at the source file level. They hold the declarations of all other programming elements. Interfaces, which define element signatures but provide no implementation, also appear at module level. For more information on the module-level elements, see the following:

- Class Statement
- Structure Statement
- Module Statement
- Interface Statement

Data elements at namespace level are enumerations and delegates.

Module-Level Programming Elements

Procedures, operators, properties, and events are the only programming elements that can hold executable code (statements that perform actions at run time). They are the *module-level* elements of your program. For more information on the procedure-level elements, see the following:

- Function Statement
- Sub Statement
- Declare Statement
- Operator Statement
- Property Statement
- Event Statement

Data elements at module level are variables, constants, enumerations, and delegates.

Procedure-Level Programming Elements

Most of the contents of *procedure-level* elements are executable statements, which constitute the run-time code of your program. All executable code must be in some procedure (`Function` , `Sub` , `Operator` , `Get` , `Set` , `AddHandler` , `RemoveHandler` , `RaiseEvent`). For more information, see Statements.

Data elements at procedure level are limited to local variables and constants.

The Main Procedure

The `Main` procedure is the first code to run when your application has been loaded. `Main` serves as the starting point and overall control for your application. There are four varieties of `Main` :

- `Sub Main()`
- `Sub Main(ByVal cmdArgs() As String)`
- `Function Main() As Integer`
- `Function Main(ByVal cmdArgs() As String) As Integer`

The most common variety of this procedure is `Sub Main()` . For more information, see Main Procedure in Visual Basic.

See also

- Main Procedure in Visual Basic
- Visual Basic Naming Conventions
- Visual Basic Limitations

Main Procedure in Visual Basic

8/22/2019 • 3 minutes to read • Edit Online

Every Visual Basic application must contain a procedure called `Main`. This procedure serves as the starting point and overall control for your application. The .NET Framework calls your `Main` procedure when it has loaded your application and is ready to pass control to it. Unless you are creating a Windows Forms application, you must write the `Main` procedure for applications that run on their own.

`Main` contains the code that runs first. In `Main`, you can determine which form is to be loaded first when the program starts, find out if a copy of your application is already running on the system, establish a set of variables for your application, or open a database that the application requires.

Requirements for the Main Procedure

A file that runs on its own (usually with extension .exe) must contain a `Main` procedure. A library (for example with extension .dll) does not run on its own and does not require a `Main` procedure. The requirements for the different types of projects you can create are as follows:

- Console applications run on their own, and you must supply at least one `Main` procedure.
- Windows Forms applications run on their own. However, the Visual Basic compiler automatically generates a `Main` procedure in such an application, and you do not need to write one.
- Class libraries do not require a `Main` procedure. These include Windows Control Libraries and Web Control Libraries. Web applications are deployed as class libraries.

Declaring the Main Procedure

There are four ways to declare the `Main` procedure. It can take arguments or not, and it can return a value or not.

> **NOTE**
> If you declare `Main` in a class, you must use the `Shared` keyword. In a module, `Main` does not need to be `Shared`.

- The simplest way is to declare a `Sub` procedure that does not take arguments or return a value.

```
Module mainModule
    Sub Main()
        MsgBox("The Main procedure is starting the application.")
        ' Insert call to appropriate starting place in your code.
        MsgBox("The application is terminating.")
    End Sub
End Module
```

- `Main` can also return an `Integer` value, which the operating system uses as the exit code for your program. Other programs can test this code by examining the Windows ERRORLEVEL value. To return an exit code, you must declare `Main` as a `Function` procedure instead of a `Sub` procedure.

```
Module mainModule
    Function Main() As Integer
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' Insert call to appropriate starting place in your code.
        ' On return, assign appropriate value to returnValue.
        ' 0 usually means successful completion.
        MsgBox("The application is terminating with error level " &
             CStr(returnValue) & ".")
        Return returnValue
    End Function
End Module
```

- `Main` can also take a `String` array as an argument. Each string in the array contains one of the command-line arguments used to invoke your program. You can take different actions depending on their values.

```
Module mainModule
    Function Main(ByVal cmdArgs() As String) As Integer
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' See if there are any arguments.
        If cmdArgs.Length > 0 Then
            For argNum As Integer = 0 To UBound(cmdArgs, 1)
                ' Insert code to examine cmdArgs(argNum) and take
                ' appropriate action based on its value.
            Next
        End If
        ' Insert call to appropriate starting place in your code.
        ' On return, assign appropriate value to returnValue.
        ' 0 usually means successful completion.
        MsgBox("The application is terminating with error level " &
             CStr(returnValue) & ".")
        Return returnValue
    End Function
End Module
```

- You can declare `Main` to examine the command-line arguments but not return an exit code, as follows.

```
Module mainModule
    Sub Main(ByVal cmdArgs() As String)
        MsgBox("The Main procedure is starting the application.")
        Dim returnValue As Integer = 0
        ' See if there are any arguments.
        If cmdArgs.Length > 0 Then
            For argNum As Integer = 0 To UBound(cmdArgs, 1)
                ' Insert code to examine cmdArgs(argNum) and take
                ' appropriate action based on its value.
            Next
        End If
        ' Insert call to appropriate starting place in your code.
        MsgBox("The application is terminating.")
    End Sub
End Module
```

See also

- MsgBox
- Length
- UBound
- Structure of a Visual Basic Program

- /main
- Shared
- Sub Statement
- Function Statement
- Integer Data Type
- String Data Type

References and the Imports Statement (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

You can make external objects available to your project by choosing the **Add Reference** command on the **Project** menu. References in Visual Basic can point to assemblies, which are like type libraries but contain more information.

The Imports Statement

Assemblies include one or more namespaces. When you add a reference to an assembly, you can also add an `Imports` statement to a module that controls the visibility of that assembly's namespaces within the module. The `Imports` statement provides a scoping context that lets you use only the portion of the namespace necessary to supply a unique reference.

The `Imports` statement has the following syntax:

```
Imports [Aliasname =] Namespace
```

`Aliasname` refers to a short name you can use within code to refer to an imported namespace. `Namespace` is a namespace available through either a project reference, through a definition within the project, or through a previous `Imports` statement.

A module may contain any number of `Imports` statements. They must appear after any `Option` statements, if present, but before any other code.

> **NOTE**
>
> Do not confuse project references with the `Imports` statement or the `Declare` statement. Project references make external objects, such as objects in assemblies, available to Visual Basic projects. The `Imports` statement is used to simplify access to project references, but does not provide access to these objects. The `Declare` statement is used to declare a reference to an external procedure in a dynamic-link library (DLL).

Using Aliases with the Imports Statement

The `Imports` statement makes it easier to access methods of classes by eliminating the need to explicitly type the fully qualified names of references. Aliases let you assign a friendlier name to just one part of a namespace. For example, the carriage return/line feed sequence that causes a single piece of text to be displayed on multiple lines is part of the ControlChars module in the Microsoft.VisualBasic namespace. To use this constant in a program without an alias, you would need to type the following code:

```
MsgBox("Some text" & Microsoft.VisualBasic.ControlChars.CrLf &
       "Some more text")
```

`Imports` statements must always be the first lines immediately following any `Option` statements in a module. The following code fragment shows how to import and assign an alias to the Microsoft.VisualBasic.ControlChars module:

```
Imports CtrlChrs = Microsoft.VisualBasic.ControlChars
```

Future references to this namespace can be considerably shorter:

```
MsgBox("Some text" & CtrlChrs.CrLf & "Some more text")
```

If an `Imports` statement does not include an alias name, elements defined within the imported namespace can be used in the module without qualification. If the alias name is specified, it must be used as a qualifier for names contained within that namespace.

See also

- ControlChars
- Microsoft.VisualBasic
- Namespaces in Visual Basic
- Assemblies in .NET
- How to: Create and Use Assemblies Using the Command Line
- Imports Statement (.NET Namespace and Type)

Namespaces in Visual Basic

5/14/2019 • 5 minutes to read • Edit Online

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

For example, the .NET Framework defines the ListBox class in the System.Windows.Forms namespace. The following code fragment shows how to declare a variable using the fully qualified name for this class:

```
Dim LBox As System.Windows.Forms.ListBox
```

Avoiding Name Collisions

.NET Framework namespaces address a problem sometimes called *namespace pollution*, in which the developer of a class library is hampered by the use of similar names in another library. These conflicts with existing components are sometimes called *name collisions*.

For example, if you create a new class named `ListBox`, you can use it inside your project without qualification. However, if you want to use the .NET Framework ListBox class in the same project, you must use a fully qualified reference to make the reference unique. If the reference is not unique, Visual Basic produces an error stating that the name is ambiguous. The following code example demonstrates how to declare these objects:

```
' Define a new object based on your ListBox class.
Dim LBC As New ListBox
' Define a new Windows.Forms ListBox control.
Dim MyLB As New System.Windows.Forms.ListBox
```

The following illustration shows two namespace hierarchies, both containing an object named `ListBox`:

By default, every executable file you create with Visual Basic contains a namespace with the same name as your project. For example, if you define an object within a project named `ListBoxProject`, the executable file ListBoxProject.exe contains a namespace called `ListBoxProject`.

Multiple assemblies can use the same namespace. Visual Basic treats them as a single set of names. For example, you can define classes for a namespace called `SomeNameSpace` in an assembly named `Assemb1`, and define additional classes for the same namespace from an assembly named `Assemb2`.

Fully Qualified Names

Fully qualified names are object references that are prefixed with the name of the namespace in which the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing **Add Reference** from the **Project** menu) and then use the fully qualified name for the object in your code. The following code fragment shows how to use the fully qualified name for an object from another project's namespace:

```
Dim LBC As New ListBoxProject.Form1.ListBox
```

Fully qualified names prevent naming conflicts because they make it possible for the compiler to determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, you can use the `Imports` statement to define an *alias*—an abbreviated name you can use in place of a fully qualified name. For example, the following code example creates aliases for two fully qualified names, and uses these aliases to define two objects.

```
Imports LBControl = System.Windows.Forms.ListBox
Imports MyListBox = ListBoxProject.Form1.ListBox
```

```
Dim LBC As LBControl
Dim MyLB As MyListBox
```

If you use the `Imports` statement without an alias, you can use all the names in that namespace without qualification, provided they are unique to the project. If your project contains `Imports` statements for namespaces that contain items with the same name, you must fully qualify that name when you use it. Suppose, for example, your project contained the following two `Imports` statements:

```
' This namespace contains a class called Class1.
Imports MyProj1
' This namespace also contains a class called Class1.
Imports MyProj2
```

If you attempt to use `Class1` without fully qualifying it, Visual Basic produces an error stating that the name `Class1` is ambiguous.

Namespace Level Statements

Within a namespace, you can define items such as modules, interfaces, classes, delegates, enumerations, structures, and other namespaces. You cannot define items such as properties, procedures, variables and events at the namespace level. These items must be declared within containers such as modules, structures, or classes.

Global Keyword in Fully Qualified Names

If you have defined a nested hierarchy of namespaces, code inside that hierarchy might be blocked from accessing the System namespace of the .NET Framework. The following example illustrates a hierarchy in which the `SpecialSpace.System` namespace blocks access to System.

```
Namespace SpecialSpace
    Namespace System
        Class abc
            Function getValue() As System.Int32
                Dim n As System.Int32
                Return n
            End Function
        End Class
    End Namespace
End Namespace
```

As a result, the Visual Basic compiler cannot successfully resolve the reference to System.Int32, because `SpecialSpace.System` does not define `Int32`. You can use the `Global` keyword to start the qualification chain at

the outermost level of the .NET Framework class library. This allows you to specify the System namespace or any other namespace in the class library. The following example illustrates this.

```
Namespace SpecialSpace
    Namespace System
        Class abc
            Function getValue() As Global.System.Int32
                Dim n As Global.System.Int32
                Return n
            End Function
        End Class
    End Namespace
End Namespace
```

You can use `Global` to access other root-level namespaces, such as Microsoft.VisualBasic, and any namespace associated with your project.

Global Keyword in Namespace Statements

You can also use the `Global` keyword in a Namespace Statement. This lets you define a namespace out of the root namespace of your project.

All namespaces in your project are based on the root namespace for the project. Visual Studio assigns your project name as the default root namespace for all code in your project. For example, if your project is named `ConsoleApplication1`, its programming elements belong to namespace `ConsoleApplication1`. If you declare `Namespace Magnetosphere`, references to `Magnetosphere` in the project will access `ConsoleApplication1.Magnetosphere`.

The following examples use the `Global` keyword to declare a namespace out of the root namespace for the project.

```
Namespace Global.Magnetosphere

End Namespace


Namespace Global
    Namespace Magnetosphere

    End Namespace
End Namespace
```

In a namespace declaration, `Global` cannot be nested in another namespace.

You can use the Application Page, Project Designer (Visual Basic) to view and modify the **Root Namespace** of the project. For new projects, the **Root Namespace** defaults to the project name. To cause `Global` to be the top-level namespace, you can clear the **Root Namespace** entry so that the box is empty. Clearing **Root Namespace** removes the need for the `Global` keyword in namespace declarations.

If a `Namespace` statement declares a name that is also a namespace in the .NET Framework, the .NET Framework namespace becomes unavailable if the `Global` keyword is not used in a fully qualified name. To enable access to that .NET Framework namespace without using the `Global` keyword, you can include the `Global` keyword in the `Namespace` statement.

The following example has the `Global` keyword in the `System.Text` namespace declaration.

If the `Global` keyword was not present in the namespace declaration, StringBuilder could not be accessed without specifying `Global.System.Text.StringBuilder`. For a project named `ConsoleApplication1`, references to

`System.Text` would access `ConsoleApplication1.System.Text` if the `Global` keyword was not used.

```
Module Module1
    Sub Main()
        Dim encoding As New System.Text.TitanEncoding

        ' If the namespace defined below is System.Text
        ' instead of Global.System.Text, then this statement
        ' causes a compile-time error.
        Dim sb As New System.Text.StringBuilder
    End Sub
End Module

Namespace Global.System.Text
    Class TitanEncoding

    End Class
End Namespace
```

See also

- ListBox
- System.Windows.Forms
- Assemblies in .NET
- How to: Create and Use Assemblies Using the Command Line
- References and the Imports Statement
- Imports Statement (.NET Namespace and Type)
- Writing Code in Office Solutions

Visual Basic Naming Conventions

4/28/2019 • 2 minutes to read • Edit Online

When you name an element in your Visual Basic application, the first character of that name must be an alphabetic character or an underscore. Note, however, that names beginning with an underscore are not compliant with the Language Independence and Language-Independent Components (CLS).

The following suggestions apply to naming.

- Begin each separate word in a name with a capital letter, as in `FindLastRecord` and `RedrawMyForm`.
- Begin function and method names with a verb, as in `InitNameArray` or `CloseDialog`.
- Begin class, structure, module, and property names with a noun, as in `EmployeeName` or `CarAccessory`.
- Begin interface names with the prefix "I", followed by a noun or a noun phrase, like `IComponent`, or with an adjective describing the interface's behavior, like `IPersistable`. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.
- Begin event handler names with a noun describing the type of event followed by the "`EventHandler`" suffix, as in "`MouseEventHandler`".
- In names of event argument classes, include the "`EventArgs`" suffix.
- If an event has a concept of "before" or "after," use a suffix in present or past tense, as in '`ControlAdd`" or "`ControlAdded`".
- For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" can lead to confusion.
- Avoid using names in an inner scope that are the same as names in an outer scope. Errors can result if the wrong variable is accessed. If a conflict occurs between a variable and the keyword of the same name, you must identify the keyword by preceding it with the appropriate type library. For example, if you have a variable called `Date`, you can use the intrinsic `Date` function only by calling DateTime.Date.

See also

- Keywords as Element Names in Code
- Me, My, MyBase, and MyClass
- Declared Element Names
- Program Structure and Code Conventions
- Visual Basic Language Reference

Visual Basic Coding Conventions

7/24/2019 • 6 minutes to read • Edit Online

Microsoft develops samples and documentation that follow the guidelines in this topic. If you follow the same coding conventions, you may gain the following benefits:

- Your code will have a consistent look, so that readers can better focus on content, not layout.
- Readers understand your code more quickly because they can make assumptions based on previous experience.
- You can copy, change, and maintain the code more easily.
- You help ensure that your code demonstrates "best practices" for Visual Basic.

Naming Conventions

- For information about naming guidelines, see Naming Guidelines topic.
- Do not use "My" or "my" as part of a variable name. This practice creates confusion with the `My` objects.
- You do not have to change the names of objects in auto-generated code to make them fit the guidelines.

Layout Conventions

- Insert tabs as spaces, and use smart indenting with four-space indents.
- Use **Pretty listing (reformatting) of code** to reformat your code in the code editor. For more information, see Options, Text Editor, Basic (Visual Basic).
- Use only one statement per line. Don't use the Visual Basic line separator character (:).
- Avoid using the explicit line continuation character "_" in favor of implicit line continuation wherever the language allows it.
- Use only one declaration per line.
- If **Pretty listing (reformatting) of code** doesn't format continuation lines automatically, manually indent continuation lines one tab stop. However, always left-align items in a list.

```
a As Integer,
b As Integer
```

- Add at least one blank line between method and property definitions.

Commenting Conventions

- Put comments on a separate line instead of at the end of a line of code.
- Start comment text with an uppercase letter, and end comment text with a period.
- Insert one space between the comment delimiter (') and the comment text.

```
' Here is a comment.
```

- Do not surround comments with formatted blocks of asterisks.

Program Structure

- When you use the `Main` method, use the default construct for new console applications, and use `My` for command-line arguments.

```
Sub Main()
  For Each argument As String In My.Application.CommandLineArgs
    ' Add code here to use the string variable.
  Next
End Sub
```

Language Guidelines

String Data Type

- Use string interpolation to concatenate short strings, as shown in the following code.

```
MsgBox($"hello{vbCrLf}goodbye")
```

- To append strings in loops, use the StringBuilder object.

```
Dim longString As New System.Text.StringBuilder
For count As Integer = 1 To 1000
  longString.Append(count)
Next
```

Relaxed Delegates in Event Handlers

Do not explicitly qualify the arguments (Object and EventArgs) to event handlers. If you are not using the event arguments that are passed to an event (for example, sender as Object, e as EventArgs), use relaxed delegates, and leave out the event arguments in your code:

```
Public Sub Form1_Load() Handles Form1.Load
End Sub
```

Unsigned Data Type

- Use `Integer` rather than unsigned types, except where they are necessary.

Arrays

- Use the short syntax when you initialize arrays on the declaration line. For example, use the following syntax.

  ```
  Dim letters1 As String() = {"a", "b", "c"}
  ```

  Do not use the following syntax.

  ```
  Dim letters2() As String = New String() {"a", "b", "c"}
  ```

- Put the array designator on the type, not on the variable. For example, use the following syntax:

```
Dim letters4 As String() = {"a", "b", "c"}
```

Do not use the following syntax:

```
Dim letters3() As String = {"a", "b", "c"}
```

- Use the { } syntax when you declare and initialize arrays of basic data types. For example, use the following syntax:

```
Dim letters5 As String() = {"a", "b", "c"}
```

Do not use the following syntax:

```
Dim letters6(2) As String
letters6(0) = "a"
letters6(1) = "b"
letters6(2) = "c"
```

Use the With Keyword

When you make a series of calls to one object, consider using the `With` keyword:

```
With orderLog
  .Log = "Application"
  .Source = "Application Name"
  .MachineName = "Computer Name"
End With
```

Use the Try...Catch and Using Statements when you use Exception Handling

Do not use `On Error Goto`.

Use the IsNot Keyword

Use the `IsNot` keyword instead of `Not...Is Nothing`.

New Keyword

- Use short instantiation. For example, use the following syntax:

```
Dim employees As New List(Of String)
```

The preceding line is equivalent to this:

```
Dim employees2 As List(Of String) = New List(Of String)
```

- Use object initializers for new objects instead of the parameterless constructor:

```
Dim orderLog As New EventLog With {
    .Log = "Application",
    .Source = "Application Name",
    .MachineName = "Computer Name"}
```

Event Handling

- Use `Handles` rather than `AddHandler`:

```
Private Sub ToolStripMenuItem1_Click() Handles ToolStripMenuItem1.Click
End Sub
```

- Use `AddressOf`, and do not instantiate the delegate explicitly:

```
Dim closeItem As New ToolStripMenuItem(
    "Close", Nothing, AddressOf ToolStripMenuItem1_Click)
Me.MainMenuStrip.Items.Add(closeItem)
```

- When you define an event, use the short syntax, and let the compiler define the delegate:

```
Public Event SampleEvent As EventHandler(Of SampleEventArgs)
' or
Public Event SampleEvent(ByVal source As Object,
                         ByVal e As SampleEventArgs)
```

- Do not verify whether an event is `Nothing` (null) before you call the `RaiseEvent` method. `RaiseEvent` checks for `Nothing` before it raises the event.

Using Shared Members

Call `Shared` members by using the class name, not from an instance variable.

Use XML Literals

XML literals simplify the most common tasks that you encounter when you work with XML (for example, load, query, and transform). When you develop with XML, follow these guidelines:

- Use XML literals to create XML documents and fragments instead of calling XML APIs directly.
- Import XML namespaces at the file or project level to take advantage of the performance optimizations for XML literals.
- Use the XML axis properties to access elements and attributes in an XML document.
- Use embedded expressions to include values and to create XML from existing values instead of using API calls such as the `Add` method:

```
Private Function GetHtmlDocument(
    ByVal items As IEnumerable(Of XElement)) As String

  Dim htmlDoc = <html>
                  <body>
                    <table border="0" cellspacing="2">
                      <%=
                        From item In items
                        Select <tr>
                                 <td style="width:480">
                                   <%= item.<title>.Value %>
                                 </td>
                                 <td><%= item.<pubDate>.Value %></td>
                               </tr>
                      %>
                    </table>
                  </body>
                </html>

  Return htmlDoc.ToString()
End Function
```

LINQ Queries

- Use meaningful names for query variables:

```
Dim seattleCustomers = From cust In customers
                       Where cust.City = "Seattle"
```

- Provide names for elements in a query to make sure that property names of anonymous types are correctly capitalized using Pascal casing:

```
Dim customerOrders = From customer In customers
                     Join order In orders
                       On customer.CustomerID Equals order.CustomerID
                     Select Customer = customer, Order = order
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and an order ID, rename them instead of leaving them as `Name` and `ID` in the result:

```
Dim customerOrders2 = From cust In customers
                      Join ord In orders
                        On cust.CustomerID Equals ord.CustomerID
                      Select CustomerName = cust.Name,
                             OrderID = ord.ID
```

- Use type inference in the declaration of query variables and range variables:

```
Dim customerList = From cust In customers
```

- Align query clauses under the `From` statement:

```
Dim newyorkCustomers = From cust In customers
                       Where cust.City = "New York"
                       Select cust.LastName, cust.CompanyName
```

- Use `Where` clauses before other query clauses so that later query clauses operate on the filtered set of data:

```
Dim newyorkCustomers2 = From cust In customers
                        Where cust.City = "New York"
                        Order By cust.LastName
```

- Use the `Join` clause to explicitly define a join operation instead of using the `Where` clause to implicitly define a join operation:

```
Dim customerList2 = From cust In customers
                    Join order In orders
                      On cust.CustomerID Equals order.CustomerID
                    Select cust, order
```

See also

- Secure Coding Guidelines

Conditional Compilation in Visual Basic

8/22/2019 • 2 minutes to read • Edit Online

In *conditional compilation*, particular blocks of code in a program are compiled selectively while others are ignored.

For example, you may want to write debugging statements that compare the speed of different approaches to the same programming task, or you may want to localize an application for multiple languages. Conditional compilation statements are designed to run during compile time, not at run time.

You denote blocks of code to be conditionally compiled with the `#If...Then...#Else` directive. For example, to create French- and German-language versions of the same application from the same source code, you embed platform-specific code segments in `#If...Then` statements using the predefined constants `FrenchVersion` and `GermanVersion`. The following example demonstrates how:

```
#If FrenchVersion Then
   ' <code specific to the French language version>.
#ElseIf GermanVersion Then
   ' <code specific to the German language version>.
#Else
        ' <code specific to other versions>.
#End If
```

If you set the value of the `FrenchVersion` conditional compilation constant to `True` at compile time, the conditional code for the French version is compiled. If you set the value of the `GermanVersion` constant to `True`, the compiler uses the German version. If neither is set to `True`, the code in the last `Else` block runs.

> **NOTE**
> Autocompletion will not function when editing code and using conditional compilation directives if the code is not part of the current branch.

Declaring Conditional Compilation Constants

You can set conditional compilation constants in one of three ways:

- In the **Project Designer**
- At the command line when using the command-line compiler
- In your code

Conditional compilation constants have a special scope and cannot be accessed from standard code. The scope of a conditional compilation constant is dependent on the way it is set. The following table lists the scope of constants declared using each of the three ways mentioned above.

HOW CONSTANT IS SET	SCOPE OF CONSTANT
Project Designer	Public to all files in the project
Command line	Public to all files passed to the command-line compiler

HOW CONSTANT IS SET	SCOPE OF CONSTANT
`#Const` statement in code	Private to the file in which it is declared

TO SET CONSTANTS IN THE PROJECT DESIGNER

- Before creating your executable file, set constants in the **Project Designer** by following the steps provided in Managing Project and Solution Properties.

TO SET CONSTANTS AT THE COMMAND LINE

- Use the **/d** switch to enter conditional compilation constants, as in the following example:

```
vbc MyProj.vb /d:conFrenchVersion=-1:conANSI=0
```

No space is required between the **/d** switch and the first constant. For more information, see /define (Visual Basic).
Command-line declarations override declarations entered in the **Project Designer**, but do not erase them. Arguments set in **Project Designer** remain in effect for subsequent compilations.
When writing constants in the code itself, there are no strict rules as to their placement, since their scope is the entire module in which they are declared.

TO SET CONSTANTS IN YOUR CODE

- Place the constants in the declaration block of the module in which they are used. This helps keep your code organized and easier to read.

Related Topics

TITLE	DESCRIPTION
Program Structure and Code Conventions	Provides suggestions for making your code easy to read and maintain.

Reference

#Const Directive

#If...Then...#Else Directives

/define (Visual Basic)

How to: Break and Combine Statements in Code (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

When writing your code, you might at times create lengthy statements that necessitate horizontal scrolling in the Code Editor. Although this doesn't affect the way your code runs, it makes it difficult for you or anyone else to read the code as it appears on the monitor. In such cases, you should consider breaking the single long statement into several lines.

To break a single statement into multiple lines

- Use the line-continuation character, which is an underscore (`_`), at the point at which you want the line to break. The underscore must be immediately preceded by a space and immediately followed by a line terminator (carriage return).

  > **NOTE**
  > In some cases, if you omit the line-continuation character, the Visual Basic compiler will implicitly continue the statement on the next line of code. For a list of syntax elements for which you can omit the line-continuation character, see "Implicit Line Continuation" in Statements.

  In the following example, the statement is broken into four lines with line-continuation characters terminating all but the last line.

  ```
  cmd.CommandText = _
      "SELECT * FROM Titles JOIN Publishers " _
      & "ON Publishers.PubId = Titles.PubID " _
      & "WHERE Publishers.State = 'CA'"
  ```

  Using this sequence makes your code easier to read, both online and when printed.

  The line-continuation character must be the last character on a line. You can't follow it with anything else on the same line.

  Some limitations exist as to where you can use the line-continuation character; for example, you can't use it in the middle of an argument name. You can break an argument list with the line-continuation character, but the individual names of the arguments must remain intact.

  You can't continue a comment by using a line-continuation character. The compiler doesn't examine the characters in a comment for special meaning. For a multiple-line comment, repeat the comment symbol (`'`) on each line.

Although placing each statement on a separate line is the recommended method, Visual Basic also allows you to place multiple statements on the same line.

To place multiple statements on the same line

- Separate the statements with a colon (`:`), as in the following example.

  ```
  text1.Text = "Hello" : text1.BackColor = System.Drawing.Color.Red
  ```

See also

- Program Structure and Code Conventions
- Statements

How to: Collapse and Hide Sections of Code (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

The `#Region` directive enables you to collapse and hide sections of code in Visual Basic files. The `#Region` directive lets you specify a block of code that you can expand or collapse when using the Visual Studio code editor. The ability to hide code selectively makes your files more manageable and easier to read. For more information, see Outlining.

`#Region` directives support code block semantics such as `#If...#End If`. This means they cannot begin in one block and end in another; the start and end must be in the same block. `#Region` directives are not supported within functions.

To collapse and hide a section of code

- Place the section of code between the `#Region` and `#End Region` statements, as in the following example:

```
#Region "This is the code to be collapsed"
    Private components As System.ComponentModel.Container
    Dim WithEvents Form1 As System.Windows.Forms.Form

    Private Sub InitializeComponent()
        components = New System.ComponentModel.Container
        Me.Text = "Form1"
    End Sub
#End Region
```

The `#Region` block can be used multiple times in a code file; thus, users can define their own blocks of procedures and classes that can, in turn, be collapsed. `#Region` blocks can also be nested within other `#Region` blocks.

> **NOTE**
> Hiding code does not prevent it from being compiled and does not affect `#If...#End If` statements.

See also

- Conditional Compilation
- #Region Directive
- #If...Then...#Else Directives
- Outlining

How to: Label Statements (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Statement blocks are made up of lines of code delimited by colons. Lines of code preceded by an identifying string or integer are said to be *labeled*. Statement labels are used to mark a line of code to identify it for use with statements such as `On Error Goto`.

Labels may be either valid Visual Basic identifiers—such as those that identify programming elements—or integer literals. A label must appear at the beginning of a line of source code and must be followed by a colon, regardless of whether it is followed by a statement on the same line.

The compiler identifies labels by checking whether the beginning of the line matches any already-defined identifier. If it does not, the compiler assumes it is a label.

Labels have their own declaration space and do not interfere with other identifiers. A label's scope is the body of the method. Label declaration takes precedence in any ambiguous situation.

> **NOTE**
> Labels can be used only on executable statements inside methods.

To label a line of code

- Place an identifier, followed by a colon, at the beginning of the line of source code.

  For example, the following lines of code are labeled with `Jump` and `120`, respectively:

```
Jump:   FileOpen(1, "testFile", OpenMode.Input)
        ' ...
120:    FileClose(1)
```

See also

- Statements
- Declared Element Names
- Program Structure and Code Conventions

Special Characters in Code (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Sometimes you have to use special characters in your code, that is, characters that are not alphabetical or numeric. The punctuation and special characters in the Visual Basic character set have various uses, from organizing program text to defining the tasks that the compiler or the compiled program performs. They do not specify an operation to be performed.

Parentheses

Use parentheses when you define a procedure, such as a `Sub` or `Function`. You must enclose all procedure argument lists in parentheses. You also use parentheses for putting variables or arguments into logical groups, especially to override the default order of operator precedence in a complex expression. The following example illustrates this.

```
Dim a, b, c, d, e As Double
a = 3.2
b = 7.6
c = 2
d = b + c / a
e = (b + c) / a
```

Following execution of the previous code, the value of `d` is 8.225 and the value of `e` is 3. The calculation for `d` uses the default precedence of `/` over `+` and is equivalent to `d = b + (c / a)`. The parentheses in the calculation for `e` override the default precedence.

Separators

Separators do what their name suggests: they separate sections of code. In Visual Basic, the separator character is the colon (`:`). Use separators when you want to include multiple statements on a single line instead of separate lines. This saves space and improves the readability of your code. The following example shows three statements separated by colons.

```
a = 3.2 : b = 7.6 : c = 2
```

For more information, see How to: Break and Combine Statements in Code.

The colon (`:`) character is also used to identify a statement label. For more information, see How to: Label Statements.

Concatenation

Use the `&` operator for *concatenation*, or linking strings together. Do not confuse it with the `+` operator, which adds together numeric values. If you use the `+` operator to concatenate when you operate on numeric values, you can obtain incorrect results. The following example demonstrates this.

```
var1 = "10.01"
var2 = 11
resultA = var1 + var2
resultB = var1 & var2
```

Following execution of the previous code, the value of `resultA` is 21.01 and the value of `resultB` is "10.0111".

Member Access Operators

To access a member of a type, you use the dot (`.`) or exclamation point (`!`) operator between the type name and the member name.

Dot (.) Operator

Use the `.` operator on a class, structure, interface, or enumeration as a member access operator. The member can be a field, property, event, or method. The following example illustrates this.

```
Dim nextForm As New System.Windows.Forms.Form
' Access Text member (property) of Form class (on nextForm object).
nextForm.Text = "This is the next form"
' Access Close member (method) on nextForm.
nextForm.Close()
```

Exclamation Point (!) Operator

Use the `!` operator only on a class or interface as a dictionary access operator. The class or interface must have a default property that accepts a single `String` argument. The identifier immediately following the `!` operator becomes the argument value passed to the default property as a string. The following example demonstrates this.

```
Public Class hasDefault
  Default Public ReadOnly Property index(ByVal s As String) As Integer
    Get
      Return 32768 + AscW(s)
    End Get
  End Property
End Class
Public Class testHasDefault
  Public Sub compareAccess()
    Dim hD As hasDefault = New hasDefault()
    MsgBox("Traditional access returns " & hD.index("X") & vbCrLf &
      "Default property access returns " & hD("X") & vbCrLf &
      "Dictionary access returns " & hD!X)
  End Sub
End Class
```

The three output lines of `MsgBox` all display the value `32856`. The first line uses the traditional access to property `index`, the second makes use of the fact that `index` is the default property of class `hasDefault`, and the third uses dictionary access to the class.

Note that the second operand of the `!` operator must be a valid Visual Basic identifier not enclosed in double quotation marks (`" "`). In other words, you cannot use a string literal or string variable. The following change to the last line of the `MsgBox` call generates an error because `"X"` is an enclosed string literal.

```
"Dictionary access returns " & hD!"X")
```

> **NOTE**
>
> References to default collections must be explicit. In particular, you cannot use the `!` operator on a late-bound variable.

The `!` character is also used as the `Single` type character.

See also

- Program Structure and Code Conventions
- Type Characters

Comments in Code (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

As you read the code examples, you often encounter the comment symbol (`'`). This symbol tells the Visual Basic compiler to ignore the text following it, or the *comment*. Comments are brief explanatory notes added to code for the benefit of those reading it.

It is good programming practice to begin all procedures with a brief comment describing the functional characteristics of the procedure (what it does). This is for your own benefit and the benefit of anyone else who examines the code. You should separate the implementation details (how the procedure does it) from comments that describe the functional characteristics. When you include implementation details in the description, remember to update them when you update the function.

Comments can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code.

```
' This is a comment beginning at the left edge of the screen.
text1.Text = "Hi!"    ' This is an inline comment.
```

If your comment requires more than one line, use the comment symbol on each line, as the following example illustrates.

```
' This comment is too long to fit on a single line, so we break
' it into two lines. Some comments might need three or more lines.
```

Commenting Guidelines

The following table provides general guidelines for what types of comments can precede a section of code. These are suggestions; Visual Basic does not enforce rules for adding comments. Write what works best, both for you and for anyone else who reads your code.

Comment type	Comment description
Purpose	Describes what the procedure does (not how it does it)
Assumptions	Lists each external variable, control, open file, or other element accessed by the procedure
Effects	Lists each affected external variable, control, or file, and the effect it has (only if it is not obvious)
Inputs	Specifies the purpose of the argument
Returns	Explains the values returned by the procedure

Remember the following points:

- Every important variable declaration should be preceded by a comment describing the use of the variable being declared.

- Variables, controls, and procedures should be named clearly enough that commenting is needed only for complex implementation details.
- Comments cannot follow a line-continuation sequence on the same line.

You can add or remove comment symbols for a block of code by selecting one or more lines of code and choosing the **Comment** ( ) and **Uncomment** ( ) buttons on the **Edit** toolbar.

> **NOTE**
>
> You can also add comments to your code by preceding the text with the `REM` keyword. However, the `'` symbol and the **Comment/Uncomment** buttons are easier to use and require less space and memory.

See also

- Basic Instincts - Documenting Your Code With XML Comments
- How to: Create XML Documentation
- XML Comment Tags
- Program Structure and Code Conventions
- REM Statement

Keywords as Element Names in Code (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Any program element — such as a variable, class, or member — can have the same name as a restricted keyword. For example, you can create a variable named `Loop`. However, to refer to your version of it — which has the same name as the restricted `Loop` keyword — you must either precede it with a full qualification string or enclose it in square brackets (`[ ]`), as the following example shows.

```
' The following statement precedes Loop with a full qualification string.
sampleForm.Loop.Visible = True
' The following statement encloses Loop in square brackets.
 [Loop].Visible = True
```

If you do not do either of these, then Visual Basic assumes use of the intrinsic `Loop` keyword and produces an error, as in the following example:

```
' The following statement causes a compiler error.
```

```
Loop.Visible = True
```

You can use square brackets when referring to forms and controls, and when declaring a variable or defining a procedure with the same name as a restricted keyword. It can be easy to forget to qualify names or include square brackets, and thus introduce errors into your code and make it harder to read. For this reason, we recommend that you not use restricted keywords as the names of program elements. However, if a future version of Visual Basic defines a new keyword that conflicts with an existing form or control name, then you can use this technique when updating your code to work with the new version.

> **NOTE**
>
> Your program also might include element names provided by other referenced assemblies. If these names conflict with restricted keywords, then placing square brackets around them causes Visual Basic to interpret them as your defined elements.

See also

- Visual Basic Naming Conventions
- Program Structure and Code Conventions
- Keywords

Me, My, MyBase, and MyClass in Visual Basic

5/14/2019 • 2 minutes to read • Edit Online

`Me`, `My`, `MyBase`, and `MyClass` in Visual Basic have similar names, but different purposes. This topic describes each of these entities in order to distinguish them.

Me

The `Me` keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. `Me` behaves like either an object variable or a structure variable referring to the current instance. Using `Me` is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure, or module.

For example, suppose you have the following procedure in a module.

```
Sub ChangeFormColor(FormName As Form)
   Randomize()
   FormName.BackColor = Color.FromArgb(Rnd() * 256, Rnd() * 256, Rnd() * 256)
End Sub
```

You can call this procedure and pass the current instance of the Form class as an argument by using the following statement.

```
ChangeFormColor(Me)
```

My

The `My` feature provides easy and intuitive access to a number of .NET Framework classes, enabling the Visual Basic user to interact with the computer, application, settings, resources, and so on.

MyBase

The `MyBase` keyword behaves like an object variable referring to the base class of the current instance of a class. `MyBase` is commonly used to access base class members that are overridden or shadowed in a derived class. `MyBase.New` is used to explicitly call a base class constructor from a derived class constructor.

MyClass

The `MyClass` keyword behaves like an object variable referring to the current instance of a class as originally implemented. `MyClass` is similar to `Me`, but all method calls on it are treated as if the method were `NotOverridable`.

See also

- Inheritance Basics

Visual Basic Limitations

4/28/2019 • 2 minutes to read • Edit Online

Earlier versions of Visual Basic enforced boundaries in code, such as the length of variable names, the number of variables allowed in modules, and module size. In Visual Basic .NET, these restrictions have been relaxed, giving you greater freedom in writing and arranging your code.

Physical limits are dependent more on run-time memory than on compile-time considerations. If you use prudent programming practices, and divide large applications into multiple classes and modules, then there is very little chance of encountering an internal Visual Basic limitation.

The following are some limitations that you might encounter in extreme cases:

- **Name Length.** There is a maximum number of characters for the name of every declared programming element. This maximum applies to an entire qualification string if the element name is qualified. See Declared Element Names.
- **Line Length.** There is a maximum of 65535 characters in a physical line of source code. The logical source code line can be longer if you use line continuation characters. See How to: Break and Combine Statements in Code.
- **Array Dimensions.** There is a maximum number of dimensions you can declare for an array. This limits how many indexes you can use to specify an array element. See Array Dimensions in Visual Basic.
- **String Length.** There is a maximum number of Unicode characters you can store in a single string. See String Data Type.
- **Environment String Length.** There is a maximum of 32768 characters for any environment string used as a command-line argument. This is a limitation on all platforms.

See also

- Program Structure and Code Conventions
- Visual Basic Naming Conventions

Visual Basic Language Features

9/7/2018 • 2 minutes to read • Edit Online

The following topics introduce and discuss the essential components of Visual Basic, an object-oriented programming language. After creating the user interface for your application using forms and controls, you need to write the code that defines the application's behavior. As with any modern programming language, Visual Basic supports a number of common programming constructs and language elements.

If you have programmed in other languages, much of the material covered in this section might seem familiar. While most of the constructs are similar to those in other languages, the event-driven nature of Visual Basic introduces some subtle differences.

If you are new to programming, the material in this section serves as an introduction to the basic building blocks for writing code. Once you understand the basics, you can create powerful applications using Visual Basic.

In This Section

Arrays
Discusses making your code more compact and powerful by declaring and using arrays, which hold multiple related values.

Collection Initializers
Describes collection initializers, which enable you to create a collection and populate it with an initial set of values.

Constants and Enumerations
Discusses storing unchanging values for repeated use, including sets of related constant values.

Control Flow
Shows how to regulate the flow of your program's execution.

Data Types
Describes what kinds of data a programming element can hold and how that data is stored.

Declared Elements
Covers programming elements you can declare, their names and characteristics, and how the compiler resolves references to them.

Delegates
Provides an introduction to delegates and how they are used in Visual Basic.

Early and Late Binding
Describes binding, which is performed by the compiler when an object is assigned to an object variable, and the differences between early-bound and late-bound objects.

Error Types
Provides an overview of syntax errors, run-time errors, and logic errors.

Events
Shows how to declare and use events.

Interfaces
Describes what interfaces are and how you can use them in your applications.

LINQ

Provides links to topics that introduce Language-Integrated Query (LINQ) features and programming.

Objects and Classes

Provides an overview of objects and classes, how they are used, their relationships to each other, and the properties, methods, and events they expose.

Operators and Expressions

Describes the code elements that manipulate value-holding elements, how to use them efficiently, and how to combine them to yield new values.

Procedures

Describes `Sub`, `Function`, `Property`, and `Operator` procedures, as well as advanced topics such as recursive and overloaded procedures.

Statements

Describes declaration and executable statements.

Strings

Provides links to topics that describe the basic concepts about using strings in Visual Basic.

Variables

Introduces variables and describes how to use them in Visual Basic.

XML

Provides links to topics that describe how to use XML in Visual Basic.

Related Sections

Collections

Describes some of the types of collections that are provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.

Visual Basic Language Reference

Provides reference information on various aspects of Visual Basic programming.

Arrays in Visual Basic

6/7/2019 • 28 minutes to read • Edit Online

An array is a set of values, which are termed *elements*, that are logically related to each other. For example, an array may consist of the number of students in each grade in a grammar school; each element of the array is the number of students in a single grade. Similarly, an array may consist of a student's grades for a class; each element of the array is a single grade.

It is possible individual variables to store each of our data items. For example, if our application analyzes student grades, we can use a separate variable for each student's grade, such as `englishGrade1`, `englishGrade2`, etc. This approach has three major limitations:

- We have to know at design time exactly how many grades we have to handle.
- Handling large numbers of grades quickly becomes unwieldy. This in turn makes an application much more likely to have serious bugs.
- It is difficult to maintain. Each new grade that we add requires that the application be modified, recompiled, and redeployed.

By using an array, you can refer to these related values by the same name, and use a number that's called an *index* or *subscript* to identify an individual element based on its position in the array. The indexes of an array range from 0 to one less than the total number of elements in the array. When you use Visual Basic syntax to define the size of an array, you specify its highest index, not the total number of elements in the array. You can work with the array as a unit, and the ability to iterate its elements frees you from needing to know exactly how many elements it contains at design time.

Some quick examples before explanation:

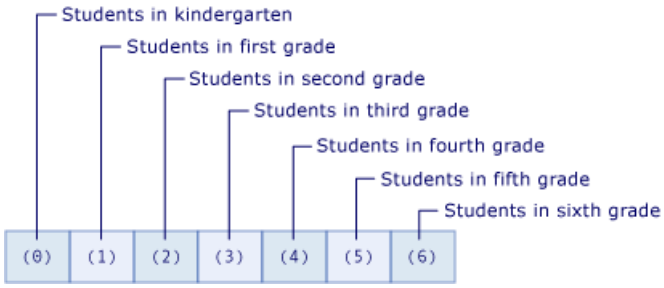
```
' Declare a single-dimension array of 5 numbers.
Dim numbers(4) As Integer

' Declare a single-dimension array and set its 4 values.
Dim numbers = New Integer() {1, 2, 4, 8}

' Change the size of an existing array to 16 elements and retain the current values.
ReDim Preserve numbers(15)

' Redefine the size of an existing array and reset the values.
ReDim numbers(15)

' Declare a 6 x 6 multidimensional array.
Dim matrix(5, 5) As Double

' Declare a 4 x 3 multidimensional array and set array element values.
Dim matrix = New Integer(3, 2) {{1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}}

' Declare a jagged array
Dim sales()() As Double = New Double(11)() {}
```

Array elements in a simple array

Let's create an array named `students` to store the number of students in each grade in a grammar school. The indexes of the elements range from 0 through 6. Using this array is simpler than declaring seven variables.

The following illustration shows the `students` array. For each element of the array:

- The index of the element represents the grade (index 0 represents kindergarten).
- The value that's contained in the element represents the number of students in that grade.



The following example contains the Visual Basic code that creates and uses the array:

```
Module SimpleArray
   Public Sub Main()
      ' Declare an array with 7 elements.
      Dim students(6) As Integer

      ' Assign values to each element.
      students(0) = 23
      students(1) = 19
      students(2) = 21
      students(3) = 17
      students(4) = 19
      students(5) = 20
      students(6) = 22

      ' Display the value of each element.
      For ctr As Integer = 0 To 6
         Dim grade As String = If(ctr = 0, "kindergarten", $"grade {ctr}")
         Console.WriteLine($"Students in {grade}: {students(ctr)}")
      Next
   End Sub
End Module
' The example displays the following output:
'     Students in kindergarten: 23
'     Students in grade 1: 19
'     Students in grade 2: 21
'     Students in grade 3: 17
'     Students in grade 4: 19
'     Students in grade 5: 20
'     Students in grade 6: 22
```

The example does three things:

- It declares a `students` array with seven elements. The number `6` in the array declaration indicates the last index in the array; it is one less than the number of elements in the array.
- It assigns values to each element in the array. Array elements are accessed by using the array name and including the index of the individual element in parentheses.
- It lists each value of the array. The example uses a `For` statement to access each element of the array by its index number.

The `students` array in the preceding example is a one-dimensional array because it uses one index. An array that uses more than one index or subscript is called *multidimensional*. For more information, see the rest of this article and Array Dimensions in Visual Basic.

Creating an array

You can define the size of an array in several ways:

- You can specify the size when the array is declared:

```
' Declare an array with 10 elements.
Dim cargoWeights(9) As Double
' Declare a 24 x 2 array.
Dim hourlyTemperatures(23, 1) As Integer
' Declare a jagged array with 31 elements.
Dim januaryInquiries(30)() As String
```

- You can use a `New` clause to supply the size of an array when it's created:

```
' Declare an array with 10 elements.
Dim cargoWeights() As Double = New Double(9) {}
' Declare a 24 x 2 array.
Dim hourlyTemperatures(,) As Integer = New Integer(23, 1) {}
' Declare a jagged array with 31 elements.
Dim januaryInquiries()() As String = New String(30)() {}
```

If you have an existing array, you can redefine its size by using the `ReDim` statement. You can specify that the `ReDim` statement keep the values that are in the array, or you can specify that it create an empty array. The following example shows different uses of the `ReDim` statement to modify the size of an existing array.

```
' Assign a new array size and retain the current values.
ReDim Preserve cargoWeights(20)
' Assign a new array size and retain only the first five values.
ReDim Preserve cargoWeights(4)
' Assign a new array size and discard all current element values.
ReDim cargoWeights(15)
```

For more information, see the ReDim Statement.

Storing values in an array

You can access each location in an array by using an index of type `Integer`. You can store and retrieve values in an array by referencing each array location by using its index enclosed in parentheses. Indexes for multidimensional arrays are separated by commas (,). You need one index for each array dimension.

The following example shows some statements that store and retrieve values in arrays.

```
Module Example
   Public Sub Main()
      ' Create a 10-element integer array.
      Dim numbers(9) As Integer
      Dim value As Integer = 2

      ' Write values to it.
      For ctr As Integer = 0 To 9
         numbers(ctr) = value
         value *= 2
      Next

      ' Read and sum the array values.
      Dim sum As Integer
      For ctr As Integer = 0 To 9
         sum += numbers(ctr)
      Next
      Console.WriteLine($"The sum of the values is {sum:N0}")
    End Sub
End Module
' The example displays the following output:
'     The sum of the values is 2,046
```

Populating an array with array literals

By using an array literal, you can populate an array with an initial set of values at the same time that you create it. An array literal consists of a list of comma-separated values that are enclosed in braces (`{}`).

When you create an array by using an array literal, you can either supply the array type or use type inference to determine the array type. The following example shows both options.

```
' Array literals with explicit type definition.
Dim numbers = New Integer() {1, 2, 4, 8}
' Array literals with type inference.
Dim doubles = {1.5, 2, 9.9, 18}
' Array literals with explicit type definition.
Dim articles() As String = { "the", "a", "an" }

' Array literals with explicit widening type definition.
Dim values() As Double = { 1, 2, 3, 4, 5 }
```

When you use type inference, the type of the array is determined by the *dominant type* in the list of literal values. The dominant type is the type to which all other types in the array can widen. If this unique type can't be determined, the dominant type is the unique type to which all other types in the array can narrow. If neither of these unique types can be determined, the dominant type is `Object`. For example, if the list of values that's supplied to the array literal contains values of type `Integer`, `Long`, and `Double`, the resulting array is of type `Double`. Because `Integer` and `Long` widen only to `Double`, `Double` is the dominant type. For more information, see Widening and Narrowing Conversions.

> **NOTE**
>
> You can use type inference only for arrays that are defined as local variables in a type member. If an explicit type definition is absent, arrays defined with array literals at the class level are of type `Object[]`. For more information, see Local type inference.

Note that the previous example defines `values` as an array of type `Double` even though all the array literals are of type `Integer`. You can create this array because the values in the array literal can widen to `Double`

values.

You can also create and populate a multidimensional array by using *nested array literals*. Nested array literals must have a number of dimensions that's consistent with the resulting array. The following example creates a two-dimensional array of integers by using nested array literals.

```
' Create and populate a 2 x 2 array.
Dim grid1 = {{1, 2}, {3, 4}}
' Create and populate a 2 x 2 array with 3 elements.
Dim grid2(,) = {{1, 2}, {3, 4}, {5, 6}}
```

When using nested array literals to create and populate an array, an error occurs if the number of elements in the nested array literals don't match. An error also occurs if you explicitly declare the array variable to have a different number of dimensions than the array literals.

Just as you can for one-dimensional arrays, you can rely on type inference when creating a multidimensional array with nested array literals. The inferred type is the dominant type for all the values in all the array literals for all nesting level. The following example creates a two-dimensional array of type `Double[,]` from values that are of type `Integer` and `Double`.

```
Dim arr = {{1, 2.0}, {3, 4}, {5, 6}, {7, 8}}
```

For additional examples, see How to: Initialize an Array Variable in Visual Basic.

Iterating through an array

When you iterate through an array, you access each element in the array from the lowest index to the highest or from the highest to the lowest. Typically, use either the For...Next Statement or the For Each...Next Statement to iterate through the elements of an array. When you don't know the upper bounds of the array, you can call the Array.GetUpperBound method to get the highest value of the index. Although lowest index value is almost always 0, you can call the Array.GetLowerBound method to get the lowest value of the index.

The following example iterates through a one-dimensional array by using the `For...Next` statement.

```
Module IterateArray
   Public Sub Main()
      Dim numbers = {10, 20, 30}

      For index = 0 To numbers.GetUpperBound(0)
         Console.WriteLine(numbers(index))
      Next
   End Sub
End Module
' The example displays the following output:
'   10
'   20
'   30
```

The following example iterates through a multidimensional array by using a `For...Next` statement. The GetUpperBound method has a parameter that specifies the dimension. `GetUpperBound(0)` returns the highest index of the first dimension, and `GetUpperBound(1)` returns the highest index of the second dimension.

```
Module IterateArray
   Public Sub Main()
      Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

      For index0 = 0 To numbers.GetUpperBound(0)
         For index1 = 0 To numbers.GetUpperBound(1)
            Console.Write($"{numbers(index0, index1)} ")
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
' Output
'  1 2
'  3 4
'  5 6
```

The following example uses a For Each...Next Statementto iterate through a one-dimensional array and a two-dimensional array.

```
Module IterateWithForEach
   Public Sub Main()
      ' Declare and iterate through a one-dimensional array.
      Dim numbers1 = {10, 20, 30}

      For Each number In numbers1
         Console.WriteLine(number)
      Next
      Console.WriteLine()

      Dim numbers = {{1, 2}, {3, 4}, {5, 6}}

      For Each number In numbers
         Console.WriteLine(number)
      Next
   End Sub
End Module
' The example displays the following output:
'  10
'  20
'  30
'
'  1
'  2
'  3
'  4
'  5
'  6
```

Array size

The size of an array is the product of the lengths of all its dimensions. It represents the total number of elements currently contained in the array. For example, the following example declares a 2-dimensional array with four elements in each dimension. As the output from the example shows, the array's size is 16 (or (3 + 1) * (3 + 1).

```
Module Example
   Public Sub Main()
      Dim arr(3, 3) As Integer
      Console.WriteLine(arr.Length)
   End Sub
End Module
' The example displays the following output:
'     16
```

> **NOTE**
>
> This discussion of array size does not apply to jagged arrays. For information on jagged arrays and determining the size of a jagged array, see the Jagged arrays section.

You can find the size of an array by using the Array.Length property. You can find the length of each dimension of a multidimensional array by using the Array.GetLength method.

You can resize an array variable by assigning a new array object to it or by using the `ReDim` Statement statement. The following example uses the `ReDim` statement to change a 100-element array to a 51-element array.

```
Module Example
   Public Sub Main()
      Dim arr(99) As Integer
      Console.WriteLine(arr.Length)

      Redim arr(50)
      Console.WriteLine(arr.Length)
   End Sub
End Module
' The example displays the following output:
'     100
'     51
```

There are several things to keep in mind when dealing with the size of an array.

Dimension Length	The index of each dimension is 0-based, which means it ranges from 0 to its upper bound. Therefore, the length of a given dimension is one greater than the declared upper bound of that dimension.
Length Limits	The length of every dimension of an array is limited to the maximum value of the `Integer` data type, which is Int32.MaxValue or (2 ^ 31) - 1. However, the total size of an array is also limited by the memory available on your system. If you attempt to initialize an array that exceeds the amount of available memory, the runtime throws an OutOfMemoryException.

Size and Element Size	An array's size is independent of the data type of its elements. The size always represents the total number of elements, not the number of bytes that they consume in memory.
Memory Consumption	It is not safe to make any assumptions regarding how an array is stored in memory. Storage varies on platforms of different data widths, so the same array can consume more memory on a 64-bit system than on a 32-bit system. Depending on system configuration when you initialize an array, the common language runtime (CLR) can assign storage either to pack elements as close together as possible, or to align them all on natural hardware boundaries. Also, an array requires a storage overhead for its control information, and this overhead increases with each added dimension.

The array type

Every array has a data type, which differs from the data type of its elements. There is no single data type for all arrays. Instead, the data type of an array is determined by the number of dimensions, or *rank*, of the array, and the data type of the elements in the array. Two array variables are of the same data type only when they have the same rank and their elements have the same data type. The lengths of the dimensions of an array do not influence the array data type.

Every array inherits from the System.Array class, and you can declare a variable to be of type `Array`, but you cannot create an array of type `Array`. For example, although the following code declares the `arr` variable to be of type `Array` and calls the Array.CreateInstance method to instantiate the array, the array's type proves to be Object[].

```
Module Example
   Public Sub Main()
      Dim arr As Array = Array.CreateInstance(GetType(Object), 19)
      Console.WriteLine(arr.Length)
      Console.WriteLine(arr.GetType().Name)
   End Sub
End Module
' The example displays the following output:
'     19
'     Object[]
```

Also, the ReDim Statement cannot operate on a variable declared as type `Array`. For these reasons, and for type safety, it is advisable to declare every array as a specific type.

You can find out the data type of either an array or its elements in several ways.

- You can call the GetType method on the variable to get a Type object that represents the run-time type of the variable. The Type object holds extensive information in its properties and methods.
- You can pass the variable to the TypeName function to get a `String` with the name of run-time type.

The following example calls the both the `GetType` method and the `TypeName` function to determine the type of an array. The array type is `Byte(,)`. Note that the Type.BaseType property also indicates that the base type of the byte array is the Array class.

```
Module Example
   Public Sub Main()
      Dim bytes(9,9) As Byte
      Console.WriteLine($"Type of {nameof(bytes)} array: {bytes.GetType().Name}")
      Console.WriteLine($"Base class of {nameof(bytes)}: {bytes.GetType().BaseType.Name}")
      Console.WriteLine()
      Console.WriteLine($"Type of {nameof(bytes)} array: {TypeName(bytes)}")
   End Sub
End Module
' The example displays the following output:
' Type of bytes array: Byte[,]
' Base class of bytes: Array
'
' Type of bytes array: Byte(,)
```

Arrays as return values and parameters

To return an array from a `Function` procedure, specify the array data type and the number of dimensions as the return type of the Function Statement. Within the function, declare a local array variable with same data type and number of dimensions. In the Return Statement, include the local array variable without parentheses.

To specify an array as a parameter to a `Sub` or `Function` procedure, define the parameter as an array with a specified data type and number of dimensions. In the call to the procedure, pass an array variable with the same data type and number of dimensions.

In the following example, the `GetNumbers` function returns an `Integer()`, a one-dimensional array of type `Integer`. The `ShowNumbers` procedure accepts an `Integer()` argument.

```
Module ReturnValuesAndParams
   Public Sub Main()
      Dim numbers As Integer() = GetNumbers()
      ShowNumbers(numbers)
   End Sub

   Private Function GetNumbers() As Integer()
      Dim numbers As Integer() = {10, 20, 30}
      Return numbers
   End Function

   Private Sub ShowNumbers(numbers As Integer())
      For index = 0 To numbers.GetUpperBound(0)
         Console.WriteLine($"{numbers(index)} ")
      Next
   End Sub
End Module
' The example displays the following output:
'   10
'   20
'   30
```

In the following example, the `GetNumbersMultiDim` function returns an `Integer(,)`, a two-dimensional array of type `Integer`. The `ShowNumbersMultiDim` procedure accepts an `Integer(,)` argument.

```
Module Example
   Public Sub Main()
      Dim numbers As Integer(,) = GetNumbersMultidim()
      ShowNumbersMultidim(numbers)
   End Sub

   Private Function GetNumbersMultidim() As Integer(,)
      Dim numbers As Integer(,) = {{1, 2}, {3, 4}, {5, 6}}
      Return numbers
   End Function

   Private Sub ShowNumbersMultidim(numbers As Integer(,))
      For index0 = 0 To numbers.GetUpperBound(0)
         For index1 = 0 To numbers.GetUpperBound(1)
            Console.Write($"{numbers(index0, index1)} ")
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'     1 2
'     3 4
'     5 6
```

Jagged arrays

Sometimes the data structure in your application is two-dimensional but not rectangular. For example, you might use an array to store data about the high temperature of each day of the month. The first dimension of the array represents the month, but the second dimension represents the number of days, and the number of days in a month is not uniform. A *jagged array*, which is also called an *array of arrays*, is designed for such scenarios. A jagged array is an array whose elements are also arrays. A jagged array and each element in a jagged array can have one or more dimensions.

The following example uses an array of months, each element of which is an array of days. The example uses a jagged array because different months have different numbers of days. The example shows how to create a jagged array, assign values to it, and retrieve and display its values.

```
Imports System.Globalization

Module JaggedArray
   Public Sub Main()
      ' Declare the jagged array of 12 elements. Each element is an array of Double.
      Dim sales(11)() As Double
      ' Set each element of the sales array to a Double array of the appropriate size.
      For month As Integer = 0 To 11
         ' The number of days in the month determines the appropriate size.
         Dim daysInMonth As Integer =
            DateTime.DaysInMonth(Year(Now), month + 1)
         sales(month) = New Double(daysInMonth - 1) {}
      Next

      ' Store values in each element.
      For month As Integer = 0 To 11
         For dayOfMonth = 0 To sales(month).GetUpperBound(0)
            sales(month)(dayOfMonth) = (month * 100) + dayOfMonth
         Next
      Next

      ' Retrieve and display the array values.
      Dim monthNames = DateTimeFormatInfo.CurrentInfo.AbbreviatedMonthNames
```

```
      ' Display the month names.
      Console.Write("    ")
      For ctr = 0 To sales.GetUpperBound(0)
         Console.Write($" {monthNames(ctr)}   ")
      Next
      Console.WriteLine()
      ' Display data for each day in each month.
      For dayInMonth = 0 To 30
         Console.Write($"{dayInMonth + 1,2}.  ")
         For monthNumber = 0 To sales.GetUpperBound(0)
            If dayInMonth > sales(monthNumber).GetUpperBound(0) Then
               Console.Write("       ")
            Else
               Console.Write($"{sales(monthNumber)(dayInMonth),-5}  ")
            End If
         Next
         Console.WriteLine()
      Next
   End Sub
End Module
' The example displays the following output:
'      Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov    Dec
'  1.  0      100    200    300    400    500    600    700    800    900    1000   1100
'  2.  1      101    201    301    401    501    601    701    801    901    1001   1101
'  3.  2      102    202    302    402    502    602    702    802    902    1002   1102
'  4.  3      103    203    303    403    503    603    703    803    903    1003   1103
'  5.  4      104    204    304    404    504    604    704    804    904    1004   1104
'  6.  5      105    205    305    405    505    605    705    805    905    1005   1105
'  7.  6      106    206    306    406    506    606    706    806    906    1006   1106
'  8.  7      107    207    307    407    507    607    707    807    907    1007   1107
'  9.  8      108    208    308    408    508    608    708    808    908    1008   1108
' 10.  9      109    209    309    409    509    609    709    809    909    1009   1109
' 11.  10     110    210    310    410    510    610    710    810    910    1010   1110
' 12.  11     111    211    311    411    511    611    711    811    911    1011   1111
' 13.  12     112    212    312    412    512    612    712    812    912    1012   1112
' 14.  13     113    213    313    413    513    613    713    813    913    1013   1113
' 15.  14     114    214    314    414    514    614    714    814    914    1014   1114
' 16.  15     115    215    315    415    515    615    715    815    915    1015   1115
' 17.  16     116    216    316    416    516    616    716    816    916    1016   1116
' 18.  17     117    217    317    417    517    617    717    817    917    1017   1117
' 19.  18     118    218    318    418    518    618    718    818    918    1018   1118
' 20.  19     119    219    319    419    519    619    719    819    919    1019   1119
' 21.  20     120    220    320    420    520    620    720    820    920    1020   1120
' 22.  21     121    221    321    421    521    621    721    821    921    1021   1121
' 23.  22     122    222    322    422    522    622    722    822    922    1022   1122
' 24.  23     123    223    323    423    523    623    723    823    923    1023   1123
' 25.  24     124    224    324    424    524    624    724    824    924    1024   1124
' 26.  25     125    225    325    425    525    625    725    825    925    1025   1125
' 27.  26     126    226    326    426    526    626    726    826    926    1026   1126
' 28.  27     127    227    327    427    527    627    727    827    927    1027   1127
' 29.  28            228    328    428    528    628    728    828    928    1028   1128
' 30.  29            229    329    429    529    629    729    829    929    1029   1129
' 31.  30            230           430           630    730           930           1130
```

The previous example assigns values to the jagged array on an element-by-element basis by using a `For...Next` loop. You can also assign values to the elements of a jagged array by using nested array literals. However, the attempt to use nested array literals (for example, `Dim valuesjagged = {{1, 2}, {2, 3, 4}}`) generates compiler error BC30568. To correct the error, enclose the inner array literals in parentheses. The parentheses force the array literal expression to be evaluated, and the resulting values are used with the outer array literal, as the following example shows.

```
Module Example
   Public Sub Main()
      Dim values1d = { 1, 2, 3 }
      Dim values2d = {{1, 2}, {2, 3}, {3, 4}}
      Dim valuesjagged = {({1, 2}), ({2, 3, 4})}
   End Sub
End Module
```

A jagged array is a one-dimensional array whose elements contain arrays. Therefore, the Array.Length property and the `Array.GetLength(0)` method return the number of elements in the one-dimensional array, and `Array.GetLength(1)` throws an IndexOutOfRangeException because a jagged array is not multidimensional. You determine the number of elements in each subarray by retrieving the value of each subarray's Array.Length property. The following example illustrates how to determine the number of elements in a jagged array.

```
Module Example
   Public Sub Main()
      Dim jagged = { ({1, 2}), ({2, 3, 4}), ({5, 6}), ({7, 8, 9, 10}) }
      Console.WriteLine($"The value of jagged.Length: {jagged.Length}.")
      Dim total = jagged.Length
      For ctr As Integer = 0 To jagged.GetUpperBound(0)
         Console.WriteLine($"Element {ctr + 1} has {jagged(ctr).Length} elements.")
         total += jagged(ctr).Length
      Next
      Console.WriteLine($"The total number of elements in the jagged array: {total}")
   End Sub
End Module
' The example displays the following output:
'     The value of jagged.Length: 4.
'     Element 1 has 2 elements.
'     Element 2 has 3 elements.
'     Element 3 has 2 elements.
'     Element 4 has 4 elements.
'     The total number of elements in the jagged array: 15
```

Zero-length arrays

Visual Basic differentiates between a uninitialized array (an array whose value is `Nothing`) and a *zero-length array* or empty array (an array that has no elements.) An uninitialized array is one that has not been dimensioned or had any values assigned to it. For example:

```
Dim arr() As String
```

A zero-length array is declared with a dimension of -1. For example:

```
Dim arrZ(-1) As String
```

You might need to create a zero-length array under the following circumstances:

- Without risking a NullReferenceException exception, your code must access members of the Array class, such as Length or Rank, or call a Visual Basic function such as UBound.
- You want to keep your code simple by not having to check for `Nothing` as a special case.

- Your code interacts with an application programming interface (API) that either requires you to pass a zero-length array to one or more procedures or returns a zero-length array from one or more procedures.

Splitting an array

In some cases, you may need to split a single array into multiple arrays. This involves identifying the point or points at which the array is to be split, and then spitting the array into two or more separate arrays.

> **NOTE**
>
> This section does not discuss splitting a single string into a string array based on some delimiter. For information on splitting a string, see the String.Split method.

The most common criteria for splitting an array are:

- The number of elements in the array. For example, you might want to split an array of more than a specified number of elements into a number of approximately equal parts. For this purpose, you can use the value returned by either the Array.Length or Array.GetLength method.
- The value of an element, which serves as a delimiter that indicates where the array should be split. You can search for a specific value by calling the Array.FindIndex and Array.FindLastIndex methods.

Once you've determined the index or indexes at which the array should be split, you can then create the individual arrays by calling the Array.Copy method.

The following example splits an array into two arrays of approximately equal size. (If the total number of array elements is odd, the first array has one more element than the second.)

```
Module Example
   Public Sub Main()
      ' Create an array of 100 elements.
      Dim arr(99) As Integer
      ' Populate the array.
      Dim rnd As new Random()
      For ctr = 0 To arr.GetUpperBound(0)
         arr(ctr) = rnd.Next()
      Next

      ' Determine how many elements should be in each array.
      Dim divisor = 2
      Dim remainder As Integer
      Dim boundary = Math.DivRem(arr.GetLength(0), divisor, remainder)

      ' Copy the array.
      Dim arr1(boundary - 1 + remainder), arr2(boundary - 1) as Integer
      Array.Copy(arr, 0, arr1, 0, boundary + remainder)
      Array.Copy(arr, boundary + remainder, arr2, 0, arr.Length - boundary)
   End Sub
End Module
```

The following example splits a string array into two arrays based on the presence of an element whose value is "zzz", which serves as the array delimiter. The new arrays do not include the element that contains the delimiter.

```
Module Example
   Public Sub Main()
      Dim rnd As New Random()

      ' Create an array of 100 elements.
      Dim arr(99) As String
      ' Populate each element with an arbitrary ASCII character.
      For ctr = 0 To arr.GetUpperBound(0)
         arr(ctr) = ChrW(Rnd.Next(&h21, &h7F))
      Next
      ' Get a random number that will represent the point to insert the delimiter.
      arr(rnd.Next(0, arr.GetUpperBound(0))) = "zzz"

      ' Find the delimiter.
      Dim location = Array.FindIndex(arr, Function(x) x = "zzz")

      ' Create the arrays.
      Dim arr1(location - 1) As String
      Dim arr2(arr.GetUpperBound(0) - location - 1) As String

      ' Populate the two arrays.
      Array.Copy(arr, 0, arr1, 0, location)
      Array.Copy(arr, location + 1, arr2, 0, arr.GetUpperBound(0) - location)
   End Sub
End Module
```

Joining arrays

You can also combine a number of arrays into a single larger array. To do this, you also use the Array.Copy method.

> **NOTE**
> This section does not discuss joining a string array into a single string. For information on joining a string array, see the String.Join method.

Before copying the elements of each array into the new array, you must first ensure that you have initialized the array so that it is large enough to accommodate the new array. You can do this in one of two ways:

- Use the `ReDim Preserve` statement to dynamically expand the array before adding new elements to it. This is the easiest technique, but it can result in performance degradation and excessive memory consumption when you are copying large arrays.
- Calculate the total number of elements needed for the new large array, then add the elements of each source array to it.

The following example uses the second approach to add four arrays with ten elements each to a single array.

```
Imports System.Collections.Generic
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim tasks As New List(Of Task(Of Integer()))
      ' Generate four arrays.
      For ctr = 0 To 3
         Dim value = ctr
         tasks.Add(Task.Run(Function()
                               Dim arr(9) As Integer
                               For ndx = 0 To arr.GetUpperBound(0)
                                  arr(ndx) = value
                               Next
                               Return arr
                            End Function))
       Next
       Task.WaitAll(tasks.ToArray())
       ' Compute the number of elements in all arrays.
       Dim elements = 0
       For Each task In tasks
          elements += task.Result.Length
       Next
       Dim newArray(elements - 1) As Integer
       Dim index = 0
       For Each task In tasks
          Dim n = task.Result.Length
          Array.Copy(task.Result, 0, newArray, index, n)
          index += n
       Next
      Console.WriteLine($"The new array has {newArray.Length} elements.")
   End Sub
End Module
' The example displays the following output:
'     The new array has 40 elements.
```

Since in this case the source arrays are all small, we can also dynamically expand the array as we add the elements of each new array to it. The following example does that.

```
Imports System.Collections.Generic
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim tasks As New List(Of Task(Of Integer()))
      ' Generate four arrays.
      For ctr = 0 To 3
         Dim value = ctr
         tasks.Add(Task.Run(Function()
                               Dim arr(9) As Integer
                               For ndx = 0 To arr.GetUpperBound(0)
                                  arr(ndx) = value
                               Next
                               Return arr
                            End Function))
       Next
       Task.WaitAll(tasks.ToArray())

       ' Dimension the target array and copy each element of each source array to it.
       Dim newArray() As Integer = {}
       ' Define the next position to copy to in newArray.
       Dim index = 0
       For Each task In tasks
          Dim n = Task.Result.Length
          ReDim Preserve newArray(newArray.GetUpperBound(0) + n)
          Array.Copy(task.Result, 0, newArray, index, n)
          index += n
       Next
      Console.WriteLine($"The new array has {newArray.Length} elements.")
   End Sub
End Module
' The example displays the following output:
'     The new array has 40 elements.
```

Collections as an alternative to arrays

Arrays are most useful for creating and working with a fixed number of strongly typed objects. Collections provide a more flexible way to work with groups of objects. Unlike arrays, which require that you explicitly change the size of an array with the `ReDim` Statement, collections grow and shrink dynamically as the needs of an application change.

When you use `ReDim` to redimension an array, Visual Basic creates a new array and releases the previous one. This takes execution time. Therefore, if the number of items you are working with changes frequently, or you cannot predict the maximum number of items you need, you'll usually obtain better performance by using a collection.

For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it.

For more information about collections, see Collections.

Related topics

TERM	DEFINITION
Array Dimensions in Visual Basic	Explains rank and dimensions in arrays.
How to: Initialize an Array Variable in Visual Basic	Describes how to populate arrays with initial values.
How to: Sort An Array in Visual Basic	Shows how to sort the elements of an array alphabetically.
How to: Assign One Array to Another Array	Describes the rules and steps for assigning an array to another array variable.
Troubleshooting Arrays	Discusses some common problems that arise when working with arrays.

See also

- System.Array
- Dim Statement
- ReDim Statement

Collection Initializers (Visual Basic)

3/8/2019 • 5 minutes to read • Edit Online

Collection initializers provide a shortened syntax that enables you to create a collection and populate it with an initial set of values. Collection initializers are useful when you are creating a collection from a set of known values, for example, a list of menu options or categories, an initial set of numeric values, a static list of strings such as day or month names, or geographic locations such as a list of states that is used for validation.

For more information about collections, see Collections.

You identify a collection initializer by using the `From` keyword followed by braces (`{}`). This is similar to the array literal syntax that is described in Arrays. The following examples show various ways to use collection initializers to create collections.

```
' Create an array of type String().
Dim winterMonths = {"December", "January", "February"}

' Create an array of type Integer()
Dim numbers = {1, 2, 3, 4, 5}

' Create a list of menu options. (Requires an extension method
' named Add for List(Of MenuOption)
Dim menuOptions = New List(Of MenuOption) From {{1, "Home"},
                                               {2, "Products"},
                                               {3, "News"},
                                               {4, "Contact Us"}}
```

NOTE

C# also provides collection initializers. C# collection initializers provide the same functionality as Visual Basic collection initializers. For more information about C# collection initializers, see Object and Collection Initializers.

Syntax

A collection initializer consists of a list of comma-separated values that are enclosed in braces (`{}`), preceded by the `From` keyword, as shown in the following code.

```
Dim names As New List(Of String) From {"Christa", "Brian", "Tim"}
```

When you create a collection, such as a List<T> or a Dictionary<TKey,TValue>, you must supply the collection type before the collection initializer, as shown in the following code.

```
Public Class AppMenu
    Public Property Items As List(Of String) =
        New List(Of String) From {"Home", "About", "Contact"}
End Class
```

NOTE

You cannot combine both a collection initializer and an object initializer to initialize the same collection object. You can use object initializers to initialize objects in a collection initializer.

Creating a Collection by Using a Collection Initializer

When you create a collection by using a collection initializer, each value that is supplied in the collection initializer is passed to the appropriate `Add` method of the collection. For example, if you create a List<T> by using a collection initializer, each string value in the collection initializer is passed to the Add method. If you want to create a collection by using a collection initializer, the specified type must be valid collection type. Examples of valid collection types include classes that implement the IEnumerable<T> interface or inherit the CollectionBase class. The specified type must also expose an `Add` method that meets the following criteria.

- The `Add` method must be available from the scope in which the collection initializer is being called. The `Add` method does not have to be public if you are using the collection initializer in a scenario where non-public methods of the collection can be accessed.
- The `Add` method must be an instance member or `Shared` member of the collection class, or an extension method.
- An `Add` method must exist that can be matched, based on overload resolution rules, to the types that are supplied in the collection initializer.

For example, the following code example shows how to create a `List(Of Customer)` collection by using a collection initializer. When the code is run, each `Customer` object is passed to the `Add(Customer)` method of the generic list.

```
Dim customers = New List(Of Customer) From
    {
        New Customer("City Power & Light", "http://www.cpandl.com/"),
        New Customer("Wide World Importers", "http://www.wideworldimporters.com/"),
        New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/")
    }
```

The following code example shows equivalent code that does not use a collection initializer.

```
Dim customers = New List(Of Customer)
customers.Add(New Customer("City Power & Light", "http://www.cpandl.com/"))
customers.Add(New Customer("Wide World Importers", "http://www.wideworldimporters.com/"))
customers.Add(New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/"))
```

If the collection has an `Add` method that has parameters that match the constructor for the `Customer` object, you could nest parameter values for the `Add` method within collection initializers, as discussed in the next section. If the collection does not have such an `Add` method, you can create one as an extension method. For an example of how to create an `Add` method as an extension method for a collection, see How to: Create an Add Extension Method Used by a Collection Initializer. For an example of how to create a custom collection that can be used with a collection initializer, see How to: Create a Collection Used by a Collection Initializer.

Nesting Collection Initializers

You can nest values within a collection initializer to identify a specific overload of an `Add` method for the collection that is being created. The values passed to the `Add` method must be separated by commas and enclosed in braces (`{}`), like you would do in an array literal or collection initializer.

When you create a collection by using nested values, each element of the nested value list is passed as an argument to the `Add` method that matches the element types. For example, the following code example creates a Dictionary<TKey,TValue> in which the keys are of type `Integer` and the values are of type `String`. Each of the nested value lists is matched to the Add method for the `Dictionary`.

```
Dim days = New Dictionary(Of Integer, String) From
    {{0, "Sunday"}, {1, "Monday"}}
```

The previous code example is equivalent to the following code.

```
Dim days = New Dictionary(Of Integer, String)
days.Add(0, "Sunday")
days.Add(1, "Monday")
```

Only nested value lists from the first level of nesting are sent to the `Add` method for the collection type. Deeper levels of nesting are treated as array literals and the nested value lists are not matched to the `Add` method of any collection.

Related Topics

TITLE	DESCRIPTION
How to: Create an Add Extension Method Used by a Collection Initializer	Shows how to create an extension method called `Add` that can be used to populate a collection with values from a collection initializer.
How to: Create a Collection Used by a Collection Initializer	Shows how to enable use of a collection initializer by including an `Add` method in a collection class that implements `IEnumerable`.

See also

- Collections
- Arrays
- Object Initializers: Named and Anonymous Types
- New Operator
- Auto-Implemented Properties
- How to: Initialize an Array Variable in Visual Basic
- Local Type Inference
- Anonymous Types
- Introduction to LINQ in Visual Basic
- How to: Create a List of Items

Constants and Enumerations in Visual Basic

5/4/2018 • 2 minutes to read • Edit Online

Constants are a way to use meaningful names in place of a value that does not change. Constants store values that, as the name implies, remain constant throughout the execution of an application. You can use constants to provide meaningful names, instead of numbers, making your code more readable.

Enumerations provide a convenient way to work with sets of related constants, and to associate constant values with names. For example, you can declare an enumeration for a set of integer constants associated with the days of the week, and then use the names of the days rather than their integer values in your code.

In This Section

TERM	DEFINITION
Constants Overview	Topics in this section describe constants and their uses.
Enumerations Overview	Topics in this section describe enumerations and their uses.

Related Sections

TERM	DEFINITION
Const Statement	Describes the `Const` statement, which is used to declare constants.
Enum Statement	Describes the `Enum` statement, which is used to create enumerations.
Option Explicit Statement	Describes the `Option Explicit` statement, which is used at module level to force explicit declaration of all variables in that module.
Option Infer Statement	Describes the `Option Infer` statement, which enables the use of local type inference in declaring variables.
Option Strict Statement	Describes the `Option Strict` statement, which restricts implicit data type conversions to only widening conversions, disallows late binding, and disallows implicit typing that results in an `Object` type.

Control Flow in Visual Basic

5/4/2018 • 2 minutes to read • Edit Online

Left unregulated, a program proceeds through its statements from beginning to end. Some very simple programs can be written with only this unidirectional flow. However, much of the power and utility of any programming language comes from the ability to change execution order with control statements and loops.

Control structures allow you to regulate the flow of your program's execution. Using control structures, you can write Visual Basic code that makes decisions or that repeats actions. Other control structures let you guarantee disposal of a resource or run a series of statements on the same object reference.

In This Section

Decision Structures
Describes control structures used for branching.

Loop Structures
Discusses control structures used to repeat processes.

Other Control Structures
Describes control structures used for resource disposal and object access.

Nested Control Structures
Covers control structures inside other control structures.

Related Sections

Control Flow Summary
Provides links to language reference pages on this subject.

Data Types in Visual Basic

4/2/2019 • 2 minutes to read • Edit Online

The *data type* of a programming element refers to what kind of data it can hold and how it stores that data. Data types apply to all values that can be stored in computer memory or participate in the evaluation of an expression. Every variable, literal, constant, enumeration, property, procedure parameter, procedure argument, and procedure return value has a data type.

Declared Data Types

You define a programming element with a declaration statement, and you specify its data type with the `As` clause. The following table shows the statements you use to declare various elements.

PROGRAMMING ELEMENT	DATA TYPE DECLARATION

Variable	In a Dim Statement
	`Dim amount As Double`
	`Static yourName As String`
	`Public billsPaid As Decimal = 0`

Literal	With a literal type character; see "Literal Type Characters" in Type Characters `Dim searchChar As Char = "." C`
Constant	In a Const Statement `Const modulus As Single = 4.17825F`
Enumeration	In an Enum Statement `Public Enum colors`
Property	In a Property Statement `Property region() As String`
Procedure parameter	In a Sub Statement, Function Statement, or Operator Statement `Sub addSale(ByVal amount As Double)`
Procedure argument	In the calling code; each argument is a programming element that has already been declared, or an expression containing declared elements `subString = Left( inputString , 5 )`

PROGRAMMING ELEMENT	DATA TYPE DECLARATION
Procedure return value	In a Function Statement or Operator Statement `Function convert(ByVal b As Byte)` `As String`

For a list of Visual Basic data types, see Data Types.

See also

- Type Characters
- Elementary Data Types
- Composite Data Types
- Generic Types in Visual Basic
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Structures
- Tuples
- Troubleshooting Data Types
- Data Types
- Efficient Use of Data Types

Type characters (Visual Basic)

1/23/2019 • 3 minutes to read • Edit Online

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements with a *type character*. The type character must immediately follow the element, with no intervening characters of any kind.

The type character is not part of the name of the element. An element defined with a type character can be referenced without the type character.

Identifier type characters

Visual Basic supplies a set of *identifier type characters* that you can use in a declaration to specify the data type of a variable or constant. The following table shows the available identifier type characters with examples of usage.

IDENTIFIER TYPE CHARACTER	DATA TYPE	EXAMPLE
`%`	`Integer`	`Dim L%`
`&`	`Long`	`Dim M&`
`@`	`Decimal`	`Const W@ = 37.5`
`!`	`Single`	`Dim Q!`
`#`	`Double`	`Dim X#`
`$`	`String`	`Dim V$ = "Secret"`

No identifier type characters exist for the `Boolean`, `Byte`, `Char`, `Date`, `Object`, `SByte`, `Short`, `UInteger`, `ULong`, or `UShort` data types, or for any composite data types such as arrays or structures.

In some cases, you can append the `$` character to a Visual Basic function, for example `Left$` instead of `Left`, to obtain a returned value of type `String`.

In all cases, the identifier type character must immediately follow the identifier name.

Literal type characters

A *literal* is a textual representation of a particular value of a data type.

Default literal types

The form of a literal as it appears in your code ordinarily determines its data type. The following table shows these default types.

TEXTUAL FORM OF LITERAL	DEFAULT DATA TYPE	EXAMPLE
Numeric, no fractional part	`Integer`	`2147483647`

TEXTUAL FORM OF LITERAL	DEFAULT DATA TYPE	EXAMPLE
Numeric, no fractional part, too large for `Integer`	`Long`	`2147483648`
Numeric, fractional part	`Double`	`1.2`
Enclosed in double quotation marks	`String`	`"A"`
Enclosed within number signs	`Date`	`#5/17/1993 9:32 AM#`

Forced literal types

Visual Basic supplies a set of *literal type characters*, which you can use to force a literal to assume a data type other than the one its form indicates. You do this by appending the character to the end of the literal. The following table shows the available literal type characters with examples of usage.

LITERAL TYPE CHARACTER	DATA TYPE	EXAMPLE
`S`	`Short`	`I = 347S`
`I`	`Integer`	`J = 347I`
`L`	`Long`	`K = 347L`
`D`	`Decimal`	`X = 347D`
`F`	`Single`	`Y = 347F`
`R`	`Double`	`Z = 347R`
`US`	`UShort`	`L = 347US`
`UI`	`UInteger`	`M = 347UI`
`UL`	`ULong`	`N = 347UL`
`C`	`Char`	`Q = "."C`

No literal type characters exist for the `Boolean`, `Byte`, `Date`, `Object`, `SByte`, or `String` data types, or for any composite data types such as arrays or structures.

Literals can also use the identifier type characters (`%`, `&`, `@`, `!`, `#`, `$`), as can variables, constants, and expressions. However, the literal type characters (`S`, `I`, `L`, `D`, `F`, `R`, `C`) can be used only with literals.

In all cases, the literal type character must immediately follow the literal value.

Hexadecimal, binary, and octal literals

The compiler normally interprets an integer literal to be in the decimal (base 10) number system. You can also define an integer literal as a hexadecimal (base 16) number with the `&H` prefix, as a binary (base 2) number with the `&B` prefix, and as an octal (base 8) number with the `&O` prefix. The digits that follow the prefix must be appropriate for the number system. The following table illustrates this.

NUMBER BASE	PREFIX	VALID DIGIT VALUES	EXAMPLE
Hexadecimal (base 16)	`&H`	0-9 and A-F	`&HFFFF`
Binary (base 2)	`&B`	0-1	`&B01111100`
Octal (base 8)	`&O`	0-7	`&O77`

Starting in Visual Basic 2017, you can use the underscore character (`_`) as a group separator to enhance the readability of an integral literal. The following example uses the `_` character to group a binary literal into 8-bit groups:

```
Dim number As Integer = &B00100010_11000101_11001111_11001101
```

You can follow a prefixed literal with a literal type character. The following example shows this.

```
Dim counter As Short = &H8000S
Dim flags As UShort = &H8000US
```

In the previous example, `counter` has the decimal value of -32768, and `flags` has the decimal value of +32768.

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

See also

- Data Types
- Elementary Data Types
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Troubleshooting Data Types
- Variable Declaration
- Data Types

Elementary Data Types (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Visual Basic supplies a set of predefined data types, which you can use for many of your programming elements. This section describes these types and how to use them.

> **NOTE**
>
> Every elementary data type in Visual Basic is supported by a structure or a class that is in the System namespace. The compiler uses each data type keyword as an alias for the underlying structure or class. For example, declaring a variable by using the reserved word `Byte` is the same as declaring it by using the fully qualified structure name System.Byte.

In This Section

Numeric Data Types
Describes the integral and non-integral numeric types.

Character Data Types
Describes the `Char` and `String` types.

Miscellaneous Data Types
Describes the `Boolean`, `Date`, and `Object` types.

Related Sections

Data Types
Introduces the Visual Basic data types and describes how to use them.

Data Types
Provides an overview of the elementary data types supplied by Visual Basic.

Numeric Data Types (Visual Basic)

4/2/2019 • 4 minutes to read • Edit Online

Visual Basic supplies several *numeric data types* for handling numbers in various representations. *Integral* types represent only whole numbers (positive, negative, and zero), and *nonintegral* types represent numbers with both integer and fractional parts.

For a table showing a side-by-side comparison of the Visual Basic data types, see Data Types.

Integral Numeric Types

Integral data types are those that represent only numbers without fractional parts.

The *signed* integral data types are SByte Data Type (8-bit), Short Data Type (16-bit), Integer Data Type (32-bit), and Long Data Type (64-bit). If a variable always stores integers rather than fractional numbers, declare it as one of these types.

The *unsigned* integral types are Byte Data Type (8-bit), UShort Data Type (16-bit), UInteger Data Type (32-bit), and ULong Data Type (64-bit). If a variable contains binary data, or data of unknown nature, declare it as one of these types.

Performance

Arithmetic operations are faster with integral types than with other data types. They are fastest with the `Integer` and `UInteger` types in Visual Basic.

Large Integers

If you need to hold an integer larger than the `Integer` data type can hold, you can use the `Long` data type instead. `Long` variables can hold numbers from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. Operations with `Long` are slightly slower than with `Integer`.

If you need even larger values, you can use the Decimal Data Type. You can hold numbers from -79,228,162,514,264,337,593,543,950,335 through 79,228,162,514,264,337,593,543,950,335 in a `Decimal` variable if you do not use any decimal places. However, operations with `Decimal` numbers are considerably slower than with any other numeric data type.

Small Integers

If you do not need the full range of the `Integer` data type, you can use the `Short` data type, which can hold integers from -32,768 through 32,767. For the smallest integer range, the `SByte` data type holds integers from -128 through 127. If you have a very large number of variables that hold small integers, the common language runtime can sometimes store your `Short` and `SByte` variables more efficiently and save memory consumption. However, operations with `Short` and `SByte` are somewhat slower than with `Integer`.

Unsigned Integers

If you know that your variable never needs to hold a negative number, you can use the *unsigned types* `Byte`, `UShort`, `UInteger`, and `ULong`. Each of these data types can hold a positive integer twice as large as its corresponding signed type (`SByte`, `Short`, `Integer`, and `Long`). In terms of performance, each unsigned type is exactly as efficient as its corresponding signed type. In particular, `UInteger` shares with `Integer` the distinction of being the most efficient of all the elementary numeric data types.

Nonintegral Numeric Types

Nonintegral data types are those that represent numbers with both integer and fractional parts.

The nonintegral numeric data types are `Decimal` (128-bit fixed point), Single Data Type (32-bit floating point), and Double Data Type (64-bit floating point). They are all signed types. If a variable can contain a fraction, declare it as one of these types.

`Decimal` is not a floating-point data type. `Decimal` numbers have a binary integer value and an integer scaling factor that specifies what portion of the value is a decimal fraction.

You can use `Decimal` variables for money values. The advantage is the precision of the values. The `Double` data type is faster and requires less memory, but it is subject to rounding errors. The `Decimal` data type retains complete accuracy to 28 decimal places.

Floating-point (`Single` and `Double`) numbers have larger ranges than `Decimal` numbers but can be subject to rounding errors. Floating-point types support fewer significant digits than `Decimal` but can represent values of greater magnitude.

Nonintegral number values can be expressed as mmmEeee, in which mmm is the *mantissa* (the significant digits) and eee is the *exponent* (a power of 10). The highest positive values of the nonintegral types are 7.9228162514264337593543950335E+28 for `Decimal`, 3.4028235E+38 for `Single`, and 1.79769313486231570E+308 for `Double`.

Performance

`Double` is the most efficient of the fractional data types, because the processors on current platforms perform floating-point operations in double precision. However, operations with `Double` are not as fast as with the integral types such as `Integer`.

Small Magnitudes

For numbers with the smallest possible magnitude (closest to 0), `Double` variables can hold numbers as small as -4.94065645841246544E-324 for negative values and 4.94065645841246544E-324 for positive values.

Small Fractional Numbers

If you do not need the full range of the `Double` data type, you can use the `Single` data type, which can hold floating-point numbers from -3.4028235E+38 through 3.4028235E+38. The smallest magnitudes for `Single` variables are -1.401298E-45 for negative values and 1.401298E-45 for positive values. If you have a very large number of variables that hold small floating-point numbers, the common language runtime can sometimes store your `Single` variables more efficiently and save memory consumption.

See also

- Elementary Data Types
- Character Data Types
- Miscellaneous Data Types
- Troubleshooting Data Types
- How to: Call a Windows Function that Takes Unsigned Types

Character Data Types (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Visual Basic provides *character data types* to deal with printable and displayable characters. While they both deal with Unicode characters, `Char` holds a single character whereas `String` contains an indefinite number of characters.

For a table that displays a side-by-side comparison of the Visual Basic data types, see Data Types.

Char Type

The `Char` data type is a single two-byte (16-bit) Unicode character. If a variable always stores exactly one character, declare it as `Char`. For example:

```
' Initialize the prefix variable to the character 'a'.
Dim prefix As Char = "a"
```

Each possible value in a `Char` or `String` variable is a *code point*, or character code, in the Unicode character set. Unicode characters include the basic ASCII character set, various other alphabet letters, accents, currency symbols, fractions, diacritics, and mathematical and technical symbols.

> **NOTE**
> The Unicode character set reserves the code points D800 through DFFF (55296 through 55551 decimal) for *surrogate pairs*, which require two 16-bit values to represent a single code point. A `Char` variable cannot hold a surrogate pair, and a `String` uses two positions to hold such a pair.

For more information, see Char Data Type.

String Type

The `String` data type is a sequence of zero or more two-byte (16-bit) Unicode characters. If a variable can contain an indefinite number of characters, declare it as `String`. For example:

```
' Initialize the name variable to "Monday".
Dim name As String = "Monday"
```

For more information, see String Data Type.

See also

- Elementary Data Types
- Composite Data Types
- Generic Types in Visual Basic
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Troubleshooting Data Types
- Type Characters

Miscellaneous Data Types (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic supplies several data types that are not oriented toward numbers or characters. Instead, they deal with specialized data such as yes/no values, date/time values, and object addresses.

For a table showing a side-by-side comparison of the Visual Basic data types, see Data Types.

Boolean Type

The Boolean Data Type is an unsigned value that is interpreted as either `True` or `False`. Its data width depends on the implementing platform. If a variable can contain only two-state values such as true/false, yes/no, or on/off, declare it as `Boolean`.

Date Type

The Date Data Type is a 64-bit value that holds both date and time information. Each increment represents 100 nanoseconds of elapsed time since the beginning (12:00 AM) of January 1 of the year 1 in the Gregorian calendar. If a variable can contain a date value, a time value, or both, declare it as `Date`.

Object Type

The Object Data Type is a 32-bit address that points to an object instance within your application or in some other application. An `Object` variable can refer to any object your application recognizes, or to data of any data type. This includes both *value types*, such as `Integer`, `Boolean`, and structure instances, and *reference types*, which are instances of objects created from classes such as `String` and Form, and array instances.

If a variable stores a pointer to an instance of a class that you do not know at compile time, or if it can point to data of various data types, declare it as `Object`.

The advantage of the `Object` data type is that you can use it to store data of any data type. The disadvantage is that you incur extra operations that take more execution time and make your application perform slower. If you use an `Object` variable for value types, you incur *boxing* and *unboxing*. If you use it for reference types, you incur *late binding*.

See also

- Type Characters
- Elementary Data Types
- Numeric Data Types
- Character Data Types
- Troubleshooting Data Types
- Early and Late Binding

Composite Data Types (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

In addition to the elementary data types Visual Basic supplies, you can also assemble items of different types to create *composite data types* such as structures, arrays, and classes. You can build composite data types from elementary types and from other composite types. For example, you can define an array of structure elements, or a structure with array members.

Data Types

A composite type is different from the data type of any of its components. For example, an array of `Integer` elements is not of the `Integer` data type.

An array data type is normally represented using the element type, parentheses, and commas as necessary. For example, a one-dimensional array of `String` elements is represented as `String()`, and a two-dimensional array of `Boolean` elements is represented as `Boolean(,)`.

Structure Types

There is no single data type comprising all structures. Instead, each definition of a structure represents a unique data type, even if two structures define identical elements in the same order. However, if you create two or more instances of the same structure, Visual Basic considers them to be of the same data type.

Tuples

A tuple is a lightweight structure that contains two or more fields whose types are predefined. Tuples are supported starting with Visual Basic 2017. Tuples are most commonly used to return multiple values from a single method call without having to pass arguments by reference or packaging the returned fields in a more heavy-weight class or structure. See the Tuples topic for more information on tuples.

Array Types

There is no single data type comprising all arrays. The data type of a particular instance of an array is determined by the following:

- The fact of being an array
- The rank (number of dimensions) of the array
- The element type of the array

In particular, the length of a given dimension is not part of the instance's data type. The following example illustrates this.

```
Dim arrayA( ) As Byte = New Byte(12) {}
Dim arrayB( ) As Byte = New Byte(100) {}
Dim arrayC( ) As Short = New Short(100) {}
Dim arrayD( , ) As Short
Dim arrayE( , ) As Short = New Short(4, 10) {}
```

In the preceding example, array variables `arrayA` and `arrayB` are considered to be of the same data type — `Byte()` — even though they are initialized to different lengths. Variables `arrayB` and `arrayC` are not of the same

type because their element types are different. Variables `arrayC` and `arrayD` are not of the same type because their ranks are different. Variables `arrayD` and `arrayE` have the same type — `Short(,)` — because their ranks and element types are the same, even though `arrayD` is not yet initialized.

For more information on arrays, see Arrays.

Class Types

There is no single data type comprising all classes. Although one class can inherit from another class, each is a separate data type. Multiple instances of the same class are of the same data type. If you assign one class instance variable to another, not only do they have the same data type, they point to the same class instance in memory.

For more information on classes, see Objects and Classes.

See also

- Data Types
- Elementary Data Types
- Generic Types in Visual Basic
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Structures
- Troubleshooting Data Types
- How to: Hold More Than One Value in a Variable

How to: Hold More Than One Value in a Variable (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

A variable holds more than one value if you declare it to be of a *composite data type*.

Composite Data Types include structures, arrays, and classes. A variable of a composite data type can hold a combination of elementary data types and other composite types. Structures and classes can hold code as well as data.

To hold more than one value in a variable

1. Determine what composite data type you want to use for your variable.
2. If the composite data type is not already defined, define it so that your variable can use it.
   - Define a structure with a Structure Statement.
   - Define an array with a Dim Statement.
   - Define a class with a Class Statement.
3. Declare your variable with a `Dim` statement.
4. Follow the variable name with an `As` clause.
5. Follow the `As` keyword with the name of the appropriate composite data type.

See also

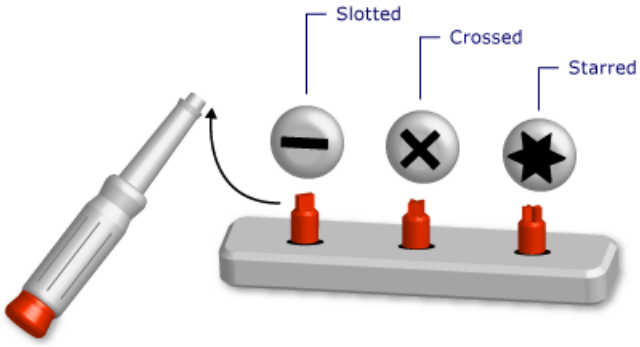
- Data Types
- Type Characters
- Composite Data Types
- Structures
- Arrays
- Objects and Classes
- Value Types and Reference Types

Generic Types in Visual Basic (Visual Basic)

5/14/2019 • 7 minutes to read • Edit Online

A *generic type* is a single programming element that adapts to perform the same functionality for a variety of data types. When you define a generic class or procedure, you do not have to define a separate version for each data type for which you might want to perform that functionality.

An analogy is a screwdriver set with removable heads. You inspect the screw you need to turn and select the correct head for that screw (slotted, crossed, starred). Once you insert the correct head in the screwdriver handle, you perform the exact same function with the screwdriver, namely turning the screw.



When you define a generic type, you parameterize it with one or more data types. This allows the using code to tailor the data types to its requirements. Your code can declare several different programming elements from the generic element, each one acting on a different set of data types. But the declared elements all perform the identical logic, no matter what data types they are using.

For example, you might want to create and use a queue class that operates on a specific data type such as `String`. You can declare such a class from System.Collections.Generic.Queue<T>, as the following example shows.

```
Public stringQ As New System.Collections.Generic.Queue(Of String)
```

You can now use `stringQ` to work exclusively with `String` values. Because `stringQ` is specific for `String` instead of being generalized for `Object` values, you do not have late binding or type conversion. This saves execution time and reduces run-time errors.

For more information on using a generic type, see How to: Use a Generic Class.

Example of a Generic Class

The following example shows a skeleton definition of a generic class.

```
Public Class classHolder(Of t)
    Public Sub processNewItem(ByVal newItem As t)
        Dim tempItem As t
        ' Insert code that processes an item of data type t.
    End Sub
End Class
```

In the preceding skeleton, `t` is a *type parameter*, that is, a placeholder for a data type that you supply when you declare the class. Elsewhere in your code, you can declare various versions of `classHolder` by supplying

various data types for `t`. The following example shows two such declarations.

```
Public integerClass As New classHolder(Of Integer)
Friend stringClass As New classHolder(Of String)
```

The preceding statements declare *constructed classes*, in which a specific type replaces the type parameter. This replacement is propagated throughout the code within the constructed class. The following example shows what the `processNewItem` procedure looks like in `integerClass`.

```
Public Sub processNewItem(ByVal newItem As Integer)
    Dim tempItem As Integer
    ' Inserted code now processes an Integer item.
End Sub
```

For a more complete example, see How to: Define a Class That Can Provide Identical Functionality on Different Data Types.

Eligible Programming Elements

You can define and use generic classes, structures, interfaces, procedures, and delegates. Note that the .NET Framework defines several generic classes, structures, and interfaces that represent commonly used generic elements. The System.Collections.Generic namespace provides dictionaries, lists, queues, and stacks. Before defining your own generic element, see if it is already available in System.Collections.Generic.

Procedures are not types, but you can define and use generic procedures. See Generic Procedures in Visual Basic.

Advantages of Generic Types

A generic type serves as a basis for declaring several different programming elements, each of which operates on a specific data type. The alternatives to a generic type are:

1. A single type operating on the `Object` data type.
2. A set of *type-specific* versions of the type, each version individually coded and operating on one specific data type such as `String`, `Integer`, or a user-defined type such as `customer`.

A generic type has the following advantages over these alternatives:

- **Type Safety.** Generic types enforce compile-time type checking. Types based on `Object` accept any data type, and you must write code to check whether an input data type is acceptable. With generic types, the compiler can catch type mismatches before run time.
- **Performance.** Generic types do not have to *box* and *unbox* data, because each one is specialized for one data type. Operations based on `Object` must box input data types to convert them to `Object` and unbox data destined for output. Boxing and unboxing reduce performance.

  Types based on `Object` are also late-bound, which means that accessing their members requires extra code at run time. This also reduces performance.
- **Code Consolidation.** The code in a generic type has to be defined only once. A set of type-specific versions of a type must replicate the same code in each version, with the only difference being the specific data type for that version. With generic types, the type-specific versions are all generated from the original generic type.
- **Code Reuse.** Code that does not depend on a particular data type can be reused with various data types

if it is generic. You can often reuse it even with a data type that you did not originally predict.

- **IDE Support.** When you use a constructed type declared from a generic type, the integrated development environment (IDE) can give you more support while you are developing your code. For example, IntelliSense can show you the type-specific options for an argument to a constructor or method.

- **Generic Algorithms.** Abstract algorithms that are type-independent are good candidates for generic types. For example, a generic procedure that sorts items using the IComparable interface can be used with any data type that implements IComparable.

Constraints

Although the code in a generic type definition should be as type-independent as possible, you might need to require a certain capability of any data type supplied to your generic type. For example, if you want to compare two items for the purpose of sorting or collating, their data type must implement the IComparable interface. You can enforce this requirement by adding a *constraint* to the type parameter.

Example of a Constraint

The following example shows a skeleton definition of a class with a constraint that requires the type argument to implement IComparable.

```
Public Class itemManager(Of t As IComparable)
    ' Insert code that defines class members.
End Class
```

If subsequent code attempts to construct a class from `itemManager` supplying a type that does not implement IComparable, the compiler signals an error.

Types of Constraints

Your constraint can specify the following requirements in any combination:

- The type argument must implement one or more interfaces
- The type argument must be of the type of, or inherit from, at most one class
- The type argument must expose a parameterless constructor accessible to the code that creates objects from it
- The type argument must be a *reference type*, or it must be a *value type*

If you need to impose more than one requirement, you use a comma-separated *constraint list* inside braces (`{ }`). To require an accessible constructor, you include the New Operator keyword in the list. To require a reference type, you include the `Class` keyword; to require a value type, you include the `Structure` keyword.

For more information on constraints, see Type List.

Example of Multiple Constraints

The following example shows a skeleton definition of a generic class with a constraint list on the type parameter. In the code that creates an instance of this class, the type argument must implement both the IComparable and IDisposable interfaces, be a reference type, and expose an accessible parameterless constructor.

```
Public Class thisClass(Of t As {IComparable, IDisposable, Class, New})
    ' Insert code that defines class members.
End Class
```

Important Terms

Generic types introduce and use the following terms:

- *Generic Type*. A definition of a class, structure, interface, procedure, or delegate for which you supply at least one data type when you declare it.
- *Type Parameter*. In a generic type definition, a placeholder for a data type you supply when you declare the type.
- *Type Argument*. A specific data type that replaces a type parameter when you declare a constructed type from a generic type.
- *Constraint*. A condition on a type parameter that restricts the type argument you can supply for it. A constraint can require that the type argument must implement a particular interface, be or inherit from a particular class, have an accessible parameterless constructor, or be a reference type or a value type. You can combine these constraints, but you can specify at most one class.
- *Constructed Type*. A class, structure, interface, procedure, or delegate declared from a generic type by supplying type arguments for its type parameters.

See also

- Data Types
- Type Characters
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Troubleshooting Data Types
- Data Types
- Of
- As
- Object Data Type
- Covariance and Contravariance
- Iterators

How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic)

7/10/2019 • 4 minutes to read • Edit Online

You can define a class from which you can create objects that provide identical functionality on different data types. To do this, you specify one or more *type parameters* in the definition. The class can then serve as a template for objects that use various data types. A class defined in this way is called a *generic class*.

The advantage of defining a generic class is that you define it just once, and your code can use it to create many objects that use a wide variety of data types. This results in better performance than defining the class with the `Object` type.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

To define a class with a type parameter

1. Define the class in the normal way.
2. Add `(Of` *typeparameter* `)` immediately after the class name to specify a type parameter.
3. If you have more than one type parameter, make a comma-separated list inside the parentheses. Do not repeat the `Of` keyword.
4. If your code performs operations on a type parameter other than simple assignment, follow that type parameter with an `As` clause to add one or more *constraints*. A constraint guarantees that the type supplied for that type parameter satisfies a requirement such as the following:

   - Supports an operation, such as `>`, that your code performs
   - Supports a member, such as a method, that your code accesses
   - Exposes a parameterless constructor

   If you do not specify any constraints, the only operations and members your code can use are those supported by the Object Data Type. For more information, see Type List.
5. Identify every class member that is to be declared with a supplied type, and declare it `As` `typeparameter`. This applies to internal storage, procedure parameters, and return values.
6. Be sure your code uses only operations and methods that are supported by any data type it can supply to `itemType`.

   The following example defines a class that manages a very simple list. It holds the list in the internal array `items`, and the using code can declare the data type of the list elements. A parameterized constructor allows the using code to set the upper bound of `items`, and the parameterless constructor sets this to 9 (for a total of 10 items).

```
Public Class simpleList(Of itemType)
  Private items() As itemType
  Private top As Integer
  Private nextp As Integer
  Public Sub New()
    Me.New(9)
  End Sub
  Public Sub New(ByVal t As Integer)
    MyBase.New()
    items = New itemType(t) {}
    top = t
    nextp = 0
  End Sub
  Public Sub add(ByVal i As itemType)
    insert(i, nextp)
  End Sub
  Public Sub insert(ByVal i As itemType, ByVal p As Integer)
    If p > nextp OrElse p < 0 Then
      Throw New System.ArgumentOutOfRangeException("p",
        " less than 0 or beyond next available list position")
    ElseIf nextp > top Then
      Throw New System.ArgumentException("No room to insert at ",
        "p")
    ElseIf p < nextp Then
      For j As Integer = nextp To p + 1 Step -1
        items(j) = items(j - 1)
      Next j
    End If
    items(p) = i
    nextp += 1
  End Sub
  Public Sub remove(ByVal p As Integer)
    If p >= nextp OrElse p < 0 Then
        Throw New System.ArgumentOutOfRangeException("p",
            " less than 0 or beyond last list item")
    ElseIf nextp = 0 Then
        Throw New System.ArgumentException("List empty; cannot remove ",
            "p")
    ElseIf p < nextp - 1 Then
        For j As Integer = p To nextp - 2
            items(j) = items(j + 1)
        Next j
    End If
    nextp -= 1
  End Sub
  Public ReadOnly Property listLength() As Integer
    Get
      Return nextp
    End Get
  End Property
  Public ReadOnly Property listItem(ByVal p As Integer) As itemType
    Get
      If p >= nextp OrElse p < 0 Then
        Throw New System.ArgumentOutOfRangeException("p",
          " less than 0 or beyond last list item")
        End If
      Return items(p)
    End Get
  End Property
End Class
```

You can declare a class from `simpleList` to hold a list of `Integer` values, another class to hold a list of `String` values, and another to hold `Date` values. Except for the data type of the list members, objects created from all these classes behave identically.

The type argument that the using code supplies to `itemType` can be an intrinsic type such as `Boolean` or `Double`, a structure, an enumeration, or any type of class, including one that your application defines.

You can test the class `simpleList` with the following code.

```
Public Sub useSimpleList()
  Dim iList As New simpleList(Of Integer)(2)
  Dim sList As New simpleList(Of String)(3)
  Dim dList As New simpleList(Of Date)(2)
  iList.add(10)
  iList.add(20)
  iList.add(30)
  sList.add("First")
  sList.add("extra")
  sList.add("Second")
  sList.add("Third")
  sList.remove(1)
  dList.add(#1/1/2003#)
  dList.add(#3/3/2003#)
  dList.insert(#2/2/2003#, 1)
  Dim s =
    "Simple list of 3 Integer items (reported length " &
     CStr(iList.listLength) & "):" &
     vbCrLf & CStr(iList.listItem(0)) &
     vbCrLf & CStr(iList.listItem(1)) &
     vbCrLf & CStr(iList.listItem(2)) &
     vbCrLf &
     "Simple list of 4 - 1 String items (reported length " &
     CStr(sList.listLength) & "):" &
     vbCrLf & CStr(sList.listItem(0)) &
     vbCrLf & CStr(sList.listItem(1)) &
     vbCrLf & CStr(sList.listItem(2)) &
     vbCrLf &
     "Simple list of 2 + 1 Date items (reported length " &
     CStr(dList.listLength) & "):" &
     vbCrLf & CStr(dList.listItem(0)) &
     vbCrLf & CStr(dList.listItem(1)) &
     vbCrLf & CStr(dList.listItem(2))
  MsgBox(s)
End Sub
```

See also

- Data Types
- Generic Types in Visual Basic
- Language Independence and Language-Independent Components
- Of
- Type List
- How to: Use a Generic Class
- Object Data Type

How to: Use a Generic Class (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

A class that takes *type parameters* is called a *generic class*. If you are using a generic class, you can generate a *constructed class* from it by supplying a *type argument* for each of these parameters. You can then declare a variable of the constructed class type, and you can create an instance of the constructed class and assign it to that variable.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

The following procedure takes a generic class defined in the .NET Framework and creates an instance from it.

To use a class that takes a type parameter

1. At the beginning of your source file, include an Imports Statement (.NET Namespace and Type) to import the System.Collections.Generic namespace. This allows you to refer to the System.Collections.Generic.Queue<T> class without having to fully qualify it to differentiate it from other queue classes such as System.Collections.Queue.

2. Create the object in the normal way, but add `(Of type)` immediately after the class name.

   The following example uses the same class (System.Collections.Generic.Queue<T>) to create two queue objects that hold items of different data types. It adds items to the end of each queue and then removes and displays items from the front of each queue.

```
Public Sub usequeue()
  Dim queueDouble As New System.Collections.Generic.Queue(Of Double)
  Dim queueString As New System.Collections.Generic.Queue(Of String)
  queueDouble.Enqueue(1.1)
  queueDouble.Enqueue(2.2)
  queueDouble.Enqueue(3.3)
  queueDouble.Enqueue(4.4)
  queueString.Enqueue("First string of three")
  queueString.Enqueue("Second string of three")
  queueString.Enqueue("Third string of three")
  Dim s As String = "Queue of Double items (reported length " &
      CStr(queueDouble.Count) & "):"
  For i As Integer = 1 To queueDouble.Count
    s &= vbCrLf & CStr(queueDouble.Dequeue())
  Next i
  s &= vbCrLf & "Queue of String items (reported length " &
      CStr(queueString.Count) & "):"
  For i As Integer = 1 To queueString.Count
    s &= vbCrLf & queueString.Dequeue()
  Next i
  MsgBox(s)
End Sub
```

See also

- Data Types
- Generic Types in Visual Basic
- Language Independence and Language-Independent Components
- Of
- Imports Statement (.NET Namespace and Type)

- How to: Define a Class That Can Provide Identical Functionality on Different Data Types
- Iterators

Generic Procedures in Visual Basic

4/2/2019 • 3 minutes to read • Edit Online

A *generic procedure*, also called a *generic method*, is a procedure defined with at least one type parameter. This allows the calling code to tailor the data types to its requirements each time it calls the procedure.

A procedure is not generic simply by virtue of being defined inside a generic class or a generic structure. To be generic, the procedure must take at least one type parameter, in addition to any normal parameters it might take. A generic class or structure can contain nongeneric procedures, and a nongeneric class, structure, or module can contain generic procedures.

A generic procedure can use its type parameters in its normal parameter list, in its return type if it has one, and in its procedure code.

Type Inference

You can call a generic procedure without supplying any type arguments at all. If you call it this way, the compiler attempts to determine the appropriate data types to pass to the procedure's type arguments. This is called *type inference*. The following code shows a call in which the compiler infers that it should pass type `String` to the type parameter `t`.

```
Public Sub testSub(Of t)(ByVal arg As t)
End Sub
Public Sub callTestSub()
    testSub("Use this string")
End Sub
```

If the compiler cannot infer the type arguments from the context of your call, it reports an error. One possible cause of such an error is an array rank mismatch. For example, suppose you define a normal parameter as an array of a type parameter. If you call the generic procedure supplying an array of a different rank (number of dimensions), the mismatch causes type inference to fail. The following code shows a call in which a two-dimensional array is passed to a procedure that expects a one-dimensional array.

```
Public Sub demoSub(Of t)(ByVal arg() As t)
End Sub

Public Sub callDemoSub()
    Dim twoDimensions(,) As Integer
    demoSub(twoDimensions)
End Sub
```

You can invoke type inference only by omitting all the type arguments. If you supply one type argument, you must supply them all.

Type inference is supported only for generic procedures. You cannot invoke type inference on generic classes, structures, interfaces, or delegates.

Example

Description

The following example defines a generic `Function` procedure to find a particular element in an array. It defines

one type parameter and uses it to construct the two parameters in the parameter list.

Code

```
Public Function findElement(Of T As IComparable) (
        ByVal searchArray As T(), ByVal searchValue As T) As Integer

    If searchArray.GetLength(0) > 0 Then
        For i As Integer = 0 To searchArray.GetUpperBound(0)
            If searchArray(i).CompareTo(searchValue) = 0 Then Return i
        Next i
    End If

    Return -1
End Function
```

Comments

The preceding example requires the ability to compare `searchValue` against each element of `searchArray`. To guarantee this ability, it constrains the type parameter `T` to implement the IComparable<T> interface. The code uses the CompareTo method instead of the `=` operator, because there is no guarantee that a type argument supplied for `T` supports the `=` operator.

You can test the `findElement` procedure with the following code.

```
Public Sub tryFindElement()
    Dim stringArray() As String = {"abc", "def", "xyz"}
    Dim stringSearch As String = "abc"
    Dim integerArray() As Integer = {7, 8, 9}
    Dim integerSearch As Integer = 8
    Dim dateArray() As Date = {#4/17/1969#, #9/20/1998#, #5/31/2004#}
    Dim dateSearch As Date = Microsoft.VisualBasic.DateAndTime.Today
    MsgBox(CStr(findElement(Of String)(stringArray, stringSearch)))
    MsgBox(CStr(findElement(Of Integer)(integerArray, integerSearch)))
    MsgBox(CStr(findElement(Of Date)(dateArray, dateSearch)))
End Sub
```

The preceding calls to `MsgBox` display "0", "1", and "-1" respectively.

See also

- Generic Types in Visual Basic
- How to: Define a Class That Can Provide Identical Functionality on Different Data Types
- How to: Use a Generic Class
- Procedures
- Procedure Parameters and Arguments
- Type List
- Parameter List

Nullable Value Types (Visual Basic)

5/15/2019 • 5 minutes to read • Edit Online

Sometimes you work with a value type that does not have a defined value in certain circumstances. For example, a field in a database might have to distinguish between having an assigned value that is meaningful and not having an assigned value. Value types can be extended to take either their normal values or a null value. Such an extension is called a *nullable type*.

Each nullable type is constructed from the generic Nullable<T> structure. Consider a database that tracks work-related activities. The following example constructs a nullable `Boolean` type and declares a variable of that type. You can write the declaration in three ways:

```
Dim ridesBusToWork1? As Boolean
Dim ridesBusToWork2 As Boolean?
Dim ridesBusToWork3 As Nullable(Of Boolean)
```

The variable `ridesBusToWork` can hold a value of `True`, a value of `False`, or no value at all. Its initial default value is no value at all, which in this case could mean that the information has not yet been obtained for this person. In contrast, `False` could mean that the information has been obtained and the person does not ride the bus to work.

You can declare variables and properties with nullable types, and you can declare an array with elements of a nullable type. You can declare procedures with nullable types as parameters, and you can return a nullable type from a `Function` procedure.

You cannot construct a nullable type on a reference type such as an array, a `String`, or a class. The underlying type must be a value type. For more information, see Value Types and Reference Types.

Using a Nullable Type Variable

The most important members of a nullable type are its HasValue and Value properties. For a variable of a nullable type, HasValue tells you whether the variable contains a defined value. If HasValue is `True`, you can read the value from Value. Note that both HasValue and Value are `ReadOnly` properties.

Default Values

When you declare a variable with a nullable type, its HasValue property has a default value of `False`. This means that by default the variable has no defined value, instead of the default value of its underlying value type. In the following example, the variable `numberOfChildren` initially has no defined value, even though the default value of the `Integer` type is 0.

```
Dim numberOfChildren? As Integer
```

A null value is useful to indicate an undefined or unknown value. If `numberOfChildren` had been declared as `Integer`, there would be no value that could indicate that the information is not currently available.

Storing Values

You store a value in a variable or property of a nullable type in the typical way. The following example assigns a value to the variable `numberOfChildren` declared in the previous example.

```
numberOfChildren = 2
```

If a variable or property of a nullable type contains a defined value, you can cause it to revert to its initial state of not having a value assigned. You do this by setting the variable or property to `Nothing`, as the following example shows.

```
numberOfChildren = Nothing
```

> **NOTE**
>
> Although you can assign `Nothing` to a variable of a nullable type, you cannot test it for `Nothing` by using the equal sign. Comparison that uses the equal sign, `someVar = Nothing`, always evaluates to `Nothing`. You can test the variable's HasValue property for `False`, or test by using the `Is` or `IsNot` operator.

Retrieving Values

To retrieve the value of a variable of a nullable type, you should first test its HasValue property to confirm that it has a value. If you try to read the value when HasValue is `False`, Visual Basic throws an InvalidOperationException exception. The following example shows the recommended way to read the variable `numberOfChildren` of the previous examples.

```
If numberOfChildren.HasValue Then
    MsgBox("There are " & CStr(numberOfChildren) & " children.")
Else
    MsgBox("It is not known how many children there are.")
End If
```

Comparing Nullable Types

When nullable `Boolean` variables are used in Boolean expressions, the result can be `True`, `False`, or `Nothing`. The following is the truth table for `And` and `Or`. Because `b1` and `b2` now have three possible values, there are nine combinations to evaluate.

B1	B2	B1 AND B2	B1 OR B2
`Nothing`	`Nothing`	`Nothing`	`Nothing`
`Nothing`	`True`	`Nothing`	`True`
`Nothing`	`False`	`False`	`Nothing`
`True`	`Nothing`	`Nothing`	`True`
`True`	`True`	`True`	`True`
`True`	`False`	`False`	`True`
`False`	`Nothing`	`False`	`Nothing`
`False`	`True`	`False`	`True`
`False`	`False`	`False`	`False`

When the value of a Boolean variable or expression is `Nothing`, it is neither `true` nor `false`. Consider the

following example.

```
Dim b1? As Boolean
Dim b2? As Boolean
b1 = True
b2 = Nothing

' The following If statement displays "Expression is not true".
If (b1 And b2) Then
    Console.WriteLine("Expression is true")
Else
    Console.WriteLine("Expression is not true")
End If

' The following If statement displays "Expression is not false".
If Not (b1 And b2) Then
    Console.WriteLine("Expression is false")
Else
    Console.WriteLine("Expression is not false")
End If
```

In this example, `b1 And b2` evaluates to `Nothing`. As a result, the `Else` clause is executed in each `If` statement, and the output is as follows:

```
Expression is not true
```

```
Expression is not false
```

> **NOTE**
>
> `AndAlso` and `OrElse`, which use short-circuit evaluation, must evaluate their second operands when the first evaluates to `Nothing`.

Propagation

If one or both of the operands of an arithmetic, comparison, shift, or type operation is nullable, the result of the operation is also nullable. If both operands have values that are not `Nothing`, the operation is performed on the underlying values of the operands, as if neither were a nullable type. In the following example, variables `compare1` and `sum1` are implicitly typed. If you rest the mouse pointer over them, you will see that the compiler infers nullable types for both of them.

```
' Variable n is a nullable type, but both m and n have proper values.
Dim m As Integer = 3
Dim n? As Integer = 2

' The comparison evaluated is 3 > 2, but compare1 is inferred to be of
' type Boolean?.
Dim compare1 = m > n
' The values summed are 3 and 2, but sum1 is inferred to be of type Integer?.
Dim sum1 = m + n

' The following line displays: 3 * 2 * 5 * True
Console.WriteLine($"{m} * {n} * {sum1} * {compare1}")
```

If one or both operands have a value of `Nothing`, the result will be `Nothing`.

```
' Change the value of n to Nothing.
n = Nothing

Dim compare2 = m > n
Dim sum2 = m + n

' Because the values of n, compare2, and sum2 are all Nothing, the
' following line displays: 3 * <null> * <null> * <null>
Console.WriteLine($"{m} * {If(n, "<null>")} * {If(sum2, "<null>")} * {If(compare2, "<null>")}")
```

Using Nullable Types with Data

A database is one of the most important places to use nullable types. Not all database objects currently support nullable types, but the designer-generated table adapters do. See TableAdapter support for nullable types.

See also

- InvalidOperationException
- HasValue
- Using Nullable Types
- Data Types
- Value Types and Reference Types
- Troubleshooting Data Types
- Fill datasets by using TableAdapters
- If Operator
- Local Type Inference
- Is Operator
- IsNot Operator

Value Types and Reference Types

7/2/2019 • 2 minutes to read • Edit Online

There are two kinds of types in Visual Basic: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of the ByRef modifier on parameters).

Value Types

A data type is a *value type* if it holds the data within its own memory allocation. Value types include the following:

- All numeric data types
- `Boolean`, `Char`, and `Date`
- All structures, even if their members are reference types
- Enumerations, since their underlying type is always `SByte`, `Short`, `Integer`, `Long`, `Byte`, `UShort`, `UInteger`, or `ULong`

Every structure is a value type, even if it contains reference type members. For this reason, value types such as `Char` and `Integer` are implemented by .NET Framework structures.

You can declare a value type by using the reserved keyword, for example, `Decimal`. You can also use the `New` keyword to initialize a value type. This is especially useful if the type has a constructor that takes parameters. An example of this is the Decimal(Int32, Int32, Int32, Boolean, Byte) constructor, which builds a new `Decimal` value from the supplied parts.

Reference Types

A *reference type* stores a reference to its data. Reference types include the following:

- `String`
- All arrays, even if their elements are value types
- Class types, such as Form
- Delegates

A class is a *reference type*. Note that every array is a reference type, even if its members are value types.

Since every reference type represents an underlying .NET Framework class, you must use the New Operator keyword when you initialize it. The following statement initializes an array.

```
Dim totals() As Single = New Single(8) {}
```

Elements That Are Not Types

The following programming elements do not qualify as types, because you cannot specify any of them as a data type for a declared element:

- Namespaces
- Modules
- Events
- Properties and procedures
- Variables, constants, and fields

Working with the Object Data Type

You can assign either a reference type or a value type to a variable of the `Object` data type. An `Object` variable always holds a reference to the data, never the data itself. However, if you assign a value type to an `Object` variable, it behaves as if it holds its own data. For more information, see Object Data Type.

You can find out whether an `Object` variable is acting as a reference type or a value type by passing it to the IsReference method in the Information class of the Microsoft.VisualBasic namespace. Information.IsReference returns `True` if the content of the `Object` variable represents a reference type.

See also

- Nullable Value Types
- Type Conversions in Visual Basic
- Structure Statement
- Efficient Use of Data Types
- Object Data Type
- Data Types

Type Conversions in Visual Basic

8/24/2018 • 2 minutes to read • Edit Online

The process of changing a value from one data type to another type is called *conversion*. Conversions are either *widening* or *narrowing*, depending on the data capacities of the types involved. They are also *implicit* or *explicit*, depending on the syntax in the source code.

In This Section

Widening and Narrowing Conversions
Explains conversions classified by whether the destination type can hold the data.

Implicit and Explicit Conversions
Discusses conversions classified by whether Visual Basic performs them automatically.

Conversions Between Strings and Other Types
Illustrates converting between strings and numeric, `Boolean`, or date/time values.

How to: Convert an Object to Another Type in Visual Basic
Shows how to convert an `Object` variable to any other data type.

Array Conversions
Steps you through the process of converting between arrays of different data types.

Related Sections

Data Types
Introduces the Visual Basic data types and describes how to use them.

Data Types
Lists the elementary data types supplied by Visual Basic.

Troubleshooting Data Types
Discusses some common problems that can arise when working with data types.

Widening and Narrowing Conversions (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

An important consideration with a type conversion is whether the result of the conversion is within the range of the destination data type.

A *widening conversion* changes a value to a data type that can allow for any possible value of the original data. Widening conversions preserve the source value but can change its representation. This occurs if you convert from an integral type to `Decimal`, or from `Char` to `String`.

A *narrowing conversion* changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a numeric type being converted to `Boolean` is reduced to either `True` or `False`.

Widening Conversions

The following table shows the standard widening conversions.

DATA TYPE	WIDENS TO DATA TYPES [1]
SByte	`SByte`, `Short`, `Integer`, `Long`, `Decimal`, `Single`, `Double`
Byte	`Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, `Double`
Short	`Short`, `Integer`, `Long`, `Decimal`, `Single`, `Double`
UShort	`UShort`, `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, `Double`
Integer	`Integer`, `Long`, `Decimal`, `Single`, `Double` [2]
UInteger	`UInteger`, `Long`, `ULong`, `Decimal`, `Single`, `Double` [2]
Long	`Long`, `Decimal`, `Single`, `Double` [2]
ULong	`ULong`, `Decimal`, `Single`, `Double` [2]
Decimal	`Decimal`, `Single`, `Double` [2]
Single	`Single`, `Double`
Double	`Double`
Any enumerated type (Enum)	Its underlying integral type and any type to which the underlying type widens.
Char	`Char`, `String`

DATA TYPE	WIDENS TO DATA TYPES
`Char` array	`Char` array, `String`
Any type	Object
Any derived type	Any base type from which it is derived [3].
Any type	Any interface it implements.
Nothing	Any data type or object type.

[1] By definition, every data type widens to itself.

[2] Conversions from `Integer`, `UInteger`, `Long`, `ULong`, or `Decimal` to `Single` or `Double` might result in loss of precision, but never in loss of magnitude. In this sense they do not incur information loss.

[3] It might seem surprising that a conversion from a derived type to one of its base types is widening. The justification is that the derived type contains all the members of the base type, so it qualifies as an instance of the base type. In the opposite direction, the base type does not contain any new members defined by the derived type.

Widening conversions always succeed at run time and never incur data loss. You can always perform them implicitly, whether the Option Strict Statement sets the type checking switch to `On` or to `Off`.

Narrowing Conversions

The standard narrowing conversions include the following:

- The reverse directions of the widening conversions in the preceding table (except that every type widens to itself)
- Conversions in either direction between Boolean and any numeric type
- Conversions from any numeric type to any enumerated type (`Enum`)
- Conversions in either direction between String and any numeric type, `Boolean`, or Date
- Conversions from a data type or object type to a type derived from it

Narrowing conversions do not always succeed at run time, and can fail or incur data loss. An error occurs if the destination data type cannot receive the value being converted. For example, a numeric conversion can result in an overflow. The compiler does not allow you to perform narrowing conversions implicitly unless the Option Strict Statement sets the type checking switch to `Off`.

> **NOTE**
> The narrowing-conversion error is suppressed for conversions from the elements in a `For Each…Next` collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in For Each...Next Statement.

When to Use Narrowing Conversions

You use a narrowing conversion when you know the source value can be converted to the destination data type without error or data loss. For example, if you have a `String` that you know contains either "True" or "False," you can use the `CBool` keyword to convert it to `Boolean`.

Exceptions During Conversion

Because widening conversions always succeed, they do not throw exceptions. Narrowing conversions, when they fail, most commonly throw the following exceptions:

- InvalidCastException — if no conversion is defined between the two types
- OverflowException — (integral types only) if the converted value is too large for the target type

If a class or structure defines a CType Function to serve as a conversion operator to or from that class or structure, that `CType` can throw any exception it deems appropriate. In addition, that `CType` might call Visual Basic functions or .NET Framework methods, which in turn could throw a variety of exceptions.

Changes During Reference Type Conversions

A conversion from a *reference type* copies only the pointer to the value. The value itself is neither copied nor changed in any way. The only thing that can change is the data type of the variable holding the pointer. In the following example, the data type is converted from the derived class to its base class, but the object that both variables now point to is unchanged.

```
' Assume class cSquare inherits from class cShape.
Dim shape As cShape
Dim square As cSquare = New cSquare
' The following statement performs a widening
' conversion from a derived class to its base class.
shape = square
```

See also

- Data Types
- Type Conversions in Visual Basic
- Implicit and Explicit Conversions
- Conversions Between Strings and Other Types
- How to: Convert an Object to Another Type in Visual Basic
- Array Conversions
- Data Types
- Type Conversion Functions

Implicit and Explicit Conversions (Visual Basic)

7/26/2019 • 4 minutes to read • Edit Online

An *implicit conversion* does not require any special syntax in the source code. In the following example, Visual Basic implicitly converts the value of `k` to a single-precision floating-point value before assigning it to `q`.

```
Dim k As Integer
Dim q As Double
' Integer widens to Double, so you can do this with Option Strict On.
k = 432
q = k
```

An *explicit conversion* uses a type conversion keyword. Visual Basic provides several such keywords, which coerce an expression in parentheses to the desired data type. These keywords act like functions, but the compiler generates the code inline, so execution is slightly faster than with a function call.

In the following extension of the preceding example, the `CInt` keyword converts the value of `q` back to an integer before assigning it to `k`.

```
' q had been assigned the value 432 from k.
q = Math.Sqrt(q)
k = CInt(q)
' k now has the value 21 (rounded square root of 432).
```

Conversion Keywords

The following table shows the available conversion keywords.

TYPE CONVERSION KEYWORD	CONVERTS AN EXPRESSION TO DATA TYPE	ALLOWABLE DATA TYPES OF EXPRESSION TO BE CONVERTED
`CBool`	Boolean Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `String`, `Object`
`CByte`	Byte Data Type	Any numeric type (including `SByte` and enumerated types), `Boolean`, `String`, `Object`
`CChar`	Char Data Type	`String`, `Object`
`CDate`	Date Data Type	`String`, `Object`
`CDbl`	Double Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CDec`	Decimal Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`

TYPE CONVERSION KEYWORD	CONVERTS AN EXPRESSION TO DATA TYPE	ALLOWABLE DATA TYPES OF EXPRESSION TO BE CONVERTED
`CInt`	Integer Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CLng`	Long Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CObj`	Object Data Type	Any type
`CSByte`	SByte Data Type	Any numeric type (including `Byte` and enumerated types), `Boolean`, `String`, `Object`
`CShort`	Short Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CSng`	Single Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CStr`	String Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `Char`, `Char` array, `Date`, `Object`
`CType`	Type specified following the comma (`,`)	When converting to an *elementary data type* (including an array of an elementary type), the same types as allowed for the corresponding conversion keyword When converting to a *composite data type*, the interfaces it implements and the classes from which it inherits When converting to a class or structure on which you have overloaded `CType`, that class or structure
`CUInt`	UInteger Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CULng`	ULong Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`
`CUShort`	UShort Data Type	Any numeric type (including `Byte`, `SByte`, and enumerated types), `Boolean`, `String`, `Object`

The CType Function

The CType Function operates on two arguments. The first is the expression to be converted, and the second is the destination data type or object class. Note that the first argument must be an expression, not a type.

`CType` is an *inline function*, meaning the compiled code makes the conversion, often without generating a function call. This improves performance.

For a comparison of `CType` with the other type conversion keywords, see DirectCast Operator and TryCast Operator.

Elementary Types

The following example demonstrates the use of `CType`.

```
k = CType(q, Integer)
' The following statement coerces w to the specific object class Label.
f = CType(w, Label)
```

Composite Types

You can use `CType` to convert values to composite data types as well as to elementary types. You can also use it to coerce an object class to the type of one of its interfaces, as in the following example.

```
' Assume class cZone implements interface iZone.
Dim h As Object
' The first argument to CType must be an expression, not a type.
Dim cZ As cZone
' The following statement coerces a cZone object to its interface iZone.
h = CType(cZ, iZone)
```

Array Types

`CType` can also convert array data types, as in the following example.

```
Dim v() As classV
Dim obArray() As Object
' Assume some object array has been assigned to obArray.
' Check for run-time type compatibility.
If TypeOf obArray Is classV()
    ' obArray can be converted to classV.
    v = CType(obArray, classV())
End If
```

For more information and an example, see Array Conversions.

Types Defining CType

You can define `CType` on a class or structure you have defined. This allows you to convert values to and from the type of your class or structure. For more information and an example, see How to: Define a Conversion Operator.

> **NOTE**
>
> Values used with a conversion keyword must be valid for the destination data type, or an error occurs. For example, if you attempt to convert a `Long` to an `Integer`, the value of the `Long` must be within the valid range for the `Integer` data type.

Caution

Specifying `CType` to convert from one class type to another fails at run time if the source type does not derive from the destination type. Such a failure throws an InvalidCastException exception.

However, if one of the types is a structure or class you have defined, and if you have defined `CType` on that structure or class, a conversion can succeed if it satisfies the requirements of your `CType`. See How to: Define a Conversion Operator.

Performing an explicit conversion is also known as *casting* an expression to a given data type or object class.

See also

- Type Conversions in Visual Basic
- Conversions Between Strings and Other Types
- How to: Convert an Object to Another Type in Visual Basic
- Structures
- Data Types
- Type Conversion Functions
- Troubleshooting Data Types

Conversions Between Strings and Other Types (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

You can convert a numeric, `Boolean`, or date/time value to a `String`. You can also convert in the reverse direction — from a string value to numeric, `Boolean`, or `Date` — provided the contents of the string can be interpreted as a valid value of the destination data type. If they cannot, a run-time error occurs.

The conversions for all these assignments, in either direction, are narrowing conversions. You should use the type conversion keywords (`CBool`, `CByte`, `CDate`, `CDbl`, `CDec`, `CInt`, `CLng`, `CSByte`, `CShort`, `CSng`, `CStr`, `CUInt`, `CULng`, `CUShort`, and `CType`). The Format and Val functions give you additional control over conversions between strings and numbers.

If you have defined a class or structure, you can define type conversion operators between `String` and the type of your class or structure. For more information, see How to: Define a Conversion Operator.

Conversion of Numbers to Strings

You can use the `Format` function to convert a number to a formatted string, which can include not only the appropriate digits but also formatting symbols such as a currency sign (such as `$`), thousands separators or *digit grouping symbols* (such as `,`), and a decimal separator (such as `.`). `Format` automatically uses the appropriate symbols according to the **Regional Options** settings specified in the Windows **Control Panel**.

Note that the concatenation (`&`) operator can convert a number to a string implicitly, as the following example shows.

```
' The following statement converts count to a String value.
Str = "The total count is " & count
```

Conversion of Strings to Numbers

You can use the `Val` function to explicitly convert the digits in a string to a number. `Val` reads the string until it encounters a character other than a digit, space, tab, line feed, or period. The sequences "&O" and "&H" alter the base of the number system and terminate the scanning. Until it stops reading, `Val` converts all appropriate characters to a numeric value. For example, the following statement returns the value `141.825`.

```
Val(" 14 1.825 miles")
```

When Visual Basic converts a string to a numeric value, it uses the **Regional Options** settings specified in the Windows **Control Panel** to interpret the thousands separator, decimal separator, and currency symbol. This means that a conversion might succeed under one setting but not another. For example, `"$14.20"` is acceptable in the English (United States) locale but not in any French locale.

See also

- Type Conversions in Visual Basic
- Widening and Narrowing Conversions
- Implicit and Explicit Conversions
- How to: Convert an Object to Another Type in Visual Basic

- Array Conversions
- Data Types
- Type Conversion Functions
- Introduction to International Applications Based on the .NET Framework

How to: Convert an Object to Another Type in Visual Basic

4/28/2019 • 2 minutes to read • Edit Online

You convert an `Object` variable to another data type by using a conversion keyword such as CType Function.

Example

The following example converts an `Object` variable to an `Integer` and a `String`.

```
Public Sub objectConversion(ByVal anObject As Object)
    Dim anInteger As Integer
    Dim aString As String
    anInteger = CType(anObject, Integer)
    aString = CType(anObject, String)
End Sub
```

If you know that the contents of an `Object` variable are of a particular data type, it is better to convert the variable to that data type. If you continue to use the `Object` variable, you incur either *boxing* and *unboxing* (for a value type) or *late binding* (for a reference type). These operations all take extra execution time and make your performance slower.

Compiling the Code

This example requires:

- A reference to the System namespace.

See also

- Object
- Type Conversions in Visual Basic
- Widening and Narrowing Conversions
- Implicit and Explicit Conversions
- Conversions Between Strings and Other Types
- Array Conversions
- Structures
- Data Types
- Type Conversion Functions

Array Conversions (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

You can convert an array type to a different array type provided you meet the following conditions:

- **Equal Rank.** The ranks of the two arrays must be the same, that is, they must have the same number of dimensions. However, the lengths of the respective dimensions do not need to be the same.
- **Element Data Type.** The data types of the elements of both arrays must be reference types. You cannot convert an `Integer` array to a `Long` array, or even to an `Object` array, because at least one value type is involved. For more information, see Value Types and Reference Types.
- **Convertibility.** A conversion, either widening or narrowing, must be possible between the element types of the two arrays. An example that fails this requirement is an attempted conversion between a `String` array and an array of a class derived from System.Attribute. These two types have nothing in common, and no conversion of any kind exists between them.

A conversion of one array type to another is widening or narrowing depending on whether the conversion of the respective elements is widening or narrowing. For more information, see Widening and Narrowing Conversions.

Conversion to an Object Array

When you declare an `Object` array without initializing it, its element type is `Object` as long as it remains uninitialized. When you set it to an array of a specific class, it takes on the type of that class. However, its underlying type is still `Object`, and you can subsequently set it to another array of an unrelated class. Since all classes derive from `Object`, you can change the array's element type from any class to any other class.

In the following example, no conversion exists between types `student` and `String`, but both derive from `Object`, so all assignments are valid.

```
' Assume student has already been defined as a class.
Dim testArray() As Object
' testArray is still an Object array at this point.
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
testArray = New student(3) {}
' testArray is now of type student().
testArray = names
' testArray is now a String array.
```

Underlying Type of an Array

If you originally declare an array with a specific class, its underlying element type is that class. If you subsequently set it to an array of another class, there must be a conversion between the two classes.

In the following example, `students` is a `student` array. Since no conversion exists between `String` and `student`, the last statement fails.

```
Dim students() As student
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
students = New Student(3) {}
' The following statement fails at compile time.
students = names
```

See also

- Data Types
- Type Conversions in Visual Basic
- Implicit and Explicit Conversions
- Conversions Between Strings and Other Types
- How to: Convert an Object to Another Type in Visual Basic
- Data Types
- Type Conversion Functions
- Arrays

Structures (Visual Basic)

8/24/2018 • 2 minutes to read • Edit Online

A *structure* is a generalization of the user-defined type (UDT) supported by previous versions of Visual Basic. In addition to fields, structures can expose properties, methods, and events. A structure can implement one or more interfaces, and you can declare individual access levels for each field.

You can combine data items of different types to create a structure. A structure associates one or more *elements* with each other and with the structure itself. When you declare a structure, it becomes a *composite data type*, and you can declare variables of that type.

Structures are useful when you want a single variable to hold several related pieces of information. For example, you might want to keep an employee's name, telephone extension, and salary together. You could use several variables for this information, or you could define a structure and use it for a single employee variable. The advantage of the structure becomes apparent when you have many employees and therefore many instances of the variable.

In This Section

How to: Declare a Structure
Shows how to declare a structure and its elements.

Structure Variables
Covers assigning a structure to a variable and accessing its elements.

Structures and Other Programming Elements
Summarizes how structures interact with arrays, objects, procedures, and each other.

Structures and Classes
Describes the similarities and differences between structures and classes.

Related Sections

Data Types
Introduces the Visual Basic data types and describes how to use them.

Data Types
Lists the elementary data types supplied by Visual Basic.

How to: Declare a Structure (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

You begin a structure declaration with the Structure Statement, and you end it with the `End Structure` statement. Between these two statements you must declare at least one *element*. The elements can be of any data type, but at least one must be either a nonshared variable or a nonshared, noncustom event.

You cannot initialize any of the structure elements in the structure declaration. When you declare a variable to be of a structure type, you assign values to the elements by accessing them through the variable.

For a discussion of the differences between structures and classes, see Structures and Classes.

For demonstration purposes, consider a situation where you want to keep track of an employee's name, telephone extension, and salary. A structure allows you to do this in a single variable.

To declare a structure

1. Create the beginning and ending statements for the structure.

   You can specify the access level of a structure using the Public, Protected, Friend, or Private keyword, or you can let it default to `Public`.

   ```
   Private Structure employee
   End Structure
   ```

2. Add elements to the body of the structure.

   A structure must have at least one element. You must declare every element and specify an access level for it. If you use the Dim Statement without any keywords, the accessibility defaults to `Public`.

   ```
   Private Structure employee
       Public givenName As String
       Public familyName As String
       Public phoneExtension As Long
       Private salary As Decimal
       Public Sub giveRaise(raise As Double)
           salary *= raise
       End Sub
       Public Event salaryReviewTime()
   End Structure
   ```

   The `salary` field in the preceding example is `Private`, which means it is inaccessible outside the structure, even from the containing class. However, the `giveRaise` procedure is `Public`, so it can be called from outside the structure. Similarly, you can raise the `salaryReviewTime` event from outside the structure.

   In addition to variables, `Sub` procedures, and events, you can also define constants, `Function` procedures, and properties in a structure. You can designate at most one property as the *default property*, provided it takes at least one argument. You can handle an event with a Shared `Sub` procedure. For more information, see How to: Declare and Call a Default Property in Visual Basic.

See also

- Data Types
- Elementary Data Types

- Composite Data Types
- Value Types and Reference Types
- Structures
- Troubleshooting Data Types
- Structure Variables
- Structures and Other Programming Elements
- Structures and Classes
- User-Defined Data Type

Structure Variables (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

Once you have created a structure, you can declare procedure-level and module-level variables as that type. For example, you can create a structure that records information about a computer system. The following example demonstrates this.

```
Public Structure systemInfo
    Public cPU As String
    Public memory As Long
    Public purchaseDate As Date
End Structure
```

You can now declare variables of that type. The following declaration illustrates this.

```
Dim mySystem, yourSystem As systemInfo
```

> **NOTE**
> In classes and modules, structures declared using the Dim Statement default to public access. If you intend a structure to be private, make sure you declare it using the Private keyword.

Access to Structure Values

To assign and retrieve values from the elements of a structure variable, you use the same syntax as you use to set and get properties on an object. You place the member access operator (`.`) between the structure variable name and the element name. The following example accesses elements of the variables previously declared as type `systemInfo`.

```
mySystem.cPU = "486"
Dim tooOld As Boolean
If yourSystem.purchaseDate < #1/1/1992# Then tooOld = True
```

Assigning Structure Variables

You can also assign one variable to another if both are of the same structure type. This copies all the elements of one structure to the corresponding elements in the other. The following declaration illustrates this.

```
yourSystem = mySystem
```

If a structure element is a reference type, such as a `String`, `Object`, or array, the pointer to the data is copied. In the previous example, if `systemInfo` had included an object variable, then the preceding example would have copied the pointer from `mySystem` to `yourSystem`, and a change to the object's data through one structure would be in effect when accessed through the other structure.

See also

- Data Types
- Elementary Data Types
- Composite Data Types
- Value Types and Reference Types
- Structures
- Troubleshooting Data Types
- How to: Declare a Structure
- Structures and Other Programming Elements
- Structures and Classes
- Structure Statement

Structures and Other Programming Elements (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

You can use structures in conjunction with arrays, objects, and procedures, as well as with each other. The interactions use the same syntax as these elements use individually.

> **NOTE**
> You cannot initialize any of the structure elements in the structure declaration. You can assign values only to elements of a variable that has been declared to be of a structure type.

Structures and Arrays

A structure can contain an array as one or more of its elements. The following example illustrates this.

```
Public Structure systemInfo
    Public cPU As String
    Public memory As Long
    Public diskDrives() As String
    Public purchaseDate As Date
End Structure
```

You access the values of an array within a structure the same way you access a property on an object. The following example illustrates this.

```
Dim mySystem As systemInfo
ReDim mySystem.diskDrives(3)
mySystem.diskDrives(0) = "1.44 MB"
```

You can also declare an array of structures. The following example illustrates this.

```
Dim allSystems(100) As systemInfo
```

You follow the same rules to access the components of this data architecture. The following example illustrates this.

```
ReDim allSystems(5).diskDrives(3)
allSystems(5).CPU = "386SX"
allSystems(5).diskDrives(2) = "100M SCSI"
```

Structures and Objects

A structure can contain an object as one or more of its elements. The following example illustrates this.

```
Protected Structure userInput
    Public userName As String
    Public inputForm As System.Windows.Forms.Form
    Public userFileNumber As Integer
End Structure
```

You should use a specific object class in such a declaration, rather than `Object`.

Structures and Procedures

You can pass a structure as a procedure argument. The following example illustrates this.

```
Public currentCPUName As String = "700MHz Pentium compatible"
Public currentMemorySize As Long = 256
Public Sub fillSystem(ByRef someSystem As systemInfo)
    someSystem.cPU = currentCPUName
    someSystem.memory = currentMemorySize
    someSystem.purchaseDate = Now
End Sub
```

The preceding example passes the structure *by reference*, which allows the procedure to modify its elements so that the changes take effect in the calling code. If you want to protect a structure against such modification, pass it by value.

You can also return a structure from a `Function` procedure. The following example illustrates this.

```
Dim allSystems(100) As systemInfo
Function findByDate(ByVal searchDate As Date) As systemInfo
    Dim i As Integer
    For i = 1 To 100
        If allSystems(i).purchaseDate = searchDate Then Return allSystems(i)
    Next i
   ' Process error: system with desired purchase date not found.
End Function
```

Structures Within Structures

Structures can contain other structures. The following example illustrates this.

```
Public Structure driveInfo
    Public type As String
    Public size As Long
End Structure
Public Structure systemInfo
    Public cPU As String
    Public memory As Long
    Public diskDrives() As driveInfo
    Public purchaseDate As Date
End Structure
```

```
Dim allSystems(100) As systemInfo
ReDim allSystems(1).diskDrives(3)
allSystems(1).diskDrives(0).type = "Floppy"
```

You can also use this technique to encapsulate a structure defined in one module within a structure defined in a different module.

Structures can contain other structures to an arbitrary depth.

See also

- Data Types
- Elementary Data Types
- Composite Data Types
- Value Types and Reference Types
- Structures
- Troubleshooting Data Types
- How to: Declare a Structure
- Structure Variables
- Structures and Classes
- Structure Statement

Structures and Classes (Visual Basic)

4/28/2019 • 4 minutes to read • Edit Online

Visual Basic unifies the syntax for structures and classes, with the result that both entities support most of the same features. However, there are also important differences between structures and classes.

Classes have the advantage of being reference types — passing a reference is more efficient than passing a structure variable with all its data. On the other hand, structures do not require allocation of memory on the global heap.

Because you cannot inherit from a structure, structures should be used only for objects that do not need to be extended. Use structures when the object you wish to create has a small instance size, and take into account the performance characteristics of classes versus structures.

Similarities

Structures and classes are similar in the following respects:

- Both are *container* types, meaning that they contain other types as members.
- Both have members, which can include constructors, methods, properties, fields, constants, enumerations, events, and event handlers. However, do not confuse these members with the declared *elements* of a structure.
- Members of both can have individualized access levels. For example, one member can be declared `Public` and another `Private`.
- Both can implement interfaces.
- Both can have shared constructors, with or without parameters.
- Both can expose a *default property*, provided that property takes at least one parameter.
- Both can declare and raise events, and both can declare delegates.

Differences

Structures and classes differ in the following particulars:

- Structures are *value types*; classes are *reference types*. A variable of a structure type contains the structure's data, rather than containing a reference to the data as a class type does.
- Structures use stack allocation; classes use heap allocation.
- All structure elements are `Public` by default; class variables and constants are `Private` by default, while other class members are `Public` by default. This behavior for class members provides compatibility with the Visual Basic 6.0 system of defaults.
- A structure must have at least one nonshared variable or nonshared, noncustom event element; a class can be completely empty.
- Structure elements cannot be declared as `Protected`; class members can.
- A structure procedure can handle events only if it is a Shared `Sub` procedure, and only by means of the AddHandler Statement; any class procedure can handle events, using either the Handles keyword or the

`AddHandler` statement. For more information, see Events.

- Structure variable declarations cannot specify initializers or initial sizes for arrays; class variable declarations can.
- Structures implicitly inherit from the System.ValueType class and cannot inherit from any other type; classes can inherit from any class or classes other than System.ValueType.
- Structures are not inheritable; classes are.
- Structures are never terminated, so the common language runtime (CLR) never calls the Finalize method on any structure; classes are terminated by the garbage collector (GC), which calls Finalize on a class when it detects there are no active references remaining.
- A structure does not require a constructor; a class does.
- Structures can have nonshared constructors only if they take parameters; classes can have them with or without parameters.

Every structure has an implicit public constructor without parameters. This constructor initializes all the structure's data elements to their default values. You cannot redefine this behavior.

Instances and Variables

Because structures are value types, each structure variable is permanently bound to an individual structure instance. But classes are reference types, and an object variable can refer to various class instances at different times. This distinction affects your usage of structures and classes in the following ways:

- **Initialization.** A structure variable implicitly includes an initialization of the elements using the structure's parameterless constructor. Therefore, `Dim s As struct1` is equivalent to `Dim s As struct1 = New struct1()`.
- **Assigning Variables.** When you assign one structure variable to another, or pass a structure instance to a procedure argument, the current values of all the variable elements are copied to the new structure. When you assign one object variable to another, or pass an object variable to a procedure, only the reference pointer is copied.
- **Assigning Nothing.** You can assign the value Nothing to a structure variable, but the instance continues to be associated with the variable. You can still call its methods and access its data elements, although variable elements are reinitialized by the assignment.

  In contrast, if you set an object variable to `Nothing`, you dissociate it from any class instance, and you cannot access any members through the variable until you assign another instance to it.
- **Multiple Instances.** An object variable can have different class instances assigned to it at different times, and several object variables can refer to the same class instance at the same time. Changes you make to the values of class members affect those members when accessed through another variable pointing to the same instance.

  Structure elements, however, are isolated within their own instance. Changes to their values are not reflected in any other structure variables, even in other instances of the same `Structure` declaration.
- **Equality.** Equality testing of two structures must be performed with an element-by-element test. Two object variables can be compared using the Equals method. Equals indicates whether the two variables point to the same instance.

See also

- Data Types

- Composite Data Types
- Value Types and Reference Types
- Structures
- Troubleshooting Data Types
- Structures and Other Programming Elements
- Objects and Classes

Tuples (Visual Basic)

6/25/2019 • 11 minutes to read • Edit Online

Starting with Visual Basic 2017, the Visual Basic language offers built-in support for tuples that makes creating tuples and accessing the elements of tuples easier. A tuple is a lightweight data structure that has a specific number and sequence of values. When you instantiate the tuple, you define the number and the data type of each value (or element). For example, a 2-tuple (or pair) has two elements. The first might be a `Boolean` value, while the second is a `String`. Because tuples make it easy to store multiple values in a single object, they are often used as a lightweight way to return multiple values from a method.

> **IMPORTANT**
>
> Tuple support requires the ValueTuple type. If the .NET Framework 4.7 is not installed, you must add the NuGet package `System.ValueTuple`, which is available on the NuGet Gallery. Without this package, you may get a compilation error similar to, "Predefined type 'ValueTuple(Of,,,)' is not defined or imported."

Instantiating and using a tuple

You instantiate a tuple by enclosing its comma-delimited values im parentheses. Each of those values then becomes a field of the tuple. For example, the following code defines a triple (or 3-tuple) with a `Date` as its first value, a `String` as its second, and a `Boolean` as its third.

```
Dim holiday = (#07/04/2017#, "Independence Day", True)
```

By default, the name of each field in a tuple consists of the string `Item` along with the field's one-based position in the tuple. For this 3-tuple, the `Date` field is `Item1`, the `String` field is `Item2`, and the `Boolean` field is `Item3`. The following example displays the values of fields of the tuple instantiated in the previous line of code

```
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
                  $"{If(holiday.Item3, ", a national holiday", String.Empty)}")
' Output: 7/4/2017 12:00:00 AM Is Independence Day, a national holiday
```

The fields of a Visual Basic tuple are read-write; after you've instantiated a tuple, you can modify its values. The following example modifies two of the three fields of the tuple created in the previous example and displays the result.

```
holiday.Item1 = #01/01/2018#
holiday.Item2 = "New Year's Day"
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
                  $"{If(holiday.Item3, ", a national holiday", String.Empty)}")
' Output: 1/1/2018 12:00:00 AM Is New Year's Day, a national holiday
```

Instantiating and using a named tuple

Rather than using default names for a tuple's fields, you can instantiate a *named tuple* by assigning your own names to the tuple's elements. The tuple's fields can then be accessed by their assigned names *or* by their default names. The following example instantiates the same 3-tuple as previously, except that it explicitly names the first field `EventDate`, the second `Name`, and the third `IsHoliday`. It then displays the field values, modifies them, and

displays the field values again.

```
Dim holiday = (EventDate:=#07/04/2017#, Name:="Independence Day", IsHoliday:=True)
Console.WriteLine($"{holiday.EventDate} Is {holiday.Name}" +
                  $"{If(holiday.IsHoliday, ", a national holiday", String.Empty)}")
holiday.Item1 = #01/01/2018#
holiday.Item2 = "New Year's Day"
Console.WriteLine($"{holiday.Item1} is {holiday.Item2}" +
                  $"{If(holiday.Item3, ", a national holiday", String.Empty)}")
' The example displays the following output:
'   7/4/2017 12:00:00 AM Is Independence Day, a national holiday
'   1/1/2018 12:00:00 AM Is New Year's Day, a national holiday
```

Inferred tuple element names

Starting with Visual Basic 15.3, Visual Basic can infer the names of tuple elements; you do not have to assign them explicitly. Inferred tuple names are useful when you initialize a tuple from a set of variables, and you want the tuple element name to be the same as the variable name.

The following example creates a `stateInfo` tuple that contains three explicitly named elements, `state`, `stateName`, and `capital`. Note that, in naming the elements, the tuple initialization statement simply assigns the named elements the values of the identically named variables.

```
Dim state = "MI"
Dim stateName = "Michigan"
Dim capital = "Lansing"
Dim stateInfo = ( state:=state, stateName:=stateName, capital:=capital )
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.State}, Capital {stateInfo.capital}")
' The example displays the following output:
'      Michigan: 2-letter code: MI, Capital Lansing
```

Because elements and variables have the same name, the Visual Basic compiler can infer the names of the fields, as the following example shows.

```
Dim state = "MI"
Dim stateName = "Michigan"
Dim capital = "Lansing"
Dim stateInfo = ( state, stateName, capital )
Console.WriteLine($"{stateInfo.stateName}: 2-letter code: {stateInfo.State}, Capital {stateInfo.capital}")
' The example displays the following output:
'      Michigan: 2-letter code: MI, Capital Lansing
```

To enable inferred tuple element names, you must define the version of the Visual Basic compiler to use in your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.3</LangVersion>
</PropertyGroup>
```

The version number can be any version of the Visual Basic compiler starting with 15.3. Rather than hard-coding a specific compiler version, you can also specify "Latest" as the value of `LangVersion` to compile with the most recent version of the Visual Basic compiler installed on your system.

For more information, see setting the Visual Basic language version.

In some cases, the Visual Basic compiler cannot infer the tuple element name from the candidate name, and the tuple field can only be referenced using its default name, such as `Item1`, `Item2`, etc. These include:

- The candidate name is the same as the name of a tuple member, such as `Item3`, `Rest`, or `ToString`.
- The candidate name is duplicated in the tuple.

When field name inference fails, Visual Basic does not generate a compiler error, nor is an exception thrown at runtime. Instead, tuple fields must be referenced by their predefined names, such as `Item1` and `Item2`.

Tuples versus structures

A Visual Basic tuple is a value type that is an instance of one of the a **System.ValueTuple** generic types. For example, the `holiday` tuple defined in the previous example is an instance of the ValueTuple<T1,T2,T3> structure. It is designed to be a lightweight container for data. Since the tuple aims to make it easy to create an object with multiple data items, it lacks some of the features that a custom structure might have. These include:

- Custom members. You cannot define your own properties, methods, or events for a tuple.
- Validation. You cannot validate the data assigned to fields.
- Immutability. Visual Basic tuples are mutable. In contrast, a custom structure allows you to control whether an instance is mutable or immutable.

If custom members, property and field validation, or immutability are important, you should use the Visual Basic Structure statement to define a custom value type.

A Visual Basic tuple does inherit the members of its **ValueTuple** type. In addition to its fields, these include the following methods:

MEMBER	DESCRIPTION
CompareTo	Compares the current tuple to another tuple with the same number of elements.
Equals	Determines whether the current tuple is equal to another tuple or object.
GetHashCode	Calculates the hash code for the current instance.
ToString	Returns the string representation of this tuple, which takes the form `(Item1, Item2...)`, where `Item1` and `Item2` represent the values of the tuple's fields.

In addition, the **ValueTuple** types implement IStructuralComparable and IStructuralEquatable interfaces, which allow you to define customer comparers.

Assignment and tuples

Visual Basic supports assignment between tuple types that have the same number of fields. The field types can be converted if one of the following is true:

- The source and target field are of the same type.
- A widening (or implicit) conversion of the source type to the target type is defined.
- `Option Strict` is `On`, and a narrowing (or explicit) conversion of the source type to the target type is defined. This conversion can throw an exception if the source value is outside the range of the target type.

Other conversions are not considered for assignments. Let's look at the kinds of assignments that are allowed between tuple types.

Consider these variables used in the following examples:

```
' The number and field types of all these tuples are compatible.
' The only difference Is the field names being used.
Dim unnamed = (42, "The meaning of life")
Dim anonymous = (16, "a perfect square")
Dim named = (Answer:=42, Message:="The meaning of life")
Dim differentNamed = (SecretConstant:=42, Label:="The meaning of life")
```

The first two variables, `unnamed` and `anonymous`, do not have semantic names provided for the fields. Their field names are the default `Item1` and `Item2`. The last two variables, `named` and `differentName` have semantic field names. Note that these two tuples have different names for the fields.

All four of these tuples have the same number of fields (referred to as 'arity'), and the types of those fields are identical. Therefore, all of these assignments work:

```
' Assign named to unnamed.
named = unnamed

' Despite the assignment, named still has fields that can be referred to as 'answer' and 'message'.
Console.WriteLine($"{named.Answer}, {named.Message}")
' Output:  42, The meaning of life

' Assign unnamed to anonymous.
anonymous = unnamed
' Because of the assignment, the value of the elements of anonymous changed.
Console.WriteLine($"{anonymous.Item1}, {anonymous.Item2}")
' Output:   42, The meaning of life

' Assign one named tuple to the other.
named = differentNamed
' The field names are Not assigned. 'named' still has 'answer' and 'message' fields.
Console.WriteLine($"{named.Answer}, {named.Message}")
' Output:   42, The meaning of life
```

Notice that the names of the tuples are not assigned. The values of the fields are assigned following the order of the fields in the tuple.

Finally, notice that we can assign the `named` tuple to the `conversion` tuple, even though the first field of `named` is an `Integer`, and the first field of `conversion` is a `Long`. This assignment succeeds because converting an `Integer` to a `Long` is a widening conversion.

```
' Assign an (Integer, String) tuple to a (Long, String) tuple (using implicit conversion).
Dim conversion As (Long, String) = named
Console.WriteLine($"{conversion.Item1} ({conversion.Item1.GetType().Name}), " +
                  $"{conversion.Item2} ({conversion.Item2.GetType().Name})")
' Output:      42 (Int64), The meaning of life (String)
```

Tuples with different numbers of fields are not assignable:

```
' Does not compile.
' VB30311: Value of type '(Integer, Integer, Integer)' cannot be converted
'          to '(Answer As Integer, Message As String)'
var differentShape = (1, 2, 3)
named = differentShape
```

Tuples as method return values

A method can return only a single value. Frequently, though, you'd like a method call to return multiple values. There are several ways to work around this limitation:

- You can create a custom class or structure whose properties or fields represent values returned by the method. Thus is a heavyweight solution; it requires that you define a custom type whose only purpose is to retrieve values from a method call.
- You can return a single value from the method, and return the remaining values by passing them by reference to the method. This involves the overhead of instantiating a variable and risks inadvertently overwriting the value of the variable that you pass by reference.
- You can use a tuple, which provides a lightweight solution to retrieving multiple return values.

For example, the **TryParse** methods in .NET return a `Boolean` value that indicates whether the parsing operation succeeded. The result of the parsing operation is returned in a variable passed by reference to the method. Normally, a call to the a parsing method such as Int32.TryParse looks like the following:

```
Dim numericString As String = "123456"
Dim number As Integer
Dim result = Int32.TryParse(numericString, number)
Console.WriteLine($"{If(result, $"Success: {number:N0}", "Failure")}")
'      Output: 123,456
```

We can return a tuple from the parsing operation if we wrap the call to the Int32.TryParse method in our own method. In the following example, `NumericLibrary.ParseInteger` calls the Int32.TryParse method and returns a named tuple with two elements.

```
Imports System.Globalization

Public Module NumericLibrary
    Public Function ParseInteger(value As String) As (Success As Boolean, Number As Int32)
        Dim number As Integer
        Return (Int32.TryParse(value, NumberStyles.Any, CultureInfo.InvariantCulture, number), number)
    End Function
End Module
```

You can then call the method with code like the following:

```
Dim numericString As String = "123,456"
Dim result = ParseInteger(numericString)
Console.WriteLine($"{If(result.Success, $"Success: {result.Number:N0}", "Failure")}")
Console.ReadLine()
'      Output: Success: 123,456
```

Visual Basic tuples and tuples in the .NET Framework

A Visual Basic tuple is an instance of one of the **System.ValueTuple** generic types, which were introduced in the .NET Framework 4.7. The .NET Framework also includes a set of generic **System.Tuple** classes. These classes, however, differ from Visual Basic tuples and the **System.ValueTuple** generic types in a number of ways:

- The elements of the **Tuple** classes are properties named `Item1`, `Item2`, and so on. In Visual Basic tuples and the **ValueTuple** types, tuple elements are fields.
- You cannot assign meaningful names to the elements of a **Tuple** instance or of a **ValueTuple** instance. Visual Basic allows you to assign names that communicate the meaning of the fields.
- The properties of a **Tuple** instance are read-only; the tuples are immutable. In Visual Basic tuples and the

ValueTuple types, tuple fields are read-write; the tuples are mutable.

- The generic **Tuple** types are reference types. Using these **Tuple** types means allocating objects. On hot paths, this can have a measurable impact on your application's performance. Visual Basic tuples and the **ValueTuple** types are value types.

Extension methods in the TupleExtensions class make it easy to convert between Visual Basic tuples and .NET **Tuple** objects. The **ToTuple** method converts a Visual Basic tuple to a .NET **Tuple** object, and the **ToValueTuple** method converts a .NET **Tuple** object to a Visual Basic tuple.

The following example creates a tuple, converts it to a .NET **Tuple** object, and converts it back to a Visual Basic tuple. The example then compares this tuple with the original one to ensure that they are equal.

```
Module Example
    Sub Main()
        Dim cityInfo = (name:="New York", area:=468.5, population:=8_550_405)
        Console.WriteLine($"{cityInfo}, type {cityInfo.GetType().Name}")

        ' Convert the Visual Basic tuple to a .NET tuple.
        Dim cityInfoT = TupleExtensions.ToTuple(cityInfo)
        Console.WriteLine($"{cityInfoT}, type {cityInfoT.GetType().Name}")

        ' Convert the .NET tuple back to a Visual Basic tuple and ensure they are the same.
        Dim cityInfo2 = TupleExtensions.ToValueTuple(cityInfoT)
        Console.WriteLine($"{cityInfo2}, type {cityInfo2.GetType().Name}")
        Console.WriteLine($"{NameOf(cityInfo)} = {NameOf(cityInfo2)}: {cityInfo.Equals(cityInfo2)}")
        Console.ReadLine()
    End Sub
End Module
' The example displays the following output:
'       (New York, 468.5, 8550405), type ValueTuple`3
'       (New York, 468.5, 8550405), type Tuple`3
'       (New York, 468.5, 8550405), type ValueTuple`3
'       cityInfo = cityInfo2 :  True
```

See also

- Visual Basic Language Reference

Efficient Use of Data Types (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

Undeclared variables and variables declared without a data type are assigned the `Object` data type. This makes it easy to write programs quickly, but it can cause them to execute more slowly.

Strong Typing

Specifying data types for all your variables is known as *strong typing*. Using strong typing has several advantages:

- It enables IntelliSense support for your variables. This allows you to see their properties and other members as you type in the code.
- It takes advantage of compiler type checking. This catches statements that can fail at run time due to errors such as overflow. It also catches calls to methods on objects that do not support them.
- It results in faster execution of your code.

Most Efficient Data Types

For variables that never contain fractions, the integral data types are more efficient than the nonintegral types. In Visual Basic, `Integer` and `UInteger` are the most efficient numeric types.

For fractional numbers, `Double` is the most efficient data type, because the processors on current platforms perform floating-point operations in double precision. However, operations with `Double` are not as fast as with the integral types such as `Integer`.

Specifying Data Type

Use the Dim Statement to declare a variable of a specific type. You can simultaneously specify its access level by using the Public, Protected, Friend, or Private keyword, as in the following example.

```
Private x As Double
Protected s As String
```

Character Conversion

The `AscW` and `ChrW` functions operate in Unicode. You should use them in preference to `Asc` and `Chr`, which must translate into and out of Unicode.

See also

- Asc
- AscW
- Chr
- ChrW
- Data Types
- Numeric Data Types

- Variable Declaration
- Using IntelliSense

Troubleshooting Data Types (Visual Basic)

8/22/2019 • 7 minutes to read • Edit Online

This page lists some common problems that can occur when you perform operations on intrinsic data types.

Floating-Point Expressions Do Not Compare as Equal

When you work with floating-point numbers (Single Data Type and Double Data Type), remember that they are stored as binary fractions. This means they cannot hold an exact representation of any quantity that is not a binary fraction (of the form k / (2 ^ n) where k and n are integers). For example, 0.5 (= 1/2) and 0.3125 (= 5/16) can be held as precise values, whereas 0.2 (= 1/5) and 0.3 (= 3/10) can be only approximations.

Because of this imprecision, you cannot rely on exact results when you operate on floating-point values. In particular, two values that are theoretically equal might have slightly different representations.

TO COMPARE FLOATING-POINT QUANTITIES

1. Calculate the absolute value of their difference by using the Abs method of the Math class in the System namespace.
2. Determine an acceptable maximum difference, such that you can consider the two quantities to be equal for practical purposes if their difference is no larger.
3. Compare the absolute value of the difference to the acceptable difference.

The following example demonstrates both incorrect and correct comparison of two `Double` values.

```
Dim oneThird As Double = 1.0 / 3.0
Dim pointThrees As Double = 0.333333333333333

' The following comparison does not indicate equality.
Dim exactlyEqual As Boolean = (oneThird = pointThrees)

' The following comparison indicates equality.
Dim closeEnough As Double = 0.000000000000001
Dim absoluteDifference As Double = Math.Abs(oneThird - pointThrees)
Dim practicallyEqual As Boolean = (absoluteDifference < closeEnough)

MsgBox("1.0 / 3.0 is represented as " & oneThird.ToString("G17") &
    vbCrLf & "0.333333333333333 is represented as " &
    pointThrees.ToString("G17") &
    vbCrLf & "Exact comparison generates " & CStr(exactlyEqual) &
    vbCrLf & "Acceptable difference comparison generates " &
    CStr(practicallyEqual))
```

The previous example uses the ToString method of the Double structure so that it can specify better precision than the `CStr` keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Mod Operator Does Not Return Accurate Result

Because of the imprecision of floating-point storage, the Mod Operator can return an unexpected result when at least one of the operands is floating-point.

The Decimal Data Type does not use floating-point representation. Many numbers that are inexact in `Single` and `Double` are exact in `Decimal` (for example 0.2 and 0.3). Although arithmetic is slower in `Decimal` than in floating-point, it might be worth the performance decrease to achieve better precision.

TO FIND THE INTEGER REMAINDER OF FLOATING-POINT QUANTITIES

1. Declare variables as `Decimal`.
2. Use the literal type character `D` to force literals to `Decimal`, in case their values are too large for the `Long` data type.

The following example demonstrates the potential imprecision of floating-point operands.

```
Dim two As Double = 2.0
Dim zeroPointTwo As Double = 0.2
Dim quotient As Double = two / zeroPointTwo
Dim doubleRemainder As Double = two Mod zeroPointTwo

MsgBox("2.0 is represented as " & two.ToString("G17") &
    vbCrLf & "0.2 is represented as " & zeroPointTwo.ToString("G17") &
    vbCrLf & "2.0 / 0.2 generates " & quotient.ToString("G17") &
    vbCrLf & "2.0 Mod 0.2 generates " &
    doubleRemainder.ToString("G17"))

Dim decimalRemainder As Decimal = 2D Mod 0.2D
MsgBox("2.0D Mod 0.2D generates " & CStr(decimalRemainder))
```

The previous example uses the ToString method of the Double structure so that it can specify better precision than the `CStr` keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Because `zeroPointTwo` is `Double`, its value for 0.2 is an infinitely repeating binary fraction with a stored value of 0.20000000000000001. Dividing 2.0 by this quantity yields 9.9999999999999995 with a remainder of 0.19999999999999991.

In the expression for `decimalRemainder`, the literal type character `D` forces both operands to `Decimal`, and 0.2 has a precise representation. Therefore the `Mod` operator yields the expected remainder of 0.0.

Note that it is not sufficient to declare `decimalRemainder` as `Decimal`. You must also force the literals to `Decimal`, or they use `Double` by default and `decimalRemainder` receives the same inaccurate value as `doubleRemainder`.

Boolean Type Does Not Convert to Numeric Type Accurately

Boolean Data Type values are not stored as numbers, and the stored values are not intended to be equivalent to numbers. For compatibility with earlier versions, Visual Basic provides conversion keywords (CType Function, `CBool`, `CInt`, and so on) to convert between `Boolean` and numeric types. However, other languages sometimes perform these conversions differently, as do the .NET Framework methods.

You should never write code that relies on equivalent numeric values for `True` and `False`. Whenever possible, you should restrict usage of `Boolean` variables to the logical values for which they are designed. If you must mix `Boolean` and numeric values, make sure that you understand the conversion method that you select.

Conversion in Visual Basic

When you use the `CType` or `CBool` conversion keywords to convert numeric data types to `Boolean`, 0 becomes `False` and all other values become `True`. When you convert `Boolean` values to numeric types by using the conversion keywords, `False` becomes 0 and `True` becomes -1.

Conversion in the Framework

The ToInt32 method of the Convert class in the System namespace converts `True` to +1.

If you must convert a `Boolean` value to a numeric data type, be careful about which conversion method you

use.

Character Literal Generates Compiler Error

In the absence of any type characters, Visual Basic assumes default data types for literals. The default type for a character literal — enclosed in quotation marks (`" "`) — is `String`.

The `String` data type does not widen to the Char Data Type. This means that if you want to assign a literal to a `Char` variable, you must either make a narrowing conversion or force the literal to the `Char` type.

TO CREATE A CHAR LITERAL TO ASSIGN TO A VARIABLE OR CONSTANT

1. Declare the variable or constant as `Char`.
2. Enclose the character value in quotation marks (`" "`).
3. Follow the closing double quotation mark with the literal type character `C` to force the literal to `Char`. This is necessary if the type checking switch (Option Strict Statement) is `On`, and it is desirable in any case.

The following example demonstrates both unsuccessful and successful assignments of a literal to a `Char` variable.

```
Dim charVar As Char
' The following statement attempts to convert a String literal to Char.
' Because Option Strict is On, it generates a compiler error.
charVar = "Z"
' The following statement succeeds because it specifies a Char literal.
charVar = "Z"c
' The following statement succeeds because it converts String to Char.
charVar = CChar("Z")
```

There is always a risk in using narrowing conversions, because they can fail at run time. For example, a conversion from `String` to `Char` can fail if the `String` value contains more than one character. Therefore, it is better programming to use the `C` type character.

String Conversion Fails at Run Time

The String Data Type participates in very few widening conversions. `String` widens only to itself and `Object`, and only `Char` and `Char()` (a `Char` array) widen to `String`. This is because `String` variables and constants can contain values that other data types cannot contain.

When the type checking switch (Option Strict Statement) is `On`, the compiler disallows all implicit narrowing conversions. This includes those involving `String`. Your code can still use conversion keywords such as `CStr` and CType Function, which direct the .NET Framework to attempt the conversion.

> **NOTE**
>
> The narrowing-conversion error is suppressed for conversions from the elements in a `For Each…Next` collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in For Each...Next Statement.

Narrowing Conversion Protection

The disadvantage of narrowing conversions is that they can fail at run time. For example, if a `String` variable contains anything other than "True" or "False," it cannot be converted to `Boolean`. If it contains punctuation characters, conversion to any numeric type fails. Unless you know that your `String` variable always holds values that the destination type can accept, you should not try a conversion.

If you must convert from `String` to another data type, the safest procedure is to enclose the attempted conversion in the Try...Catch...Finally Statement. This lets you deal with a run-time failure.

Character Arrays

A single `Char` and an array of `Char` elements both widen to `String`. However, `String` does not widen to `Char()`. To convert a `String` value to a `Char` array, you can use the ToCharArray method of the System.String class.

Meaningless Values

In general, `String` values are not meaningful in other data types, and conversion is highly artificial and dangerous. Whenever possible, you should restrict usage of `String` variables to the character sequences for which they are designed. You should never write code that relies on equivalent values in other types.

See also

- Data Types
- Type Characters
- Value Types and Reference Types
- Type Conversions in Visual Basic
- Data Types
- Type Conversion Functions
- Efficient Use of Data Types

Declared Elements in Visual Basic

4/28/2019 • 2 minutes to read • Edit Online

A *declared element* is a programming element that is defined in a declaration statement. Declared elements include variables, constants, enumerations, classes, structures, modules, interfaces, procedures, procedure parameters, function returns, external procedure references, operators, properties, events, and delegates.

Declaration statements include the following:

- Dim Statement
- Const Statement
- Enum Statement
- Class Statement
- Structure Statement
- Module Statement
- Interface Statement
- Function Statement
- Sub Statement
- Declare Statement
- Operator Statement
- Property Statement
- Event Statement
- Delegate Statement

In This Section

Declared Element Names
Describes how to name elements and use alphabetic case.

Declared Element Characteristics
Covers characteristics, such as scope, possessed by declared elements.

References to Declared Elements
Describes how the compiler matches a reference to a declaration and how to qualify a name.

Related Sections

Program Structure and Code Conventions
Presents guidelines for making your code easier to read, understand, and maintain.

Statements
Describes statements that name and define procedures, variables, arrays, and constants.

Declaration Contexts and Default Access Levels

Lists the types of declared elements and shows for each one its declaration statement, in what context you can declare it, and its default access level.

Delegates (Visual Basic)

3/8/2019 • 5 minutes to read • Edit Online

Delegates are objects that refer to methods. They are sometimes described as *type-safe function pointers* because they are similar to function pointers used in other programming languages. But unlike function pointers, Visual Basic delegates are a reference type based on the class System.Delegate. Delegates can reference both shared methods — methods that can be called without a specific instance of a class — and instance methods.

Delegates and Events

Delegates are useful in situations where you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately, the object raising the events cannot know ahead of time which event handler is handling a specific event. Visual Basic lets you dynamically associate event handlers with events by creating a delegate for you when you use the `AddHandler` statement. At run time, the delegate forwards calls to the appropriate event handler.

Although you can create your own delegates, in most cases Visual Basic creates the delegate and takes care of the details for you. For example, an `Event` statement implicitly defines a delegate class named `<EventName>EventHandler` as a nested class of the class containing the `Event` statement, and with the same signature as the event. The `AddressOf` statement implicitly creates an instance of a delegate that refers to a specific procedure. The following two lines of code are equivalent. In the first line, you see the explicit creation of an instance of `EventHandler`, with a reference to method `Button1_Click` sent as the argument. The second line is a more convenient way to do the same thing.

```
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
' The following line of code is shorthand for the previous line.
AddHandler Button1.Click, AddressOf Me.Button1_Click
```

You can use the shorthand way of creating delegates anywhere the compiler can determine the delegate's type by the context.

Declaring Events that Use an Existing Delegate Type

In some situations, you may want to declare an event to use an existing delegate type as its underlying delegate. The following syntax demonstrates how:

```
Delegate Sub DelegateType()
Event AnEvent As DelegateType
```

This is useful when you want to route multiple events to the same handler.

Delegate Variables and Parameters

You can use delegates for other, non-event related tasks, such as free threading or with procedures that need to call different versions of functions at run time.

For example, suppose you have a classified-ad application that includes a list box with the names of cars. The ads are sorted by title, which is normally the make of the car. A problem you may face occurs when some cars include the year of the car before the make. The problem is that the built-in sort functionality of the list box sorts only by

character codes; it places all the ads starting with dates first, followed by the ads starting with the make.

To fix this, you can create a sort procedure in a class that uses the standard alphabetic sort on most list boxes, but is able to switch at run time to the custom sort procedure for car ads. To do this, you pass the custom sort procedure to the sort class at run time, using delegates.

AddressOf and Lambda Expressions

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be a reference to a method, or a lambda expression.

To specify a reference to a method, use the following syntax:

`AddressOf` [`expression` .] `methodName`

The compile-time type of the `expression` must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The `methodName` can be either a shared method or an instance method. The `methodName` is not optional, even if you create a delegate for the default method of the class.

To specify a lambda expression, use the following syntax:

`Function` ([`parm` As `type` , `parm2` As `type2` , ...]) `expression`

The following example shows both `AddressOf` and lambda expressions used to specify the reference for a delegate.

```
Module Module1

    Sub Main()
        ' Create an instance of InOrderClass and assign values to the properties.
        ' InOrderClass method ShowInOrder displays the numbers in ascending
        ' or descending order, depending on the comparison method you specify.
        Dim inOrder As New InOrderClass
        inOrder.Num1 = 5
        inOrder.Num2 = 4

        ' Use AddressOf to send a reference to the comparison function you want
        ' to use.
        inOrder.ShowInOrder(AddressOf GreaterThan)
        inOrder.ShowInOrder(AddressOf LessThan)

        ' Use lambda expressions to do the same thing.
        inOrder.ShowInOrder(Function(m, n) m > n)
        inOrder.ShowInOrder(Function(m, n) m < n)
    End Sub

    Function GreaterThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
        Return num1 > num2
    End Function

    Function LessThan(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
        Return num1 < num2
    End Function

    Class InOrderClass
        ' Define the delegate function for the comparisons.
        Delegate Function CompareNumbers(ByVal num1 As Integer, ByVal num2 As Integer) As Boolean
        ' Display properties in ascending or descending order.
        Sub ShowInOrder(ByVal compare As CompareNumbers)
            If compare(_num1, _num2) Then
                Console.WriteLine(_num1 & "  " & _num2)
            Else
                Console.WriteLine(_num2 & "  " & _num1)
            End If
        End Sub

        Private _num1 As Integer
        Property Num1() As Integer
            Get
                Return _num1
            End Get
            Set(ByVal value As Integer)
                _num1 = value
            End Set
        End Property

        Private _num2 As Integer
        Property Num2() As Integer
            Get
                Return _num2
            End Get
            Set(ByVal value As Integer)
                _num2 = value
            End Set
        End Property
    End Class
End Module
```

The signature of the function must match that of the delegate type. For more information about lambda expressions, see Lambda Expressions. For more examples of lambda expression and `AddressOf` assignments to delegates, see Relaxed Delegate Conversion.

Related Topics

TITLE	DESCRIPTION
How to: Invoke a Delegate Method	Provides an example that shows how to associate a method with a delegate and then invoke that method through the delegate.
How to: Pass Procedures to Another Procedure in Visual Basic	Demonstrates how to use delegates to pass one procedure to another procedure.
Relaxed Delegate Conversion	Describes how you can assign subs and functions to delegates or handlers even when their signatures are not identical
Events	Provides an overview of events in Visual Basic.

Early and Late Binding (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler performs a process called `binding` when an object is assigned to an object variable. An object is *early bound* when it is assigned to a variable declared to be of a specific object type. Early bound objects allow the compiler to allocate memory and perform other optimizations before an application executes. For example, the following code fragment declares a variable to be of type FileStream:

```
'  Create a variable to hold a new object.
Dim FS As System.IO.FileStream
' Assign a new object to the variable.
FS = New System.IO.FileStream("C:\tmp.txt",
    System.IO.FileMode.Open)
```

Because FileStream is a specific object type, the instance assigned to `FS` is early bound.

By contrast, an object is *late bound* when it is assigned to a variable declared to be of type `Object`. Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects. For example, the following code fragment declares an object variable to hold an object returned by the `CreateObject` function:

```
' To use this example, you must have Microsoft Excel installed on your computer.
' Compile with Option Strict Off to allow late binding.
Sub TestLateBinding()
    Dim xlApp As Object
    Dim xlBook As Object
    Dim xlSheet As Object
    xlApp = CreateObject("Excel.Application")
    ' Late bind an instance of an Excel workbook.
    xlBook = xlApp.Workbooks.Add
    ' Late bind an instance of an Excel worksheet.
    xlSheet = xlBook.Worksheets(1)
    xlSheet.Activate()
    ' Show the application.
    xlSheet.Application.Visible = True
    ' Place some text in the second row of the sheet.
    xlSheet.Cells(2, 2) = "This is column B row 2"
End Sub
```

Advantages of Early Binding

You should use early-bound objects whenever possible, because they allow the compiler to make important optimizations that yield more efficient applications. Early-bound objects are significantly faster than late-bound objects and make your code easier to read and maintain by stating exactly what kind of objects are being used. Another advantage to early binding is that it enables useful features such as automatic code completion and Dynamic Help because the Visual Studio integrated development environment (IDE) can determine exactly what type of object you are working with as you edit the code. Early binding reduces the number and severity of run-time errors because it allows the compiler to report errors when a program is compiled.

> **NOTE**
>
> Late binding can only be used to access type members that are declared as `Public`. Accessing members declared as `Friend` or `Protected Friend` results in a run-time error.

See also

- CreateObject
- Object Lifetime: How Objects Are Created and Destroyed
- Object Data Type

Error Types (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

In Visual Basic, errors fall into one of three categories: syntax errors, run-time errors, and logic errors.

Syntax Errors

Syntax errors are those that appear while you write code. If you're using Visual Studio, Visual Basic checks your code as you type it in the **Code Editor** window and alerts you if you make a mistake, such as misspelling a word or using a language element improperly. If you compile from the command line, Visual Basic displays a compiler error with information about the syntax error. Syntax errors are the most common type of errors. You can fix them easily in the coding environment as soon as they occur.

> **NOTE**
>
> The `Option Explicit` statement is one means of avoiding syntax errors. It forces you to declare, in advance, all the variables to be used in the application. Therefore, when those variables are used in the code, any typographic errors are caught immediately and can be fixed.

Run-Time Errors

Run-time errors are those that appear only after you compile and run your code. These involve code that may appear to be correct in that it has no syntax errors, but that will not execute. For example, you might correctly write a line of code to open a file. But if the file does not exist, the application cannot open the file, and it throws an exception. You can fix most run-time errors by rewriting the faulty code or by using exception handling, and then recompiling and rerunning it.

Logic Errors

Logic errors are those that appear once the application is in use. They are most often faulty assumptions made by the developer, or unwanted or unexpected results in response to user actions. For example, a mistyped key might provide incorrect information to a method, or you may assume that a valid value is always supplied to a method when that is not the case. Although logic errors can be handled by using exception handling (for example, by testing whether an argument is `Nothing` and throwing an ArgumentNullException), most commonly they should be addressed by correcting the error in logic and recompiling the application.

See also

- Try...Catch...Finally Statement
- Debugger Basics

Events (Visual Basic)

8/22/2019 • 6 minutes to read • Edit Online

While you might visualize a Visual Studio project as a series of procedures that execute in a sequence, in reality, most programs are event driven—meaning the flow of execution is determined by external occurrences called *events*.

An event is a signal that informs an application that something important has occurred. For example, when a user clicks a control on a form, the form can raise a `Click` event and call a procedure that handles the event. Events also allow separate tasks to communicate. Say, for example, that your application performs a sort task separately from the main application. If a user cancels the sort, your application can send a cancel event instructing the sort process to stop.

Event Terms and Concepts

This section describes the terms and concepts used with events in Visual Basic.

Declaring Events

You declare events within classes, structures, modules, and interfaces using the `Event` keyword, as in the following example:

```
Event AnEvent(ByVal EventNumber As Integer)
```

Raising Events

An event is like a message announcing that something important has occurred. The act of broadcasting the message is called *raising* the event. In Visual Basic, you raise events with the `RaiseEvent` statement, as in the following example:

```
RaiseEvent AnEvent(EventNumber)
```

Events must be raised within the scope of the class, module, or structure where they are declared. For example, a derived class cannot raise events inherited from a base class.

Event Senders

Any object capable of raising an event is an *event sender*, also known as an *event source*. Forms, controls, and user-defined objects are examples of event senders.

Event Handlers

Event handlers are procedures that are called when a corresponding event occurs. You can use any valid subroutine with a matching signature as an event handler. You cannot use a function as an event handler, however, because it cannot return a value to the event source.

Visual Basic uses a standard naming convention for event handlers that combines the name of the event sender, an underscore, and the name of the event. For example, the `Click` event of a button named `button1` would be named `Sub button1_Click`.

> **NOTE**
>
> We recommend that you use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid subroutine name.

Associating Events with Event Handlers

Before an event handler becomes usable, you must first associate it with an event by using either the `Handles` or `AddHandler` statement.

WithEvents and the Handles Clause

The `WithEvents` statement and `Handles` clause provide a declarative way of specifying event handlers. An event raised by an object declared with the `WithEvents` keyword can be handled by any procedure with a `Handles` statement for that event, as shown in the following example:

```
' Declare a WithEvents variable.
Dim WithEvents EClass As New EventClass

' Call the method that raises the object's events.
Sub TestEvents()
    EClass.RaiseEvents()
End Sub

' Declare an event handler that handles multiple events.
Sub EClass_EventHandler() Handles EClass.XEvent, EClass.YEvent
    MsgBox("Received Event.")
End Sub

Class EventClass
    Public Event XEvent()
    Public Event YEvent()
    ' RaiseEvents raises both events.
    Sub RaiseEvents()
        RaiseEvent XEvent()
        RaiseEvent YEvent()
    End Sub
End Class
```

The `WithEvents` statement and the `Handles` clause are often the best choice for event handlers because the declarative syntax they use makes event handling easier to code, read and debug. However, be aware of the following limitations on the use of `WithEvents` variables:

- You cannot use a `WithEvents` variable as an object variable. That is, you cannot declare it as `Object`—you must specify the class name when you declare the variable.
- Because shared events are not tied to class instances, you cannot use `WithEvents` to declaratively handle shared events. Similarly, you cannot use `WithEvents` or `Handles` to handle events from a `Structure`. In both cases, you can use the `AddHandler` statement to handle those events.
- You cannot create arrays of `WithEvents` variables.

`WithEvents` variables allow a single event handler to handle one or more kind of event, or one or more event handlers to handle the same kind of event.

Although the `Handles` clause is the standard way of associating an event with an event handler, it is limited to associating events with event handlers at compile time.

In some cases, such as with events associated with forms or controls, Visual Basic automatically stubs out an

empty event handler and associates it with an event. For example, when you double-click a command button on a form in design mode, Visual Basic creates an empty event handler and a `WithEvents` variable for the command button, as in the following code:

```
Friend WithEvents Button1 As System.Windows.Forms.Button
Protected Sub Button1_Click() Handles Button1.Click
End Sub
```

AddHandler and RemoveHandler

The `AddHandler` statement is similar to the `Handles` clause in that both allow you to specify an event handler. However, `AddHandler`, used with `RemoveHandler`, provides greater flexibility than the `Handles` clause, allowing you to dynamically add, remove, and change the event handler associated with an event. If you want to handle shared events or events from a structure, you must use `AddHandler`.

`AddHandler` takes two arguments: the name of an event from an event sender such as a control, and an expression that evaluates to a delegate. You do not need to explicitly specify the delegate class when using `AddHandler`, since the `AddressOf` statement always returns a reference to the delegate. The following example associates an event handler with an event raised by an object:

```
AddHandler Obj.XEvent, AddressOf Me.XEventHandler
```

`RemoveHandler`, which disconnects an event from an event handler, uses the same syntax as `AddHandler`. For example:

```
RemoveHandler Obj.XEvent, AddressOf Me.XEventHandler
```

In the following example, an event handler is associated with an event, and the event is raised. The event handler catches the event and displays a message.

Then the first event handler is removed and a different event handler is associated with the event. When the event is raised again, a different message is displayed.

Finally, the second event handler is removed and the event is raised for a third time. Because there is no longer an event handler associated with the event, no action is taken.

```
Module Module1

    Sub Main()
        Dim c1 As New Class1
        ' Associate an event handler with an event.
        AddHandler c1.AnEvent, AddressOf EventHandler1
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler1
        ' Now associate a different event handler with the event.
        AddHandler c1.AnEvent, AddressOf EventHandler2
        ' Call a method to raise the event.
        c1.CauseTheEvent()
        ' Stop handling the event.
        RemoveHandler c1.AnEvent, AddressOf EventHandler2
        ' This event will not be handled.
        c1.CauseTheEvent()
    End Sub

    Sub EventHandler1()
        ' Handle the event.
        MsgBox("EventHandler1 caught event.")
    End Sub

    Sub EventHandler2()
        ' Handle the event.
        MsgBox("EventHandler2 caught event.")
    End Sub

    Public Class Class1
        ' Declare an event.
        Public Event AnEvent()
        Sub CauseTheEvent()
            ' Raise an event.
            RaiseEvent AnEvent()
        End Sub
    End Class

End Module
```

Handling Events Inherited from a Base Class

Derived classes—classes that inherit characteristics from a base class—can handle events raised by their base class using the `Handles MyBase` statement.

To handle events from a base class

- Declare an event handler in the derived class by adding a `Handles MyBase.` *eventname* statement to the declaration line of your event-handler procedure, where *eventname* is the name of the event in the base class you are handling. For example:

```
Public Class BaseClass
    Public Event BaseEvent(ByVal i As Integer)
    ' Place methods and properties here.
End Class

Public Class DerivedClass
    Inherits BaseClass
    Sub EventHandler(ByVal x As Integer) Handles MyBase.BaseEvent
        ' Place code to handle events from BaseClass here.
    End Sub
End Class
```

Related Sections

TITLE	DESCRIPTION
Walkthrough: Declaring and Raising Events	Provides a step-by-step description of how to declare and raise events for a class.
Walkthrough: Handling Events	Demonstrates how to write an event-handler procedure.
How to: Declare Custom Events To Avoid Blocking	Demonstrates how to define a custom event that allows its event handlers to be called asynchronously.
How to: Declare Custom Events To Conserve Memory	Demonstrates how to define a custom event that uses memory only when the event is handled.
Troubleshooting Inherited Event Handlers in Visual Basic	Lists common issues that arise with event handlers in inherited components.
Events	Provides an overview of the event model in the .NET Framework.
Creating Event Handlers in Windows Forms	Describes how to work with events associated with Windows Forms objects.
Delegates	Provides an overview of delegates in Visual Basic.

Interfaces (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

Interfaces define the properties, methods, and events that classes can implement. Interfaces allow you to define features as small groups of closely related properties, methods, and events; this reduces compatibility problems because you can develop enhanced implementations for your interfaces without jeopardizing existing code. You can add new features at any time by developing additional interfaces and implementations.

There are several other reasons why you might want to use interfaces instead of class inheritance:

- Interfaces are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces are more flexible than base classes because you can define a single implementation that can implement multiple interfaces.
- Interfaces are better in situations in which you do not have to inherit implementation from a base class.
- Interfaces are useful when you cannot use class inheritance. For example, structures cannot inherit from classes, but they can implement interfaces.

Declaring Interfaces

Interface definitions are enclosed within the `Interface` and `End Interface` statements. Following the `Interface` statement, you can add an optional `Inherits` statement that lists one or more inherited interfaces. The `Inherits` statements must precede all other statements in the declaration except comments. The remaining statements in the interface definition should be `Event`, `Sub`, `Function`, `Property`, `Interface`, `Class`, `Structure`, and `Enum` statements. Interfaces cannot contain any implementation code or statements associated with implementation code, such as `End Sub` or `End Property`.

In a namespace, interface statements are `Friend` by default, but they can also be explicitly declared as `Public` or `Friend`. Interfaces defined within classes, modules, interfaces, and structures are `Public` by default, but they can also be explicitly declared as `Public`, `Friend`, `Protected`, or `Private`.

> **NOTE**
>
> The `Shadows` keyword can be applied to all interface members. The `Overloads` keyword can be applied to `Sub`, `Function`, and `Property` statements declared in an interface definition. In addition, `Property` statements can have the `Default`, `ReadOnly`, or `WriteOnly` modifiers. None of the other modifiers— `Public`, `Private`, `Friend`, `Protected`, `Shared`, `Overrides`, `MustOverride`, or `Overridable` —are allowed. For more information, see Declaration Contexts and Default Access Levels.

For example, the following code defines an interface with one function, one property, and one event.

```
Interface IAsset
    Event ComittedChange(ByVal Success As Boolean)
    Property Division() As String
    Function GetID() As Integer
End Interface
```

Implementing Interfaces

The Visual Basic reserved word `Implements` is used in two ways. The `Implements` statement signifies that a class or structure implements an interface. The `Implements` keyword signifies that a class member or structure member implements a specific interface member.

Implements Statement

If a class or structure implements one or more interfaces, it must include the `Implements` statement immediately after the `Class` or `Structure` statement. The `Implements` statement requires a comma-separated list of interfaces to be implemented by a class. The class or structure must implement all interface members using the `Implements` keyword.

Implements Keyword

The `Implements` keyword requires a comma-separated list of interface members to be implemented. Generally, only a single interface member is specified, but you can specify multiple members. The specification of an interface member consists of the interface name, which must be specified in an implements statement within the class; a period; and the name of the member function, property, or event to be implemented. The name of a member that implements an interface member can use any legal identifier, and it is not limited to the `InterfaceName_MethodName` convention used in earlier versions of Visual Basic.

For example, the following code shows how to declare a subroutine named `Sub1` that implements a method of an interface:

```
Class Class1
    Implements interfaceclass.interface2

    Sub Sub1(ByVal i As Integer) Implements interfaceclass.interface2.Sub1
    End Sub
End Class
```

The parameter types and return types of the implementing member must match the interface property or member declaration in the interface. The most common way to implement an element of an interface is with a member that has the same name as the interface, as shown in the previous example.

To declare the implementation of an interface method, you can use any attributes that are legal on instance method declarations, including `Overloads`, `Overrides`, `Overridable`, `Public`, `Private`, `Protected`, `Friend`, `Protected Friend`, `MustOverride`, `Default`, and `Static`. The `Shared` attribute is not legal since it defines a class rather than an instance method.

Using `Implements`, you can also write a single method that implements multiple methods defined in an interface, as in the following example:

```
Class Class2
    Implements I1, I2

    Protected Sub M1() Implements I1.M1, I1.M2, I2.M3, I2.M4
    End Sub
End Class
```

You can use a private member to implement an interface member. When a private member implements a member of an interface, that member becomes available by way of the interface even though it is not available directly on object variables for the class.

Interface Implementation Examples

Classes that implement an interface must implement all its properties, methods, and events.

The following example defines two interfaces. The second interface, `Interface2`, inherits `Interface1` and defines an additional property and method.

```
Interface Interface1
    Sub sub1(ByVal i As Integer)
End Interface

' Demonstrates interface inheritance.
Interface Interface2
    Inherits Interface1
    Sub M1(ByVal y As Integer)
    ReadOnly Property Num() As Integer
End Interface
```

The next example implements `Interface1`, the interface defined in the previous example:

```
Public Class ImplementationClass1
    Implements Interface1
    Sub Sub1(ByVal i As Integer) Implements Interface1.sub1
        ' Insert code here to implement this method.
    End Sub
End Class
```

The final example implements `Interface2`, including a method inherited from `Interface1`:

```
Public Class ImplementationClass2
    Implements Interface2
    Dim INum As Integer = 0
    Sub sub1(ByVal i As Integer) Implements Interface2.sub1
        ' Insert code here that implements this method.
    End Sub
    Sub M1(ByVal x As Integer) Implements Interface2.M1
        ' Insert code here to implement this method.
    End Sub

    ReadOnly Property Num() As Integer Implements Interface2.Num
        Get
            Num = INum
        End Get
    End Property
End Class
```

You can implement a readonly property with a readwrite property (that is, you do not have to declare it readonly in the implementing class). Implementing an interface promises to implement at least the members that the interface declares, but you can offer more functionality, such as allowing your property to be writable.

Related Topics

TITLE	DESCRIPTION
Walkthrough: Creating and Implementing Interfaces	Provides a detailed procedure that takes you through the process of defining and implementing your own interface.
Variance in Generic Interfaces	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.

Walkthrough: Creating and Implementing Interfaces (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Interfaces describe the characteristics of properties, methods, and events, but leave the implementation details up to structures or classes.

This walkthrough demonstrates how to declare and implement an interface.

> **NOTE**
> This walkthrough doesn't provide information about how to create a user interface.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To define an interface

1. Open a new Visual Basic Windows Application project.
2. Add a new module to the project by clicking **Add Module** on the **Project** menu.
3. Name the new module `Module1.vb` and click **Add**. The code for the new module is displayed.
4. Define an interface named `TestInterface` within `Module1` by typing `Interface TestInterface` between the `Module` and `End Module` statements, and then pressing ENTER. The **Code Editor** indents the `Interface` keyword and adds an `End Interface` statement to form a code block.
5. Define a property, method, and event for the interface by placing the following code between the `Interface` and `End Interface` statements:

```
Property Prop1() As Integer
Sub Method1(ByVal X As Integer)
Event Event1()
```

Implementation

You may notice that the syntax used to declare interface members is different from the syntax used to declare class members. This difference reflects the fact that interfaces cannot contain implementation code.

To implement the interface

1. Add a class named `ImplementationClass` by adding the following statement to `Module1`, after the `End Interface` statement but before the `End Module` statement, and then pressing ENTER:

```
Class ImplementationClass
```

If you are working within the integrated development environment, the **Code Editor** supplies a matching `End Class` statement when you press ENTER.

2. Add the following `Implements` statement to `ImplementationClass`, which names the interface the class implements:

```
Implements TestInterface
```

When listed separately from other items at the top of a class or structure, the `Implements` statement indicates that the class or structure implements an interface.

If you are working within the integrated development environment, the **Code Editor** implements the class members required by `TestInterface` when you press ENTER, and you can skip the next step.

3. If you are not working within the integrated development environment, you must implement all the members of the interface `MyInterface`. Add the following code to `ImplementationClass` to implement `Event1`, `Method1`, and `Prop1`:

```
Event Event1() Implements TestInterface.Event1

Public Sub Method1(ByVal X As Integer) Implements TestInterface.Method1
End Sub

Public Property Prop1() As Integer Implements TestInterface.Prop1
    Get
    End Get
    Set(ByVal value As Integer)
    End Set
End Property
```

The `Implements` statement names the interface and interface member being implemented.

4. Complete the definition of `Prop1` by adding a private field to the class that stored the property value:

```
' Holds the value of the property.
Private pval As Integer
```

Return the value of the `pval` from the property get accessor.

```
Return pval
```

Set the value of `pval` in the property set accessor.

```
pval = value
```

5. Complete the definition of `Method1` by adding the following code.

```
MsgBox("The X parameter for Method1 is " & X)
RaiseEvent Event1()
```

To test the implementation of the interface

1. Right-click the startup form for your project in the **Solution Explorer**, and click **View Code**. The editor displays the class for your startup form. By default, the startup form is called `Form1`.

2. Add the following `testInstance` field to the `Form1` class:

```
Dim WithEvents testInstance As TestInterface
```

   By declaring `testInstance` as `WithEvents`, the `Form1` class can handle its events.

3. Add the following event handler to the `Form1` class to handle events raised by `testInstance`:

```
Sub EventHandler() Handles testInstance.Event1
    MsgBox("The event handler caught the event.")
End Sub
```

4. Add a subroutine named `Test` to the `Form1` class to test the implementation class:

```
Sub Test()
    '  Create an instance of the class.
    Dim T As New ImplementationClass
    ' Assign the class instance to the interface.
    ' Calls to the interface members are
    ' executed through the class instance.
    testInstance = T
    ' Set a property.
    testInstance.Prop1 = 9
    ' Read the property.
    MsgBox("Prop1 was set to " & testInstance.Prop1)
    '  Test the method and raise an event.
    testInstance.Method1(5)
End Sub
```

   The `Test` procedure creates an instance of the class that implements `MyInterface`, assigns that instance to the `testInstance` field, sets a property, and runs a method through the interface.

5. Add code to call the `Test` procedure from the `Form1 Load` procedure of your startup form:

```
Private Sub Form1_Load(ByVal sender As System.Object,
                       ByVal e As System.EventArgs) Handles MyBase.Load
    Test() ' Test the class.
End Sub
```

6. Run the `Test` procedure by pressing F5. The message "Prop1 was set to 9" is displayed. After you click OK, the message "The X parameter for Method1 is 5" is displayed. Click OK, and the message "The event handler caught the event" is displayed.

See also

- Implements Statement
- Interfaces
- Interface Statement
- Event Statement

LINQ in Visual Basic

4/2/2019 • 2 minutes to read • Edit Online

This section contains overviews, examples, and background information that will help you understand and use Visual Basic and Language-Integrated Query (LINQ).

In This Section

Introduction to LINQ in Visual Basic
Provides an introduction to LINQ providers, operators, query structure, and language features.

How to: Query a Database
Provides an example of how to connect to a SQL Server database and execute a query by using LINQ.

How to: Call a Stored Procedure
Provides an example of how to connect to a SQL Server database and call a stored procedure by using LINQ.

How to: Modify Data in a Database
Provides an example of how to connect to a SQL Server database and retrieve and modify data by using LINQ.

How to: Combine Data with Joins
Provides examples of how to join data in a manner similar to database joins by using LINQ.

How to: Sort Query Results
Provides an example of how to order the results of a query by using LINQ.

How to: Filter Query Results
Provides an example of how to include search criteria in a query by using LINQ.

How to: Count, Sum, or Average Data
Provides examples of how to include aggregate functions to Count, Sum, or Average data returned from a query by using LINQ.

How to: Find the Minimum or Maximum Value in a Query Result
Provides examples of how to include aggregate functions to determine the minimum and maximum values of data returned from a query by using LINQ.

How to: Return a LINQ Query Result as a Specific Type
Provides an example of how to return the results of a LINQ query as a specific type instead of as an anonymous type.

See also

- LINQ (Language-Integrated Query)
- Overview of LINQ to XML in Visual Basic
- LINQ to DataSet Overview
- LINQ to SQL

Objects and classes in Visual Basic

7/10/2019 • 11 minutes to read • Edit Online

An *object* is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object.

When you create an application in Visual Basic, you constantly work with objects. You can use objects provided by Visual Basic, such as controls, forms, and data access objects. You can also use objects from other applications within your Visual Basic application. You can even create your own objects and define additional properties and methods for them. Objects act like prefabricated building blocks for programs — they let you write a piece of code once and reuse it over and over.

This topic discusses objects in detail.

Objects and classes

Each object in Visual Basic is defined by a *class*. A class describes the variables, properties, procedures, and events of an object. Objects are instances of classes; you can create as many objects you need once you have defined a class.

To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie, for example size and shape. The class is used to create objects. The objects are the cookies.

You must create an object before you can access its members.

To create an object from a class

1. Determine from which class you want to create an object.
2. Write a Dim Statement to create a variable to which you can assign a class instance. The variable should be of the type of the desired class.

```
Dim nextCustomer As customer
```

3. Add the New Operator keyword to initialize the variable to a new instance of the class.

```
Dim nextCustomer As New customer
```

4. You can now access the members of the class through the object variable.

```
nextCustomer.accountNumber = lastAccountNumber + 1
```

> **NOTE**
>
> Whenever possible, you should declare the variable to be of the class type you intend to assign to it. This is called *early binding*. If you don't know the class type at compile time, you can invoke *late binding* by declaring the variable to be of the Object Data Type. However, late binding can make performance slower and limit access to the run-time object's members. For more information, see Object Variable Declaration.

Multiple instances

Objects newly created from a class are often identical to each other. Once they exist as individual objects, however, their variables and properties can be changed independently of the other instances. For example, if you add three check boxes to a form, each check box object is an instance of the CheckBox class. The individual CheckBox objects share a common set of characteristics and capabilities (properties, variables, procedures, and events) defined by the class. However, each has its own name, can be separately enabled and disabled, and can be placed in a different location on the form.

Object members

An object is an element of an application, representing an *instance* of a class. Fields, properties, methods, and events are the building blocks of objects and constitute their *members*.

Member Access

You access a member of an object by specifying, in order, the name of the object variable, a period (`.`), and the name of the member. The following example sets the Text property of a Label object.

```
warningLabel.Text = "Data not saved"
```

IntelliSense listing of members

IntelliSense lists members of a class when you invoke its List Members option, for example when you type a period (`.`) as a member-access operator. If you type the period following the name of a variable declared as an instance of that class, IntelliSense lists all the instance members and none of the shared members. If you type the period following the class name itself, IntelliSense lists all the shared members and none of the instance members. For more information, see Using IntelliSense.

Fields and properties

Fields and *properties* represent information stored in an object. You retrieve and set their values with assignment statements the same way you retrieve and set local variables in a procedure. The following example retrieves the Width property and sets the ForeColor property of a Label object.

```
Dim warningWidth As Integer = warningLabel.Width
warningLabel.ForeColor = System.Drawing.Color.Red
```

Note that a field is also called a *member variable*.

Use property procedures when:

- You need to control when and how a value is set or retrieved.
- The property has a well-defined set of values that need to be validated.
- Setting the value causes some perceptible change in the object's state, such as an `IsVisible` property.
- Setting the property causes changes to other internal variables or to the values of other properties.
- A set of steps must be performed before the property can be set or retrieved.

Use fields when:

- The value is of a self-validating type. For example, an error or automatic data conversion occurs if a value other than `True` or `False` is assigned to a `Boolean` variable.
- Any value in the range supported by the data type is valid. This is true of many properties of type `Single` or `Double`.

- The property is a `String` data type, and there is no constraint on the size or value of the string.
- For more information, see Property Procedures.

Methods

A *method* is an action that an object can perform. For example, Add is a method of the ComboBox object that adds a new entry to a combo box.

The following example demonstrates the Start method of a Timer object.

```
Dim safetyTimer As New System.Windows.Forms.Timer
safetyTimer.Start()
```

Note that a method is simply a *procedure* that is exposed by an object.

For more information, see Procedures.

Events

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be caused by the system. Code that signals an event is said to *raise* the event, and code that responds to it is said to *handle* it.

You can also develop your own custom events to be raised by your objects and handled by other objects. For more information, see Events.

Instance members and shared members

When you create an object from a class, the result is an instance of that class. Members that are not declared with the Shared keyword are *instance members*, which belong strictly to that particular instance. An instance member in one instance is independent of the same member in another instance of the same class. An instance member variable, for example, can have different values in different instances.

Members declared with the `Shared` keyword are *shared members*, which belong to the class as a whole and not to any particular instance. A shared member exists only once, no matter how many instances of its class you create, or even if you create no instances. A shared member variable, for example, has only one value, which is available to all code that can access the class.

Accessing nonshared members

To access a nonshared member of an object

1. Make sure the object has been created from its class and assigned to an object variable.

   ```
   Dim secondForm As New System.Windows.Forms.Form
   ```

2. In the statement that accesses the member, follow the object variable name with the *member-access operator* (`.`) and then the member name.

   ```
   secondForm.Show()
   ```

Accessing shared members

To access a shared member of an object

- Follow the class name with the *member-access operator* (`.`) and then the member name. You should always access a `Shared` member of the object directly through the class name.

  ```
  MsgBox("This computer is called " & Environment.MachineName)
  ```

- If you have already created an object from the class, you can alternatively access a `Shared` member through the object's variable.

Differences between classes and modules

The main difference between classes and modules is that classes can be instantiated as objects while standard modules cannot. Because there is only one copy of a standard module's data, when one part of your program changes a public variable in a standard module, any other part of the program gets the same value if it then reads that variable. In contrast, object data exists separately for each instantiated object. Another difference is that unlike standard modules, classes can implement interfaces.

> **NOTE**
>
> When the `Shared` modifier is applied to a class member, it is associated with the class itself instead of a particular instance of the class. The member is accessed directly by using the class name, the same way module members are accessed.

Classes and modules also use different scopes for their members. Members defined within a class are scoped within a specific instance of the class and exist only for the lifetime of the object. To access class members from outside a class, you must use fully qualified names in the format of *Object.Member*.

On the other hand, members declared within a module are publicly accessible by default, and can be accessed by any code that can access the module. This means that variables in a standard module are effectively global variables because they are visible from anywhere in your project, and they exist for the life of the program.

Reusing classes and objects

Objects let you declare variables and procedures once and then reuse them whenever needed. For example, if you want to add a spelling checker to an application you could define all the variables and support functions to provide spell-checking functionality. If you create your spelling checker as a class, you can then reuse it in other applications by adding a reference to the compiled assembly. Better yet, you may be able to save yourself some work by using a spelling checker class that someone else has already developed.

The .NET Framework provides many examples of components that are available for use. The following example uses the TimeZone class in the System namespace. TimeZone provides members that allow you to retrieve information about the time zone of the current computer system.

```
Public Sub examineTimeZone()
    Dim tz As System.TimeZone = System.TimeZone.CurrentTimeZone
    Dim s As String = "Current time zone is "
    s &= CStr(tz.GetUtcOffset(Now).Hours) & " hours and "
    s &= CStr(tz.GetUtcOffset(Now).Minutes) & " minutes "
    s &= "different from UTC (coordinated universal time)"
    s &= vbCrLf & "and is currently "
    If tz.IsDaylightSavingTime(Now) = False Then s &= "not "
    s &= "on ""summer time""."
    MsgBox(s)
End Sub
```

In the preceding example, the first Dim Statement declares an object variable of type TimeZone and assigns to it a TimeZone object returned by the CurrentTimeZone property.

Relationships among objects

Objects can be related to each other in several ways. The principal kinds of relationship are *hierarchical* and *containment*.

Hierarchical relationship

When classes are derived from more fundamental classes, they are said to have a *hierarchical relationship*. Class hierarchies are useful when describing items that are a subtype of a more general class.

In the following example, suppose you want to define a special kind of Button that acts like a normal Button but also exposes a method that reverses the foreground and background colors.

To define a class is derived from an already existing class

1. Use a Class Statement to define a class from which to create the object you need.

   ```
   Public Class reversibleButton
   ```

   Be sure an `End Class` statement follows the last line of code in your class. By default, the integrated development environment (IDE) automatically generates an `End Class` when you enter a `Class` statement.

2. Follow the `Class` statement immediately with an Inherits Statement. Specify the class from which your new class derives.

   ```
   Inherits System.Windows.Forms.Button
   ```

   Your new class inherits all the members defined by the base class.

3. Add the code for the additional members your derived class exposes. For example, you might add a `reverseColors` method, and your derived class might look as follows:

   ```
   Public Class reversibleButton
       Inherits System.Windows.Forms.Button
           Public Sub reverseColors()
               Dim saveColor As System.Drawing.Color = Me.BackColor
               Me.BackColor = Me.ForeColor
               Me.ForeColor = saveColor
           End Sub
   End Class
   ```

   If you create an object from the `reversibleButton` class, it can access all the members of the Button class, as well as the `reverseColors` method and any other new members you define on `reversibleButton`.

Derived classes inherit members from the class they are based on, allowing you to add complexity as you progress in a class hierarchy. For more information, see Inheritance Basics.

Compiling the code

Be sure the compiler can access the class from which you intend to derive your new class. This might mean fully qualifying its name, as in the preceding example, or identifying its namespace in an Imports Statement (.NET Namespace and Type). If the class is in a different project, you might need to add a reference to that project. For more information, see Managing references in a project.

Containment relationship

Another way that objects can be related is a *containment relationship*. Container objects logically encapsulate other objects. For example, the OperatingSystem object logically contains a Version object, which it returns through its Version property. Note that the container object does not physically contain any other object.

Collections

One particular type of object containment is represented by *collections*. Collections are groups of similar objects that can be enumerated. Visual Basic supports a specific syntax in the For Each...Next Statement that

allows you to iterate through the items of a collection. Additionally, collections often allow you to use an Item[String] to retrieve elements by their index or by associating them with a unique string. Collections can be easier to use than arrays because they allow you to add or remove items without using indexes. Because of their ease of use, collections are often used to store forms and controls.

Related topics

Walkthrough: Defining Classes
Provides a step-by-step description of how to create a class.

Overloaded Properties and Methods
Overloaded Properties and Methods

Inheritance Basics
Covers inheritance modifiers, overriding methods and properties, MyClass, and MyBase.

Object Lifetime: How Objects Are Created and Destroyed
Discusses creating and disposing of class instances.

Anonymous Types
Describes how to create and use anonymous types, which allow you to create objects without writing a class definition for the data type.

Object Initializers: Named and Anonymous Types
Discusses object initializers, which are used to create instances of named and anonymous types by using a single expression.

How to: Infer Property Names and Types in Anonymous Type Declarations
Explains how to infer property names and types in anonymous type declarations. Provides examples of successful and unsuccessful inference.

Operators and Expressions in Visual Basic

4/28/2019 • 2 minutes to read • Edit Online

An *operator* is a code element that performs an operation on one or more code elements that hold values. Value elements include variables, constants, literals, properties, returns from `Function` and `Operator` procedures, and expressions.

An *expression* is a series of value elements combined with operators, which yields a new value. The operators act on the value elements by performing calculations, comparisons, or other operations.

Types of Operators

Visual Basic provides the following types of operators:

- Arithmetic Operators perform familiar calculations on numeric values, including shifting their bit patterns.
- Comparison Operators compare two expressions and return a `Boolean` value representing the result of the comparison.
- Concatenation Operators join multiple strings into a single string.
- Logical and Bitwise Operators in Visual Basic combine `Boolean` or numeric values and return a result of the same data type as the values.

The value elements that are combined with an operator are called *operands* of that operator. Operators combined with value elements form expressions, except for the assignment operator, which forms a *statement*. For more information, see Statements.

Evaluation of Expressions

The end result of an expression represents a value, which is typically of a familiar data type such as `Boolean`, `String`, or a numeric type.

The following are examples of expressions.

```
5 + 4
```

```
' The preceding expression evaluates to 9.
```

```
15 * System.Math.Sqrt(9) + x
```

```
' The preceding expression evaluates to 45 plus the value of x.
```

```
"Concat" & "ena" & "tion"
```

```
' The preceding expression evaluates to "Concatenation".
```

```
763 < 23
```

```
' The preceding expression evaluates to False.
```

Several operators can perform actions in a single expression or statement, as the following example illustrates.

```
x = 45 + y * z ^ 2
```

In the preceding example, Visual Basic performs the operations in the expression on the right side of the assignment operator (`=`), then assigns the resulting value to the variable `x` on the left. There is no practical limit to the number of operators that can be combined into an expression, but an understanding of Operator Precedence in Visual Basic is necessary to ensure that you get the results you expect.

See also

- Operators
- Efficient Combination of Operators
- Statements

Procedures in Visual Basic

4/28/2019 • 3 minutes to read • Edit Online

A *procedure* is a block of Visual Basic statements enclosed by a declaration statement (`Function` , `Sub` , `Operator` , `Get` , `Set`) and a matching `End` declaration. All executable statements in Visual Basic must be within some procedure.

Calling a Procedure

You invoke a procedure from some other place in the code. This is known as a *procedure call*. When the procedure is finished running, it returns control to the code that invoked it, which is known as the *calling code*. The calling code is a statement, or an expression within a statement, that specifies the procedure by name and transfers control to it.

Returning from a Procedure

A procedure returns control to the calling code when it has finished running. To do this, it can use a Return Statement, the appropriate Exit Statement statement for the procedure, or the procedure's End <keyword> Statement statement. Control then passes to the calling code following the point of the procedure call.

- With a `Return` statement, control returns immediately to the calling code. Statements following the `Return` statement do not run. You can have more than one `Return` statement in the same procedure.
- With an `Exit Sub` or `Exit Function` statement, control returns immediately to the calling code. Statements following the `Exit` statement do not run. You can have more than one `Exit` statement in the same procedure, and you can mix `Return` and `Exit` statements in the same procedure.
- If a procedure has no `Return` or `Exit` statements, it concludes with an `End Sub` or `End Function` , `End Get` , or `End Set` statement following the last statement of the procedure body. The `End` statement returns control immediately to the calling code. You can have only one `End` statement in a procedure.

Parameters and Arguments

In most cases, a procedure needs to operate on different data each time you call it. You can pass this information to the procedure as part of the procedure call. The procedure defines zero or more *parameters*, each of which represents a value it expects you to pass to it. Corresponding to each parameter in the procedure definition is an *argument* in the procedure call. An argument represents the value you pass to the corresponding parameter in a given procedure call.

Types of Procedures

Visual Basic uses several types of procedures:

- Sub Procedures perform actions but do not return a value to the calling code.
- Event-handling procedures are `Sub` procedures that execute in response to an event raised by user action or by an occurrence in a program.
- Function Procedures return a value to the calling code. They can perform other actions before returning.

  Some functions written in C# return a *reference return value*. Function callers can modify the return value, and this modification is reflected in the state of the called object. Starting with Visual Basic 2017, Visual

Basic code can consume reference return values, although it cannot return a value by reference. For more information, see Reference return values.

- Property Procedures return and assign values of properties on objects or modules.
- Operator Procedures define the behavior of a standard operator when one or both of the operands is a newly-defined class or structure.
- Generic Procedures in Visual Basic define one or more *type parameters* in addition to their normal parameters, so the calling code can pass specific data types each time it makes a call.

Procedures and Structured Code

Every line of executable code in your application must be inside some procedure, such as `Main`, `calculate`, or `Button1_Click`. If you subdivide large procedures into smaller ones, your application is more readable.

Procedures are useful for performing repeated or shared tasks, such as frequently used calculations, text and control manipulation, and database operations. You can call a procedure from many different places in your code, so you can use procedures as building blocks for your application.

Structuring your code with procedures gives you the following benefits:

- Procedures allow you to break your programs into discrete logical units. You can debug separate units more easily than you can debug an entire program without procedures.
- After you develop procedures for use in one program, you can use them in other programs, often with little or no modification. This helps you avoid code duplication.

See also

- How to: Create a Procedure
- Sub Procedures
- Function Procedures
- Property Procedures
- Operator Procedures
- Procedure Parameters and Arguments
- Recursive Procedures
- Procedure Overloading
- Generic Procedures in Visual Basic
- Objects and Classes

Statements in Visual Basic

8/24/2018 • 12 minutes to read • Edit Online

A statement in Visual Basic is a complete instruction. It can contain keywords, operators, variables, constants, and expressions. Each statement belongs to one of the following categories:

- **Declaration Statements**, which name a variable, constant, or procedure, and can also specify a data type.
- **Executable Statements**, which initiate actions. These statements can call a method or function, and they can loop or branch through blocks of code. Executable statements include **Assignment Statements**, which assign a value or expression to a variable or constant.

This topic describes each category. Also, this topic describes how to combine multiple statements on a single line and how to continue a statement over multiple lines.

Declaration statements

You use declaration statements to name and define procedures, variables, properties, arrays, and constants. When you declare a programming element, you can also define its data type, access level, and scope. For more information, see Declared Element Characteristics.

The following example contains three declarations.

```
Public Sub ApplyFormat()
    Const limit As Integer = 33
    Dim thisWidget As New widget
    ' Insert code to implement the procedure.
End Sub
```

The first declaration is the `Sub` statement. Together with its matching `End Sub` statement, it declares a procedure named `applyFormat`. It also specifies that `applyFormat` is `Public`, which means that any code that can refer to it can call it.

The second declaration is the `Const` statement, which declares the constant `limit`, specifying the `Integer` data type and a value of 33.

The third declaration is the `Dim` statement, which declares the variable `thisWidget`. The data type is a specific object, namely an object created from the `Widget` class. You can declare a variable to be of any elementary data type or of any object type that is exposed in the application you are using.

Initial Values

When the code containing a declaration statement runs, Visual Basic reserves the memory required for the declared element. If the element holds a value, Visual Basic initializes it to the default value for its data type. For more information, see "Behavior" in Dim Statement.

You can assign an initial value to a variable as part of its declaration, as the following example illustrates.

```
Dim m As Integer = 45
' The preceding declaration creates m and assigns the value 45 to it.
```

If a variable is an object variable, you can explicitly create an instance of its class when you declare it by using

the New Operator keyword, as the following example illustrates.

```
Dim f As New System.Windows.Forms.Form()
```

Note that the initial value you specify in a declaration statement is not assigned to a variable until execution reaches its declaration statement. Until that time, the variable contains the default value for its data type.

Executable statements

An executable statement performs an action. It can call a procedure, branch to another place in the code, loop through several statements, or evaluate an expression. An assignment statement is a special case of an executable statement.

The following example uses an `If...Then...Else` control structure to run different blocks of code based on the value of a variable. Within each block of code, a `For...Next` loop runs a specified number of times.

```
Public Sub StartWidget(ByVal aWidget As widget,
    ByVal clockwise As Boolean, ByVal revolutions As Integer)
    Dim counter As Integer
    If clockwise = True Then
        For counter = 1 To revolutions
            aWidget.SpinClockwise()
        Next counter
    Else
        For counter = 1 To revolutions
            aWidget.SpinCounterClockwise()
        Next counter
    End If
End Sub
```

The `If` statement in the preceding example checks the value of the parameter `clockwise`. If the value is `True`, it calls the `spinClockwise` method of `aWidget`. If the value is `False`, it calls the `spinCounterClockwise` method of `aWidget`. The `If...Then...Else` control structure ends with `End If`.

The `For...Next` loop within each block calls the appropriate method a number of times equal to the value of the `revolutions` parameter.

Assignment statements

Assignment statements carry out assignment operations, which consist of taking the value on the right side of the assignment operator (`=`) and storing it in the element on the left, as in the following example.

```
v = 42
```

In the preceding example, the assignment statement stores the literal value 42 in the variable `v`.

Eligible programming elements

The programming element on the left side of the assignment operator must be able to accept and store a value. This means it must be a variable or property that is not ReadOnly, or it must be an array element. In the context of an assignment statement, such an element is sometimes called an *lvalue*, for "left value."

The value on the right side of the assignment operator is generated by an expression, which can consist of any combination of literals, constants, variables, properties, array elements, other expressions, or function calls. The following example illustrates this.

```
x = y + z + FindResult(3)
```

The preceding example adds the value held in variable `y` to the value held in variable `z`, and then adds the value returned by the call to function `findResult`. The total value of this expression is then stored in variable `x`.

Data types in assignment statements

In addition to numeric values, the assignment operator can also assign `String` values, as the following example illustrates.

```
Dim a, b As String
a = "String variable assignment"
b = "Con" & "cat" & "enation"
' The preceding statement assigns the value "Concatenation" to b.
```

You can also assign `Boolean` values, using either a `Boolean` literal or a `Boolean` expression, as the following example illustrates.

```
Dim r, s, t As Boolean
r = True
s = 45 > 1003
t = 45 > 1003 Or 45 > 17
' The preceding statements assign False to s and True to t.
```

Similarly, you can assign appropriate values to programming elements of the `Char`, `Date`, or `Object` data type. You can also assign an object instance to an element declared to be of the class from which that instance is created.

Compound assignment statements

Compound assignment statements first perform an operation on an expression before assigning it to a programming element. The following example illustrates one of these operators, `+=`, which increments the value of the variable on the left side of the operator by the value of the expression on the right.

```
n += 1
```

The preceding example adds 1 to the value of `n`, and then stores that new value in `n`. It is a shorthand equivalent of the following statement:

```
n = n + 1
```

A variety of compound assignment operations can be performed using operators of this type. For a list of these operators and more information about them, see Assignment Operators.

The concatenation assignment operator (`&=`) is useful for adding a string to the end of already existing strings, as the following example illustrates.

```
Dim q As String = "Sample "
q &= "String"
' q now contains "Sample String".
```

Type Conversions in Assignment Statements

The value you assign to a variable, property, or array element must be of a data type appropriate to that

destination element. In general, you should try to generate a value of the same data type as that of the destination element. However, some types can be converted to other types during assignment.

For information on converting between data types, see Type Conversions in Visual Basic. In brief, Visual Basic automatically converts a value of a given type to any other type to which it widens. A *widening conversion* is one in that always succeeds at run time and does not lose any data. For example, Visual Basic converts an `Integer` value to `Double` when appropriate, because `Integer` widens to `Double`. For more information, see Widening and Narrowing Conversions.

Narrowing conversions (those that are not widening) carry a risk of failure at run time, or of data loss. You can perform a narrowing conversion explicitly by using a type conversion function, or you can direct the compiler to perform all conversions implicitly by setting `Option Strict Off`. For more information, see Implicit and Explicit Conversions.

Putting multiple statements on one line

You can have multiple statements on a single line separated by the colon (`:`) character. The following example illustrates this.

```
Dim sampleString As String = "Hello World" : MsgBox(sampleString)
```

Though occasionally convenient, this form of syntax makes your code hard to read and maintain. Thus, it is recommended that you keep one statement to a line.

Continuing a statement over multiple lines

A statement usually fits on one line, but when it is too long, you can continue it onto the next line using a line-continuation sequence, which consists of a space followed by an underscore character (`_`) followed by a carriage return. In the following example, the `MsgBox` executable statement is continued over two lines.

```
Public Sub DemoBox()
    Dim nameVar As String
    nameVar = "John"
    MsgBox("Hello " & nameVar _
        & ". How are you?")
End Sub
```

Implicit line continuation

In many cases, you can continue a statement on the next consecutive line without using the underscore character (`_`). The following syntax elements implicitly continue the statement on the next line of code.

- After a comma (`,`). For example:

  ```
  Public Function GetUsername(ByVal username As String,
                              ByVal delimiter As Char,
                              ByVal position As Integer) As String

      Return username.Split(delimiter)(position)
  End Function
  ```

- After an open parenthesis (`(`) or before a closing parenthesis (`)`). For example:

```
Dim username = GetUsername(
    Security.Principal.WindowsIdentity.GetCurrent().Name,
    CChar("\"),
    1
  )
```

- After an open curly brace (`{`) or before a closing curly brace (`}`). For example:

```
Dim customer = New Customer With {
  .Name = "Terry Adams",
  .Company = "Adventure Works",
  .Email = "terry@www.adventure-works.com"
}
```

  For more information, see Object Initializers: Named and Anonymous Types or Collection Initializers.

- After an open embedded expression (`<%=`) or before the close of an embedded expression (`%>`) within an XML literal. For example:

```
Dim customerXml = <Customer>
                      <Name>
                          <%=
                              customer.Name
                          %>
                      </Name>
                      <Email>
                          <%=
                              customer.Email
                          %>
                      </Email>
                  </Customer>
```

  For more information, see Embedded Expressions in XML.

- After the concatenation operator (`&`). For example:

```
cmd.CommandText =
    "SELECT * FROM Titles JOIN Publishers " &
    "ON Publishers.PubId = Titles.PubID " &
    "WHERE Publishers.State = 'CA'"
```

  For more information, see Operators Listed by Functionality.

- After assignment operators (`=` , `&=` , `:=` , `+=` , `-=` , `*=` , `/=` , `\=` , `^=` , `<<=` , `>>=`). For example:

```
Dim fileStream =
  My.Computer.FileSystem.
    OpenTextFileReader(filePath)
```

  For more information, see Operators Listed by Functionality.

- After binary operators (`+` , `-` , `/` , `*` , `Mod` , `<>` , `<` , `>` , `<=` , `>=` , `^` , `>>` , `<<` , `And` , `AndAlso` , `Or` , `OrElse` , `Like` , `Xor`) within an expression. For example:

```
Dim memoryInUse =
  My.Computer.Info.TotalPhysicalMemory +
  My.Computer.Info.TotalVirtualMemory -
  My.Computer.Info.AvailablePhysicalMemory -
  My.Computer.Info.AvailableVirtualMemory
```

For more information, see Operators Listed by Functionality.

- After the `Is` and `IsNot` operators. For example:

```
If TypeOf inStream Is
  IO.FileStream AndAlso
  inStream IsNot
  Nothing Then

    ReadFile(inStream)

End If
```

For more information, see Operators Listed by Functionality.

- After a member qualifier character (`.`) and before the member name. For example:

```
Dim fileStream =
  My.Computer.FileSystem.
    OpenTextFileReader(filePath)
```

However, you must include a line-continuation character (`_`) following a member qualifier character when you are using the `With` statement or supplying values in the initialization list for a type. Consider breaking the line after the assignment operator (for example, `=`) when you are using `With` statements or object initialization lists. For example:

```
' Not allowed:
' Dim aType = New With { .
'    PropertyName = "Value"

' Allowed:
Dim aType = New With {.PropertyName =
    "Value"}



Dim log As New EventLog()

' Not allowed:
' With log
'     .
'       Source = "Application"
' End With

' Allowed:
With log
    .Source =
      "Application"
End With
```

For more information, see With...End With Statement or Object Initializers: Named and Anonymous Types.

- After an XML axis property qualifier (`.` or `.@` or `...`). However, you must include a line-continuation character (`_`) when you specify a member qualifier when you are using the `With` keyword. For example:

```
Dim customerName = customerXml.
  <Name>.Value

Dim customerEmail = customerXml...
  <Email>.Value
```

  For more information, see XML Axis Properties.

- After a less-than sign (<) or before a greater-than sign (`>`) when you specify an attribute. Also after a greater-than sign (`>`) when you specify an attribute. However, you must include a line-continuation character (`_`) when you specify assembly-level or module-level attributes. For example:

```
<
Serializable()
>
Public Class Customer
    Public Property Name As String
    Public Property Company As String
    Public Property Email As String
End Class
```

  For more information, see Attributes overview.

- Before and after query operators (`Aggregate` , `Distinct` , `From` , `Group By` , `Group Join` , `Join` , `Let` , `Order By` , `Select` , `Skip` , `Skip While` , `Take` , `Take While` , `Where` , `In` , `Into` , `On` , `Ascending` , and `Descending`). You cannot break a line between the keywords of query operators that are made up of multiple keywords (`Order By` , `Group Join` , `Take While` , and `Skip While`). For example:

```
Dim vsProcesses = From proc In
                    Process.GetProcesses
                  Where proc.MainWindowTitle.Contains("Visual Studio")
                  Select proc.ProcessName, proc.Id,
                         proc.MainWindowTitle
```

  For more information, see Queries.

- After the `In` keyword in a `For Each` statement. For example:

```
For Each p In
  vsProcesses

    Console.WriteLine("{0}" & vbTab & "{1}" & vbTab & "{2}",
      p.ProcessName,
      p.Id,
      p.MainWindowTitle)
Next
```

  For more information, see For Each...Next Statement.

- After the `From` keyword in a collection initializer. For example:

```
Dim days = New List(Of String) From
  {
   "Mo", "Tu", "We", "Th", "F", "Sa", "Su"
  }
```

For more information, see Collection Initializers.

Adding comments

Source code is not always self-explanatory, even to the programmer who wrote it. To help document their code, therefore, most programmers make liberal use of embedded comments. Comments in code can explain a procedure or a particular instruction to anyone reading or working with it later. Visual Basic ignores comments during compilation, and they do not affect the compiled code.

Comment lines begin with an apostrophe (`'`) or `REM` followed by a space. They can be added anywhere in code, except within a string. To append a comment to a statement, insert an apostrophe or `REM` after the statement, followed by the comment. Comments can also go on their own separate line. The following example demonstrates these possibilities.

```
' This is a comment on a separate code line.
REM This is another comment on a separate code line.
x += a(i) * b(i) ' Add this amount to total.
MsgBox(statusMessage) REM Inform operator of status.
```

Checking compilation errors

If, after you type a line of code, the line is displayed with a wavy blue underline (an error message may appear as well), there is a syntax error in the statement. You must find out what is wrong with the statement (by looking in the task list, or hovering over the error with the mouse pointer and reading the error message) and correct it. Until you have fixed all syntax errors in your code, your program will fail to compile correctly.

Related sections

TERM	DEFINITION
Assignment Operators	Provides links to language reference pages covering assignment operators such as `=`, `*=`, and `&=`.
Operators and Expressions	Shows how to combine elements with operators to yield new values.
How to: Break and Combine Statements in Code	Shows how to break a single statement into multiple lines and how to place multiple statements on the same line.
How to: Label Statements	Shows how to label a line of code.

Strings in Visual Basic

4/2/2019 • 2 minutes to read • Edit Online

This section describes the basic concepts behind using strings in Visual Basic.

In This Section

Introduction to Strings in Visual Basic

Lists topics that describe the basic concepts behind using strings in Visual Basic.

How to: Create Strings Using a StringBuilder in Visual Basic

Demonstrates how to efficiently create a long string from many smaller strings.

How to: Search Within a String

Demonstrates how to determine the index of the first occurrence of a substring.

Converting Between Strings and Other Data Types in Visual Basic

Lists topics that describe how to convert strings into other data types.

Validating Strings in Visual Basic

Lists topics that discuss how to validate strings.

Walkthrough: Encrypting and Decrypting Strings in Visual Basic

Demonstrates how to encrypt and decrypt strings by using the cryptographic service provider version of the Triple Data Encryption Standard algorithm.

See also

- Visual Basic Language Features

Variables in Visual Basic

8/22/2019 • 2 minutes to read • Edit Online

You often have to store values when you perform calculations with Visual Basic. For example, you might want to calculate several values, compare them, and perform different operations on them, depending on the result of the comparison. You have to retain the values if you want to compare them.

Usage

Visual Basic, just like most programming languages, uses variables for storing values. A *variable* has a name (the word that you use to refer to the value that the variable contains). A variable also has a data type (which determines the kind of data that the variable can store). A variable can represent an array if it has to store an indexed set of closely related data items.

Local type inference enables you to declare variables without explicitly stating a data type. Instead, the compiler infers the type of the variable from the type of the initialization expression. For more information, see Local Type Inference and Option Infer Statement.

Assigning Values

You use assignment statements to perform calculations and assign the result to a variable, as the following example shows.

```
' The following statement assigns the value 10 to the variable.
applesSold = 10
' The following statement increments the variable.
applesSold = applesSold + 1
' The variable now holds the value 11.
```

NOTE

The equal sign (`=`) in this example is an assignment operator, not an equality operator. The value is being assigned to the variable `applesSold` .

For more information, see How to: Move Data Into and Out of a Variable.

Variables and Properties

Like a variable, a *property* represents a value that you can access. However, it is more complex than a variable. A property uses code blocks that control how to set and retrieve its value. For more information, see Differences Between Properties and Variables in Visual Basic.

See also

- Variable Declaration
- Object Variables
- Troubleshooting Variables
- How to: Move Data Into and Out of a Variable
- Differences Between Properties and Variables in Visual Basic

- Local Type Inference

XML in Visual Basic

1/23/2019 • 2 minutes to read • Edit Online

Visual Basic provides integrated language support that enables it to interact with LINQ to XML.

In this section

The topics in this section introduce using LINQ to XML with Visual Basic.

TOPIC	DESCRIPTION
Overview of LINQ to XML in Visual Basic	Describes how Visual Basic supports LINQ to XML.
Creating XML in Visual Basic	Describes how to create XML literal objects by using LINQ to XML.
Manipulating XML in Visual Basic	Describes how to load and parse XML by using Visual Basic.
Accessing XML in Visual Basic	Describes the XML axis properties and LINQ to XML methods for accessing XML elements and attributes.

See also

- System.Xml.Linq
- XML Literals
- XML Axis Properties
- LINQ to XML

COM Interop (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

The Component Object Model (COM) allows an object to expose its functionality to other components and to host applications. Most of today's software includes COM objects. Although .NET assemblies are the best choice for new applications, you may at times need to employ COM objects. This section covers some of the issues associated with creating and using COM objects with Visual Basic.

In This Section

Introduction to COM Interop
Provides an overview of COM interoperability.

How to: Reference COM Objects from Visual Basic
Covers how to add references to COM objects that have type libraries.

How to: Work with ActiveX Controls
Demonstrates how to use existing ActiveX controls to add features to the Visual Studio Toolbox.

Walkthrough: Calling Windows APIs
Steps you through the process of calling the APIs that are part of the Windows operating system.

How to: Call Windows APIs
Demonstrates how to define and call the `MessageBox` function in User32.dll.

How to: Call a Windows Function that Takes Unsigned Types
Demonstrates how to call a Windows function that has a parameter of an unsigned type.

Walkthrough: Creating COM Objects with Visual Basic
Steps you through the process of creating COM objects with and without the COM class template.

Troubleshooting Interoperability
Covers some of the problems you may encounter when using COM.

COM Interoperability in .NET Framework Applications
Provides an overview of how to use COM objects and .NET Framework objects in the same application.

Walkthrough: Implementing Inheritance with COM Objects
Describes using existing COM objects as the basis for new objects.

Related Sections

Interoperating with Unmanaged Code
Describes interoperability services provided by the common language runtime.

Exposing COM Components to the .NET Framework
Describes the process of calling COM types through COM interop.

Exposing .NET Framework Components to COM
Describes the preparation and use of managed types from COM.

Applying Interop Attributes
Covers attributes you can use when working with unmanaged code.

Introduction to COM Interop (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

The Component Object Model (COM) lets an object expose its functionality to other components and to host applications. While COM objects have been fundamental to Windows programming for many years, applications designed for the common language runtime (CLR) offer many advantages.

.NET Framework applications will eventually replace those developed with COM. Until then, you may have to use or create COM objects by using Visual Studio. Interoperability with COM, or *COM interop*, enables you to use existing COM objects while transitioning to the .NET Framework at your own pace.

By using the .NET Framework to create COM components, you can use registration-free COM interop. This lets you control which DLL version is enabled when more than one version is installed on a computer, and lets end users use XCOPY or FTP to copy your application to an appropriate directory on their computer where it can be run. For more information, see Registration-Free COM Interop.

Managed Code and Data

Code developed for the .NET Framework is referred to as *managed code*, and contains metadata that is used by the CLR. Data used by .NET Framework applications is called *managed data* because the runtime manages data-related tasks such as allocating and reclaiming memory and performing type checking. By default, Visual Basic .NET uses managed code and data, but you can access the unmanaged code and data of COM objects using interop assemblies (described later on this page).

Assemblies

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit containing one or more files. Each assembly contains an assembly manifest.

Type Libraries and Assembly Manifests

Type libraries describe characteristics of COM objects, such as member names and data types. Assembly manifests perform the same function for .NET Framework applications. They include information about the following:

- Assembly identity, version, culture, and digital signature.
- Files that make up the assembly implementation.
- Types and resources that make up the assembly. This includes those that are exported from it.
- Compile-time dependencies on other assemblies.
- Permissions required for the assembly to run correctly.

For more information about assemblies and assembly manifests, see Assemblies in .NET.

Importing and Exporting Type Libraries

Visual Studio contains a utility, Tlbimp, that lets you import information from a type library into a .NET Framework application. You can generate type libraries from assemblies by using the Tlbexp utility.

For information about Tlbimp and Tlbexp, see Tlbimp.exe (Type Library Importer) and Tlbexp.exe (Type Library Exporter).

Interop Assemblies

Interop assemblies are .NET Framework assemblies that bridge between managed and unmanaged code, mapping COM object members to equivalent .NET Framework managed members. Interop assemblies created by Visual Basic .NET handle many of the details of working with COM objects, such as interoperability marshaling.

Interoperability Marshaling

All .NET Framework applications share a set of common types that enable interoperability of objects, regardless of the programming language that is used. The parameters and return values of COM objects sometimes use data types that differ from those used in managed code. *Interoperability marshaling* is the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. For more information, see Interop Marshaling.

See also

- COM Interop
- Walkthrough: Implementing Inheritance with COM Objects
- Interoperating with Unmanaged Code
- Troubleshooting Interoperability
- Assemblies in .NET
- Tlbimp.exe (Type Library Importer)
- Tlbexp.exe (Type Library Exporter)
- Interop Marshaling
- Registration-Free COM Interop

How to: Reference COM Objects from Visual Basic

5/14/2019 • 2 minutes to read • Edit Online

In Visual Basic, adding references to COM objects that have type libraries requires the creation of an interop assembly for the COM library. References to the members of the COM object are routed to the interop assembly and then forwarded to the actual COM object. Responses from the COM object are routed to the interop assembly and forwarded to your .NET Framework application.

You can reference a COM object without using an interop assembly by embedding the type information for the COM object in a .NET assembly. To embed type information, set the `Embed Interop Types` property to `True` for the reference to the COM object. If you are compiling by using the command-line compiler, use the `/link` option to reference the COM library. For more information, see /link (Visual Basic).

Visual Basic automatically creates interop assemblies when you add a reference to a type library from the integrated development environment (IDE). When working from the command line, you can use the Tlbimp utility to manually create interop assemblies.

To add references to COM objects

1. On the **Project** menu, choose **Add Reference** and then click the **COM** tab in the dialog box.
2. Select the component you want to use from the list of COM objects.
3. To simplify access to the interop assembly, add an `Imports` statement to the top of the class or module in which you will use the COM object. For example, the following code example imports the namespace `INKEDLib` for objects referenced in the `Microsoft InkEdit Control 1.0` library.

```
Imports INKEDLib

Class Sample
    Private s As IInkCursor

End Class
```

To create an interop assembly using Tlbimp

1. Add the location of Tlbimp to the search path, if it is not already part of the search path and you are not currently in the directory where it is located.
2. Call Tlbimp from a command prompt, providing the following information:

   - Name and location of the DLL that contains the type library
   - Name and location of the namespace where the information should be placed
   - Name and location of the target interop assembly

   The following code provides an example:

```
Tlbimp test3.dll /out:NameSpace1 /out:Interop1.dll
```

   You can use Tlbimp to create interop assemblies for type libraries, even for unregistered COM objects. However, the COM objects referred to by interop assemblies must be properly registered on the computer where they are to be used. You can register a COM object by using the Regsvr32 utility included with the Windows operating system.

See also

- COM Interop
- Tlbimp.exe (Type Library Importer)
- Tlbexp.exe (Type Library Exporter)
- Walkthrough: Implementing Inheritance with COM Objects
- Troubleshooting Interoperability
- Imports Statement (.NET Namespace and Type)

How to: Work with ActiveX Controls (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

ActiveX controls are COM components or objects you can insert into a Web page or other application to reuse packaged functionality someone else has programmed. You can use ActiveX controls developed for Visual Basic 6.0 and earlier versions to add features to the **Toolbox** of Visual Studio.

To add ActiveX controls to the toolbox

1. On the **Tools** menu, click **Choose Toolbox Items**.

   The **Choose Toolbox** dialog box appears.

2. Click the **COM Components** tab.

3. Select the check box next to the ActiveX control you want to use, and then click **OK**.

   The new control appears with the other tools in the **Toolbox**.

> **NOTE**
>
> You can use the Aximp utility to manually create an interop assembly for ActiveX controls. For more information, see Aximp.exe (Windows Forms ActiveX Control Importer).

See also

- COM Interop
- How to: Add ActiveX Controls to Windows Forms
- Aximp.exe (Windows Forms ActiveX Control Importer)
- Considerations When Hosting an ActiveX Control on a Windows Form
- Troubleshooting Interoperability

Walkthrough: Calling Windows APIs (Visual Basic)

8/22/2019 • 9 minutes to read • Edit Online

Windows APIs are dynamic-link libraries (DLLs) that are part of the Windows operating system. You use them to perform tasks when it is difficult to write equivalent procedures of your own. For example, Windows provides a function named `FlashWindowEx` that lets you make the title bar for an application alternate between light and dark shades.

The advantage of using Windows APIs in your code is that they can save development time because they contain dozens of useful functions that are already written and waiting to be used. The disadvantage is that Windows APIs can be difficult to work with and unforgiving when things go wrong.

Windows APIs represent a special category of interoperability. Windows APIs do not use managed code, do not have built-in type libraries, and use data types that are different than those used with Visual Studio. Because of these differences, and because Windows APIs are not COM objects, interoperability with Windows APIs and the .NET Framework is performed using platform invoke, or PInvoke. Platform invoke is a service that enables managed code to call unmanaged functions implemented in DLLs. For more information, see Consuming Unmanaged DLL Functions. You can use PInvoke in Visual Basic by using either the `Declare` statement or applying the `DllImport` attribute to an empty procedure.

Windows API calls were an important part of Visual Basic programming in the past, but are seldom necessary with Visual Basic .NET. Whenever possible, you should use managed code from the .NET Framework to perform tasks, instead of Windows API calls. This walkthrough provides information for those situations in which using Windows APIs is necessary.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

API Calls Using Declare

The most common way to call Windows APIs is by using the `Declare` statement.

To declare a DLL procedure

1. Determine the name of the function you want to call, plus its arguments, argument types, and return value, as well as the name and location of the DLL that contains it.

   > **NOTE**
   > For complete information about the Windows APIs, see the Win32 SDK documentation in the Platform SDK Windows API. For more information about the constants that Windows APIs use, examine the header files such as Windows.h included with the Platform SDK.

2. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.

3. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.

4. Add the following `Declare` function either to the class or module in which you want to use the DLL:

```
Declare Auto Function MBox Lib "user32.dll" Alias "MessageBox" (
    ByVal hWnd As Integer,
    ByVal txt As String,
    ByVal caption As String,
    ByVal Typ As Integer) As Integer
```

Parts of the Declare Statement

The `Declare` statement includes the following elements.

Auto modifier

The `Auto` modifier instructs the runtime to convert the string based on the method name according to common language runtime rules (or alias name if specified).

Lib and Alias keywords

The name following the `Function` keyword is the name your program uses to access the imported function. It can be the same as the real name of the function you are calling, or you can use any valid procedure name and then employ the `Alias` keyword to specify the real name of the function you are calling.

Specify the `Lib` keyword, followed by the name and location of the DLL that contains the function you are calling. You do not need to specify the path for files located in the Windows system directories.

Use the `Alias` keyword if the name of the function you are calling is not a valid Visual Basic procedure name, or conflicts with the name of other items in your application. `Alias` indicates the true name of the function being called.

Argument and Data Type Declarations

Declare the arguments and their data types. This part can be challenging because the data types that Windows uses do not correspond to Visual Studio data types. Visual Basic does a lot of the work for you by converting arguments to compatible data types, a process called *marshaling*. You can explicitly control how arguments are marshaled by using the MarshalAsAttribute attribute defined in the System.Runtime.InteropServices namespace.

> **NOTE**
>
> Previous versions of Visual Basic allowed you to declare parameters `As Any`, meaning that data of any data type could be used. Visual Basic requires that you use a specific data type for all `Declare` statements.

Windows API Constants

Some arguments are combinations of constants. For example, the `MessageBox` API shown in this walkthrough accepts an integer argument called `Typ` that controls how the message box is displayed. You can determine the numeric value of these constants by examining the `#define` statements in the file WinUser.h. The numeric values are generally shown in hexadecimal, so you may want to use a calculator to add them and convert to decimal. For example, if you want to combine the constants for the exclamation style `MB_ICONEXCLAMATION` 0x00000030 and the Yes/No style `MB_YESNO` 0x00000004, you can add the numbers and get a result of 0x00000034, or 52 decimal. Although you can use the decimal result directly, it is better to declare these values as constants in your application and combine them using the `Or` operator.

To declare constants for Windows API calls

1. Consult the documentation for the Windows function you are calling. Determine the name of the constants it uses and the name of the .h file that contains the numeric values for these constants.

2. Use a text editor, such as Notepad, to view the contents of the header (.h) file, and find the values associated with the constants you are using. For example, the `MessageBox` API uses the constant `MB_ICONQUESTION` to show a question mark in the message box. The definition for `MB_ICONQUESTION` is in WinUser.h and appears

as follows:

```
#define MB_ICONQUESTION 0x00000020L
```

3. Add equivalent `Const` statements to your class or module to make these constants available to your application. For example:

```
Const MB_ICONQUESTION As Integer = &H20
Const MB_YESNO As Integer = &H4
Const IDYES As Integer = 6
Const IDNO As Integer = 7
```

To call the DLL procedure

1. Add a button named `Button1` to the startup form for your project, and then double-click it to view its code. The event handler for the button is displayed.

2. Add code to the `Click` event handler for the button you added, to call the procedure and provide the appropriate arguments:

```
Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button1.Click

    ' Stores the return value.
    Dim RetVal As Integer
    RetVal = MBox(0, "Declare DLL Test", "Windows API MessageBox",
        MB_ICONQUESTION Or MB_YESNO)

    ' Check the return value.
    If RetVal = IDYES Then
        MsgBox("You chose Yes")
    Else
        MsgBox("You chose No")
    End If
End Sub
```

3. Run the project by pressing F5. The message box is displayed with both **Yes** and **No** response buttons. Click either one.

Data Marshaling

Visual Basic automatically converts the data types of parameters and return values for Windows API calls, but you can use the `MarshalAs` attribute to explicitly specify unmanaged data types that an API expects. For more information about interop marshaling, see Interop Marshaling.

To use Declare and MarshalAs in an API call

1. Determine the name of the function you want to call, plus its arguments, data types, and return value.

2. To simplify access to the `MarshalAs` attribute, add an `Imports` statement to the top of the code for the class or module, as in the following example:

```
Imports System.Runtime.InteropServices
```

3. Add a function prototype for the imported function to the class or module you are using, and apply the `MarshalAs` attribute to the parameters or return value. In the following example, an API call that expects the type `void*` is marshaled as `AsAny`:

```
Declare Sub SetData Lib "..\LIB\UnmgdLib.dll" (
    ByVal x As Short,
    <MarshalAsAttribute(UnmanagedType.AsAny)>
        ByVal o As Object)
```

API Calls Using DllImport

The `DllImport` attribute provides a second way to call functions in DLLs without type libraries. `DllImport` is roughly equivalent to using a `Declare` statement but provides more control over how functions are called.

You can use `DllImport` with most Windows API calls as long as the call refers to a shared (sometimes called *static*) method. You cannot use methods that require an instance of a class. Unlike `Declare` statements, `DllImport` calls cannot use the `MarshalAs` attribute.

To call a Windows API using the DllImport attribute

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.
3. Add a button named `Button2` to the startup form.
4. Double-click `Button2` to open the code view for the form.
5. To simplify access to `DllImport`, add an `Imports` statement to the top of the code for the startup form class:

   ```
   Imports System.Runtime.InteropServices
   ```

6. Declare an empty function preceding the `End Class` statement for the form, and name the function `MoveFile`.
7. Apply the `Public` and `Shared` modifiers to the function declaration and set parameters for `MoveFile` based on the arguments the Windows API function uses:

   ```
   Public Shared Function MoveFile(
       ByVal src As String,
       ByVal dst As String) As Boolean
       ' Leave the body of the function empty.
   End Function
   ```

   Your function can have any valid procedure name; the `DllImport` attribute specifies the name in the DLL. It also handles interoperability marshaling for the parameters and return values, so you can choose Visual Studio data types that are similar to the data types the API uses.

8. Apply the `DllImport` attribute to the empty function. The first parameter is the name and location of the DLL containing the function you are calling. You do not need to specify the path for files located in the Windows system directories. The second parameter is a named argument that specifies the name of the function in the Windows API. In this example, the `DllImport` attribute forces calls to `MoveFile` to be forwarded to `MoveFileW` in KERNEL32.DLL. The `MoveFileW` method copies a file from the path `src` to the path `dst`.

```
<DllImport("KERNEL32.DLL", EntryPoint:="MoveFileW", SetLastError:=True,
    CharSet:=CharSet.Unicode, ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(
    ByVal src As String,
    ByVal dst As String) As Boolean
    ' Leave the body of the function empty.
End Function
```

9. Add code to the `Button2_Click` event handler to call the function:

```
Private Sub Button2_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles Button2.Click

    Dim RetVal As Boolean = MoveFile("c:\tmp\Test.txt", "c:\Test.txt")
    If RetVal = True Then
        MsgBox("The file was moved successfully.")
    Else
        MsgBox("The file could not be moved.")
    End If
End Sub
```

10. Create a file named Test.txt and place it in the C:\Tmp directory on your hard drive. Create the Tmp directory if necessary.
11. Press F5 to start the application. The main form appears.
12. Click **Button2**. The message "The file was moved successfully" is displayed if the file can be moved.

See also

- DllImportAttribute
- MarshalAsAttribute
- Declare Statement
- Auto
- Alias
- COM Interop
- Creating Prototypes in Managed Code
- Marshaling a Delegate as a Callback Method

How to: Call Windows APIs (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

This example defines and calls the `MessageBox` function in user32.dll and then passes a string to it.

Example

```
' Defines the MessageBox function.
Public Class Win32
    Declare Auto Function MessageBox Lib "user32.dll" (
        ByVal hWnd As Integer, ByVal txt As String,
        ByVal caption As String, ByVal Type As Integer
        ) As Integer
End Class

' Calls the MessageBox function.
Public Class DemoMessageBox
    Public Shared Sub Main()
        Win32.MessageBox(0, "Here's a MessageBox", "Platform Invoke Sample", 0)
    End Sub
End Class
```

Compiling the Code

This example requires:

- A reference to the System namespace.

Robust Programming

The following conditions may cause an exception:

- The method is not static, is abstract, or has been previously defined. The parent type is an interface, or the length of *name* or *dllName* is zero. (ArgumentException)
- The *name* or *dllName* is `Nothing`. (ArgumentNullException)
- The containing type has been previously created using `CreateType`. (InvalidOperationException)

See also

- A Closer Look at Platform Invoke
- Platform Invoke Examples
- Consuming Unmanaged DLL Functions
- Defining a Method with Reflection Emit
- Walkthrough: Calling Windows APIs
- COM Interop

How to: Call a Windows Function that Takes Unsigned Types (Visual Basic)

3/5/2019 • 2 minutes to read • Edit Online

If you are consuming a class, module, or structure that has members of unsigned integer types, you can access these members with Visual Basic.

To call a Windows function that takes an unsigned type

1. Use a Declare Statement to tell Visual Basic which library holds the function, what its name is in that library, what its calling sequence is, and how to convert strings when calling it.
2. In the `Declare` statement, use `UInteger`, `ULong`, `UShort`, or `Byte` as appropriate for each parameter with an unsigned type.
3. Consult the documentation for the Windows function you are calling to find the names and values of the constants it uses. Many of these are defined in the WinUser.h file.
4. Declare the necessary constants in your code. Many Windows constants are 32-bit unsigned values, and you should declare these `As UInteger`.
5. Call the function in the normal way. The following example calls the Windows function `MessageBox`, which takes an unsigned integer argument.

```
Public Class windowsMessage
    Private Declare Auto Function mb Lib "user32.dll" Alias "MessageBox" (
        ByVal hWnd As Integer,
        ByVal lpText As String,
        ByVal lpCaption As String,
        ByVal uType As UInteger) As Integer
    Private Const MB_OK As UInteger = 0
    Private Const MB_ICONEXCLAMATION As UInteger = &H30
    Private Const IDOK As UInteger = 1
    Private Const IDCLOSE As UInteger = 8
    Private Const c As UInteger = MB_OK Or MB_ICONEXCLAMATION
    Public Function messageThroughWindows() As String
        Dim r As Integer = mb(0, "Click OK if you see this!",
            "Windows API call", c)
        Dim s As String = "Windows API MessageBox returned " &
             CStr(r)& vbCrLf & "(IDOK = " & CStr(IDOK) &
             ", IDCLOSE = " & CStr(IDCLOSE) & ")"
        Return s
    End Function
End Class
```

You can test the function `messageThroughWindows` with the following code.

```
Public Sub consumeWindowsMessage()
    Dim w As New windowsMessage
    w.messageThroughWindows()
End Sub
```

Caution

The `UInteger`, `ULong`, `UShort`, and `SByte` data types are not part of the Language Independence and Language-Independent Components (CLS), so CLS-compliant code cannot consume a component that

uses them.

> **IMPORTANT**
> Making a call to unmanaged code, such as the Windows application programming interface (API), exposes your code to potential security risks.

> **IMPORTANT**
> Calling the Windows API requires unmanaged code permission, which might affect its execution in partial-trust situations. For more information, see SecurityPermission and Code Access Permissions.

See also

- Data Types
- Integer Data Type
- UInteger Data Type
- Declare Statement
- Walkthrough: Calling Windows APIs

Walkthrough: Creating COM Objects with Visual Basic

8/22/2019 • 4 minutes to read • Edit Online

When creating new applications or components, it is best to create .NET Framework assemblies. However, Visual Basic also makes it easy to expose a .NET Framework component to COM. This enables you to provide new components for earlier application suites that require COM components. This walkthrough demonstrates how to use Visual Basic to expose .NET Framework objects as COM objects, both with and without the COM class template.

The easiest way to expose COM objects is by using the COM class template. The COM class template creates a new class, and then configures your project to generate the class and interoperability layer as a COM object and register it with the operating system.

> **NOTE**
> Although you can also expose a class created in Visual Basic as a COM object for unmanaged code to use, it is not a true COM object and cannot be used by Visual Basic. For more information, see COM Interoperability in .NET Framework Applications.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To create a COM object by using the COM class template

1. Open a new Windows Application project from the **File** menu by clicking **New Project**.
2. In the **New Project** dialog box under the **Project Types** field, check that Windows is selected. Select **Class Library** from the **Templates** list, and then click **OK**. The new project is displayed.
3. Select **Add New Item** from the **Project** menu. The **Add New Item** dialog box is displayed.
4. Select **COM Class** from the **Templates** list, and then click **Add**. Visual Basic adds a new class and configures the new project for COM interop.
5. Add code such as properties, methods, and events to the COM class.
6. Select **Build ClassLibrary1** from the **Build** menu. Visual Basic builds the assembly and registers the COM object with the operating system.

Creating COM Objects without the COM Class Template

You can also create a COM class manually instead of using the COM class template. This procedure is helpful when you are working from the command line or when you want more control over how COM objects are defined.

To set up your project to generate a COM object

1. Open a new Windows Application project from the **File** menu by clicking **NewProject**.
2. In the **New Project** dialog box under the **Project Types** field, check that Windows is selected. Select **Class**

Library from the **Templates** list, and then click **OK**. The new project is displayed.

3. In **Solution Explorer**, right-click your project, and then click **Properties**. The **Project Designer** is displayed.
4. Click the **Compile** tab.
5. Select the **Register for COM Interop** check box.

To set up the code in your class to create a COM object

1. In **Solution Explorer**, double-click **Class1.vb** to display its code.
2. Rename the class to `ComClass1`.
3. Add the following constants to `ComClass1`. They will store the Globally Unique Identifier (GUID) constants that the COM objects are required to have.

```
Public Const ClassId As String = ""
Public Const InterfaceId As String = ""
Public Const EventsId As String = ""
```

4. On the **Tools** menu, click **Create Guid**. In the **Create GUID** dialog box, click **Registry Format** and then click **Copy**. Click **Exit**.
5. Replace the empty string for the `ClassId` with the GUID, removing the leading and trailing braces. For example, if the GUID provided by Guidgen is `"{2C8B0AEE-02C9-486e-B809-C780A11530FE}"` then your code should appear as follows.

```
Public Const ClassId As String = "2C8B0AEE-02C9-486e-B809-C780A11530FE"
```

6. Repeat the previous steps for the `InterfaceId` and `EventsId` constants, as in the following example.

```
Public Const InterfaceId As String = "3D8B5BA4-FB8C-5ff8-8468-11BF6BD5CF91"
Public Const EventsId As String = "2B691787-6ED7-401e-90A4-B3B9C0360E31"
```

> **NOTE**
> Make sure that the GUIDs are new and unique; otherwise, your COM component could conflict with other COM components.

7. Add the `ComClass` attribute to `ComClass1`, specifying the GUIDs for the Class ID, Interface ID, and Events ID as in the following example:

```
<ComClass(ComClass1.ClassId, ComClass1.InterfaceId, ComClass1.EventsId)>
Public Class ComClass1
```

8. COM classes must have a parameterless `Public Sub New()` constructor, or the class will not register correctly. Add a parameterless constructor to the class:

```
Public Sub New()
    MyBase.New()
End Sub
```

9. Add properties, methods, and events to the class, ending it with an `End Class` statement. Select **Build**

Solution from the **Build** menu. Visual Basic builds the assembly and registers the COM object with the operating system.

> **NOTE**
>
> The COM objects you generate with Visual Basic cannot be used by other Visual Basic applications because they are not true COM objects. Attempts to add references to such COM objects will raise an error. For details, see COM Interoperability in .NET Framework Applications.

See also

- ComClassAttribute
- COM Interop
- Walkthrough: Implementing Inheritance with COM Objects
- #Region Directive
- COM Interoperability in .NET Framework Applications
- Troubleshooting Interoperability

Troubleshooting Interoperability (Visual Basic)

8/22/2019 • 9 minutes to read • Edit Online

When you interoperate between COM and the managed code of the .NET Framework, you may encounter one or more of the following common issues.

Interop Marshaling

At times, you may have to use data types that are not part of the .NET Framework. Interop assemblies handle most of the work for COM objects, but you may have to control the data types that are used when managed objects are exposed to COM. For example, structures in class libraries must specify the `BStr` unmanaged type on strings sent to COM objects created by Visual Basic 6.0 and earlier versions. In such cases, you can use the MarshalAsAttribute attribute to cause managed types to be exposed as unmanaged types.

Exporting Fixed-Length Strings to Unmanaged Code

In Visual Basic 6.0 and earlier versions, strings are exported to COM objects as sequences of bytes without a null termination character. For compatibility with other languages, Visual Basic .NET includes a termination character when exporting strings. The best way to address this incompatibility is to export strings that lack the termination character as arrays of `Byte` or `Char`.

Exporting Inheritance Hierarchies

Managed class hierarchies flatten out when exposed as COM objects. For example, if you define a base class with a member, and then inherit the base class in a derived class that is exposed as a COM object, clients that use the derived class in the COM object will not be able to use the inherited members. Base class members can be accessed from COM objects only as instances of a base class, and then only if the base class is also created as a COM object.

Overloaded Methods

Although you can create overloaded methods with Visual Basic, they are not supported by COM. When a class that contains overloaded methods is exposed as a COM object, new method names are generated for the overloaded methods.

For example, consider a class that has two overloads of the `Synch` method. When the class is exposed as a COM object, the new generated method names could be `Synch` and `Synch_2`.

The renaming can cause two problems for consumers of the COM object.

1. Clients might not expect the generated method names.
2. The generated method names in the class exposed as a COM object can change when new overloads are added to the class or its base class. This can cause versioning problems.

To solve both problems, give each method a unique name, instead of using overloading, when you develop objects that will be exposed as COM objects.

Use of COM Objects Through Interop Assemblies

You use interop assemblies almost as if they are managed code replacements for the COM objects they represent.

However, because they are wrappers and not actual COM objects, there are some differences between using interop assemblies and standard assemblies. These areas of difference include the exposure of classes, and data types for parameters and return values.

Classes Exposed as Both Interfaces and Classes

Unlike classes in standard assemblies, COM classes are exposed in interop assemblies as both an interface and a class that represents the COM class. The interface's name is identical to that of the COM class. The name of the interop class is the same as that of the original COM class, but with the word "Class" appended. For example, suppose you have a project with a reference to an interop assembly for a COM object. If the COM class is named `MyComClass`, IntelliSense and the Object Browser show an interface named `MyComClass` and a class named `MyComClassClass`.

Creating Instances of a .NET Framework Class

Generally, you create an instance of a .NET Framework class using the `New` statement with a class name. Having a COM class represented by an interop assembly is the one case in which you can use the `New` statement with an interface. Unless you are using the COM class with an `Inherits` statement, you can use the interface just as you would a class. The following code demonstrates how to create a `Command` object in a project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object:

```
Dim cmd As New ADODB.Command
```

However, if you are using the COM class as the base for a derived class, you must use the interop class that represents the COM class, as in the following code:

```
Class DerivedCommand
    Inherits ADODB.CommandClass
End Class
```

> **NOTE**
> Interop assemblies implicitly implement interfaces that represent COM classes. You should not try to use the `Implements` statement to implement these interfaces or an error will result.

Data Types for Parameters and Return Values

Unlike members of standard assemblies, interop assembly members may have data types that differ from those used in the original object declaration. Although interop assemblies implicitly convert COM types to compatible common language runtime types, you should pay attention to the data types that are used by both sides to prevent runtime errors. For example, in COM objects created in Visual Basic 6.0 and earlier versions, values of type `Integer` assume the .NET Framework equivalent type, `Short`. It is recommended that you use the Object Browser to examine the characteristics of imported members before you use them.

Module level COM methods

Most COM objects are used by creating an instance of a COM class using the `New` keyword and then calling methods of the object. One exception to this rule involves COM objects that contain `AppObj` or `GlobalMultiUse` COM classes. Such classes resemble module level methods in Visual Basic .NET classes. Visual Basic 6.0 and earlier versions implicitly create instances of such objects for you the first time that you call one of their methods. For example, in Visual Basic 6.0 you can add a reference to the Microsoft DAO 3.6 Object Library and call the

`DBEngine` method without first creating an instance:

```
Dim db As DAO.Database
' Open the database.
Set db = DBEngine.OpenDatabase("C:\nwind.mdb")
' Use the database object.
```

Visual Basic .NET requires that you always create instances of COM objects before you can use their methods. To use these methods in Visual Basic, declare a variable of the desired class and use the new keyword to assign the object to the object variable. The `Shared` keyword can be used when you want to make sure that only one instance of the class is created.

```
' Class level variable.
Shared DBEngine As New DAO.DBEngine

Sub DAOOpenRecordset()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    ' Open the database.
    db = DBEngine.OpenDatabase("C:\nwind.mdb")

    ' Open the Recordset.
    rst = db.OpenRecordset(
        "SELECT * FROM Customers WHERE Region = 'WA'",
        DAO.RecordsetTypeEnum.dbOpenForwardOnly,
        DAO.RecordsetOptionEnum.dbReadOnly)
    ' Print the values for the fields in the debug window.
    For Each fld In rst.Fields
        Debug.WriteLine(fld.Value.ToString & ";")
    Next
    Debug.WriteLine("")
    ' Close the Recordset.
    rst.Close()
End Sub
```

Unhandled Errors in Event Handlers

One common interop problem involves errors in event handlers that handle events raised by COM objects. Such errors are ignored unless you specifically check for errors using `On Error` or `Try...Catch...Finally` statements. For example, the following example is from a Visual Basic .NET project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object.

```
' To use this example, add a reference to the
'     Microsoft ActiveX Data Objects 2.8 Library
' from the COM tab of the project references page.
Dim WithEvents cn As New ADODB.Connection
Sub ADODBConnect()
    cn.ConnectionString =
    "Provider=Microsoft.Jet.OLEDB.4.0;" &
    "Data Source=C:\NWIND.MDB"
    cn.Open()
    MsgBox(cn.ConnectionString)
End Sub

Private Sub Form1_Load(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles MyBase.Load

    ADODBConnect()
End Sub

Private Sub cn_ConnectComplete(
    ByVal pError As ADODB.Error,
    ByRef adStatus As ADODB.EventStatusEnum,
    ByVal pConnection As ADODB.Connection) Handles cn.ConnectComplete

    '  This is the event handler for the cn_ConnectComplete event raised
    '  by the ADODB.Connection object when a database is opened.
    Dim x As Integer = 6
    Dim y As Integer = 0
    Try
        x = CInt(x / y) ' Attempt to divide by zero.
        ' This procedure would fail silently without exception handling.
    Catch ex As Exception
        MsgBox("There was an error: " & ex.Message)
    End Try
End Sub
```

This example raises an error as expected. However, if you try the same example without the `Try...Catch...Finally` block, the error is ignored as if you used the `OnError Resume Next` statement. Without error handling, the division by zero silently fails. Because such errors never raise unhandled exception errors, it is important that you use some form of exception handling in event handlers that handle events from COM objects.

Understanding COM interop errors

Without error handling, interop calls often generate errors that provide little information. Whenever possible, use structured error handling to provide more information about problems when they occur. This can be especially helpful when you debug applications. For example:

```
Try
    ' Place call to COM object here.
Catch ex As Exception
    ' Display information about the failed call.
End Try
```

You can find information such as the error description, HRESULT, and the source of COM errors by examining the contents of the exception object.

ActiveX Control Issues

Most ActiveX controls that work with Visual Basic 6.0 work with Visual Basic .NET without trouble. The main exceptions are container controls, or controls that visually contain other controls. Some examples of older controls that do not work correctly with Visual Studio are as follows:

- Microsoft Forms 2.0 Frame control
- Up-Down control, also known as the spin control
- Sheridan Tab Control

There are only a few workarounds for unsupported ActiveX control problems. You can migrate existing controls to Visual Studio if you own the original source code. Otherwise, you can check with software vendors for updated .NET-compatible versions of controls to replace unsupported ActiveX controls.

Passing ReadOnly Properties of Controls ByRef

Visual Basic .NET sometimes raises COM errors such as, "Error 0x800A017F CTL_E_SETNOTSUPPORTED", when you pass `ReadOnly` properties of some older ActiveX controls as `ByRef` parameters to other procedures. Similar procedure calls from Visual Basic 6.0 do not raise an error, and the parameters are treated as if you passed them by value. The Visual Basic .NET error message indicates that you are trying to change a property that does not have a property `Set` procedure.

If you have access to the procedure being called, you can prevent this error by using the `ByVal` keyword to declare parameters that accept `ReadOnly` properties. For example:

```
Sub ProcessParams(ByVal c As Object)
    'Use the arguments here.
End Sub
```

If you do not have access to the source code for the procedure being called, you can force the property to be passed by value by adding an extra set of brackets around the calling procedure. For example, in a project that has a reference to the Microsoft ActiveX Data Objects 2.8 Library COM object, you can use:

```
Sub PassByVal(ByVal pError As ADODB.Error)
    ' The extra set of parentheses around the arguments
    ' forces them to be passed by value.
    ProcessParams((pError.Description))
End Sub
```

Deploying Assemblies That Expose Interop

Deploying assemblies that expose COM interfaces presents some unique challenges. For example, a potential problem occurs when separate applications reference the same COM assembly. This situation is common when a new version of an assembly is installed and another application is still using the old version of the assembly. If you uninstall an assembly that shares a DLL, you can unintentionally make it unavailable to the other assemblies.

To avoid this problem, you should install shared assemblies to the Global Assembly Cache (GAC) and use a MergeModule for the component. If you cannot install the application in the GAC, it should be installed to CommonFilesFolder in a version-specific subdirectory.

Assemblies that are not shared should be located side by side in the directory with the calling application.

See also

- MarshalAsAttribute
- COM Interop
- Tlbimp.exe (Type Library Importer)
- Tlbexp.exe (Type Library Exporter)
- Walkthrough: Implementing Inheritance with COM Objects

- Inherits Statement
- Global Assembly Cache

COM Interoperability in .NET Framework Applications (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

When you want to use COM objects and .NET Framework objects in the same application, you need to address the differences in how the objects exist in memory. A .NET Framework object is located in managed memory—the memory controlled by the common language runtime—and may be moved by the runtime as needed. A COM object is located in unmanaged memory and is not expected to move to another memory location. Visual Studio and the .NET Framework provide tools to control the interaction of these managed and unmanaged components. For more information about managed code, see Common Language Runtime.

In addition to using COM objects in .NET applications, you may also want to use Visual Basic to develop objects accessible from unmanaged code through COM.

The links on this page provide details on the interactions between COM and .NET Framework objects.

Related sections

COM Interop	Provides links to topics covering COM interoperability in Visual Basic, including COM objects, ActiveX controls, Win32 DLLs, managed objects, and inheritance of COM objects.
Interoperating with Unmanaged Code	Briefly describes some of the interaction issues between managed and unmanaged code, and provides links for further study.
COM Wrappers	Discusses runtime callable wrappers, which allow managed code to call COM methods, and COM callable wrappers, which allow COM clients to call .NET object methods.
Advanced COM Interoperability	Provides links to topics covering COM interoperability with respect to wrappers, exceptions, inheritance, threading, events, conversions, and marshaling.
Tlbimp.exe (Type Library Importer)	Discusses the tool you can use to convert the type definitions found within a COM type library into equivalent definitions in a common language runtime assembly.

Walkthrough: Implementing Inheritance with COM Objects (Visual Basic)

7/30/2019 • 5 minutes to read • Edit Online

You can derive Visual Basic classes from `Public` classes in COM objects, even those created in earlier versions of Visual Basic. The properties and methods of classes inherited from COM objects can be overridden or overloaded just as properties and methods of any other base class can be overridden or overloaded. Inheritance from COM objects is useful when you have an existing class library that you do not want to recompile.

The following procedure shows how to use Visual Basic 6.0 to create a COM object that contains a class, and then use it as a base class.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To build the COM object that is used in this walkthrough

1. In Visual Basic 6.0, open a new ActiveX DLL project. A project named `Project1` is created. It has a class named `Class1`.
2. In the **Project Explorer**, right-click **Project1**, and then click **Project1 Properties**. The **Project Properties** dialog box is displayed.
3. On the **General** tab of the **Project Properties** dialog box, change the project name by typing `ComObject1` in the **Project Name** field.
4. In the **Project Explorer**, right-click `Class1`, and then click **Properties**. The **Properties** window for the class is displayed.
5. Change the `Name` property to `MathFunctions`.
6. In the **Project Explorer**, right-click `MathFunctions`, and then click **View Code**. The **Code Editor** is displayed.
7. Add a local variable to hold the property value:

```
' Local variable to hold property value
Private mvarProp1 As Integer
```

8. Add Property `Let` and Property `Get` property procedures:

```
Public Property Let Prop1(ByVal vData As Integer)
   'Used when assigning a value to the property.
   mvarProp1 = vData
End Property
Public Property Get Prop1() As Integer
   'Used when retrieving a property's value.
   Prop1 = mvarProp1
End Property
```

9. Add a function:

```
Function AddNumbers(
   ByVal SomeNumber As Integer,
   ByVal AnotherNumber As Integer) As Integer

   AddNumbers = SomeNumber + AnotherNumber
End Function
```

10. Create and register the COM object by clicking **Make ComObject1.dll** on the **File** menu.

> **NOTE**
>
> Although you can also expose a class created with Visual Basic as a COM object, it is not a true COM object and cannot be used in this walkthrough. For details, see COM Interoperability in .NET Framework Applications.

Interop Assemblies

In the following procedure, you will create an interop assembly, which acts as a bridge between unmanaged code (such as a COM object) and the managed code Visual Studio uses. The interop assembly that Visual Basic creates handles many of the details of working with COM objects, such as *interop marshaling*, the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. The reference in the Visual Basic application points to the interop assembly, not the actual COM object.

To use a COM object with Visual Basic 2005 and later versions

1. Open a new Visual Basic Windows Application project.
2. On the **Project** menu, click **Add Reference**.

   The **Add Reference** dialog box is displayed.
3. On the **COM** tab, double-click `ComObject1` in the **Component Name** list and click **OK**.
4. On the **Project** menu, click **Add New Item**.

   The **Add New Item** dialog box is displayed.
5. In the **Templates** pane, click **Class**.

   The default file name, `Class1.vb`, appears in the **Name** field. Change this field to MathClass.vb and click **Add**. This creates a class named `MathClass`, and displays its code.
6. Add the following code to the top of `MathClass` to inherit from the COM class.

```
' The inherited class is called MathFunctions in the base class,
' but the interop assembly appends the word Class to the name.
Inherits ComObject1.MathFunctionsClass
```

7. Overload the public method of the base class by adding the following code to `MathClass`:

```
'  This method overloads the method AddNumbers from the base class.
Overloads Function AddNumbers(
    ByVal SomeNumber As Integer,
    ByVal AnotherNumber As Integer) As Integer

    Return SomeNumber + AnotherNumber
End Function
```

8. Extend the inherited class by adding the following code to `MathClass`:

```
'  The following function extends the inherited class.
Function SubtractNumbers(
    ByVal SomeNumber As Integer,
    ByVal AnotherNumber As Integer) As Integer

    Return AnotherNumber - SomeNumber
End Function
```

The new class inherits the properties of the base class in the COM object, overloads a method, and defines a new method to extend the class.

To test the inherited class

1. Add a button to your startup form, and then double-click it to view its code.
2. In the button's `Click` event handler procedure, add the following code to create an instance of `MathClass` and call the overloaded methods:

```
Dim Result1 As Short
Dim Result2 As Integer
Dim Result3 As Integer
Dim MathObject As New MathClass
Result1 = MathObject.AddNumbers(4S, 2S) ' Add two Shorts.
Result2 = MathObject.AddNumbers(4, 2) 'Add two Integers.
Result3 = MathObject.SubtractNumbers(2, 4) ' Subtract 2 from 4.
MathObject.Prop1 = 6 ' Set an inherited property.

MsgBox("Calling the AddNumbers method in the base class " &
       "using Short type numbers 4 and 2 = " & Result1)
MsgBox("Calling the overloaded AddNumbers method using " &
       "Integer type numbers 4 and 2 = " & Result2)
MsgBox("Calling the SubtractNumbers method " &
       "subtracting 2 from 4 = " & Result3)
MsgBox("The value of the inherited property is " &
        MathObject.Prop1)
```

3. Run the project by pressing F5.

When you click the button on the form, the `AddNumbers` method is first called with `Short` data type numbers, and Visual Basic chooses the appropriate method from the base class. The second call to `AddNumbers` is directed to the overload method from `MathClass`. The third call calls the `SubtractNumbers` method, which extends the class. The property in the base class is set, and the value is displayed.

Next Steps

You may have noticed that the overloaded `AddNumbers` function appears to have the same data type as the method inherited from the base class of the COM object. This is because the arguments and parameters of the base class method are defined as 16-bit integers in Visual Basic 6.0, but they are exposed as 16-bit integers of type `Short` in later versions of Visual Basic. The new function accepts 32-bit integers, and overloads the base class function.

When working with COM objects, make sure that you verify the size and data types of parameters. For example, when you are using a COM object that accepts a Visual Basic 6.0 collection object as an argument, you cannot provide a collection from a later version of Visual Basic.

Properties and methods inherited from COM classes can be overridden, meaning that you can declare a local property or method that replaces a property or method inherited from a base COM class. The rules for overriding

inherited COM properties are similar to the rules for overriding other properties and methods with the following exceptions:

- If you override any property or method inherited from a COM class, you must override all the other inherited properties and methods.
- Properties that use `ByRef` parameters cannot be overridden.

See also

- COM Interoperability in .NET Framework Applications
- Inherits Statement
- Short Data Type

Visual Basic Language Reference

8/24/2018 • 2 minutes to read • Edit Online

This section provides reference information for various aspects of the Visual Basic language.

In This Section

Typographic and Code Conventions
Summarizes the way that keywords, placeholders, and other elements of the language are formatted in the Visual Basic documentation.

Visual Basic Runtime Library Members
Lists the classes and modules of the Microsoft.VisualBasic namespace, with links to their member functions, methods, properties, constants, and enumerations.

Keywords
Lists all Visual Basic keywords and provides links to more information.

Attributes (Visual Basic)
Documents the attributes available in Visual Basic.

Constants and Enumerations
Documents the constants and enumerations available in Visual Basic.

Data Types
Documents the data types available in Visual Basic.

Directives
Documents the compiler directives available in Visual Basic.

Functions
Documents the run-time functions available in Visual Basic.

Modifiers
Lists the Visual Basic run-time modifiers and provides links to more information.

Modules
Documents the modules available in Visual Basic and their members.

Nothing
Describes the default value of any data type.

Objects
Documents the objects available in Visual Basic and their members.

Operators
Documents the operators available in Visual Basic.

Properties
Documents the properties available in Visual Basic.

Queries
Provides reference information about using Language-Integrated Query (LINQ) expressions in your code.

Statements

Documents the declaration and executable statements available in Visual Basic.

XML Comment Tags
Describes the documentation comments for which IntelliSense is provided in the Visual Basic Code Editor.

XML Axis Properties
Provides links to information about using XML axis properties to access XML directly in your code.

XML Literals
Provides links to information about using XML literals to incorporate XML directly in your code.

Error Messages
Provides a listing of Visual Basic compiler and run-time error messages and help on how to handle them.

Related Sections

Visual Basic
Provides comprehensive help on all areas of the Visual Basic language.

Visual Basic Command-Line Compiler
Describes how to use the command-line compiler as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE).

Select the Visual Basic language version

6/20/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler defaults to the latest major version of the language that has been released. You may choose to compile any project using a new point release of the language. Choosing a newer version of the language enables your project to make use of the latest language features. In other scenarios, you may need to validate that a project compiles cleanly when using an older version of the language.

This capability decouples the decision to install new versions of the SDK and tools in your development environment from the decision to incorporate new language features in a project. You can install the latest SDK and tools on your build machine. Each project can be configured to use a specific version of the language for its build.

There are three ways to set the language version:

- Manually edit your **.vbproj** file
- Set the language version for multiple projects in a subdirectory
- Configure the `-langversion` compiler option

Edit the vbproj file

You can set the language version in your **.vbproj** file. Add the following element:

```
<PropertyGroup>
   <LangVersion>latest</LangVersion>
</PropertyGroup>
```

The value `latest` uses the latest minor version of the Visual Basic language. Valid values are:

VALUE	MEANING
default	The compiler accepts all valid language syntax from the latest major version that it can support.
9	The compiler accepts only syntax that is included in Visual Basic 9.0 or lower.
10	The compiler accepts only syntax that is included in Visual Basic 10.0 or lower.
11	The compiler accepts only syntax that is included in Visual Basic 11.0 or lower.
12	The compiler accepts only syntax that is included in Visual Basic 12.0 or lower.
14	The compiler accepts only syntax that is included in Visual Basic 14.0 or lower.
15	The compiler accepts only syntax that is included in Visual Basic 15.0 or lower.

VALUE	MEANING
15.3	The compiler accepts only syntax that is included in Visual Basic 15.3 or lower.
15.5	The compiler accepts only syntax that is included in Visual Basic 15.5 or lower.
15.8	The compiler accepts only syntax that is included in Visual Basic 15.8 or lower.
latest	The compiler accepts all valid language syntax that it can support.

The special strings `default` and `latest` resolve to the latest major and minor language versions installed on the build machine, respectively.

Configure multiple projects

You can create a **Directory.build.props** file that contains the `<LangVersion>` element to configure multiple directories. You typically do that in your solution directory. Add the following to a **Directory.build.props** file in your solution directory:

```
<Project>
 <PropertyGroup>
   <LangVersion>15.5</LangVersion>
 </PropertyGroup>
</Project>
```

Now, builds in every subdirectory of the directory containing that file will use Visual Basic version 15.5 syntax. For more information, see the article on Customize your build.

Set the langversion compiler option

You can use the `-langversion` command-line option. For more information, see the article on the -langversion compiler option. You can see a list of the valid values by typing `vbc -langversion:?` .

Typographic and Code Conventions (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic documentation uses the following typographic and code conventions.

Typographic Conventions

EXAMPLE	DESCRIPTION
`Sub`, `If`, `ChDir`, `Print`, `True`, `Debug`	Language-specific keywords and runtime members have initial uppercase letters and are formatted as shown in this example.
SmallProject, **ButtonCollection**	Words and phrases you are instructed to type are formatted as shown in this example.
Module Statement	Links you can click to go to another Help page are formatted as shown in this example.
object, *variableName*, `argumentList`	Placeholders for information that you supply are formatted as shown in this example.
[Shadows], [*expressionList*]	In syntax, optional items are enclosed in brackets.
{ `Public` | `Friend` | `Private` }	In syntax, when you must make a choice between two or more items, the items are enclosed in braces and separated by vertical bars. You must select one, and only one, of the items.
[`Protected` | `Friend`]	In syntax, when you have the option of selecting between two or more items, the items are enclosed in square brackets and separated by vertical bars. You can select any combination of the items, or no item.
[{ `ByVal` | `ByRef` }]	In syntax, when you can select no more than one item, but you can also omit the items completely, the items are enclosed in square brackets surrounded by braces and separated by vertical bars.
*memberName*1, *memberName*2, *memberName*3	Multiple instances of the same placeholder are differentiated by subscripts, as shown in the example.
memberName1 ... *memberNameN*	In syntax, an ellipsis (...) is used to indicate an indefinite number of items of the kind immediately in front of the ellipsis. In code, ellipses signify code omitted for the sake of clarity.
ESC, ENTER	Key names and key sequences on the keyboard appear in all uppercase letters.

EXAMPLE	DESCRIPTION
ALT+F1	When plus signs (+) appear between key names, you must hold down one key while pressing the other. For example, ALT+F1 means hold down the ALT key while pressing the F1 key.

Code Conventions

EXAMPLE	DESCRIPTION
`sampleString = "Hello, world!"`	Code samples appear in a fixed-pitch font and are formatted as shown in this example.
The previous statement sets the value of `sampleString` to "Hello, world!"	Code elements in explanatory text appear in a fixed-pitch font, as shown in this example.

`' This is a comment.`
 	
`REM This is also a comment.`
	Code comments are introduced by an apostrophe (') or the REM keyword.

`sampleVar = "This is an " _`
 	
`& "example" _`
 	
`& " of how to continue code."`
	A space followed by an underscore (_) at the end of a line indicates that the statement continues on the following line.

See also

- Visual Basic Language Reference
- Keywords
- Visual Basic Runtime Library Members
- Visual Basic Naming Conventions
- How to: Break and Combine Statements in Code
- Comments in Code

Visual Basic Runtime Library Members

4/2/2019 • 2 minutes to read • Edit Online

The `Microsoft.VisualBasic` namespace contains the classes, modules, constants, and enumerations that constitute the Visual Basic runtime library. These library members provide procedures, properties, and constant values you can use in your code. Each module and class represents a particular category of functionality.

Microsoft.VisualBasic.Collection Class

Add	Clear	Contains	Count
GetEnumerator	Item[String]	Remove	

Microsoft.VisualBasic.ComClassAttribute Class

ClassID	EventID	InterfaceID	InterfaceShadows

Microsoft.VisualBasic.ControlChars Class

Back	Cr	CrLf	FormFeed
Lf	NewLine	NullChar	Quote
Tab	VerticalTab		

Microsoft.VisualBasic.Constants Class

vbAbort	vbAbortRetryIgnore	vbApplicationModal	vbArchive
vbArray	vbBack	vbBinaryCompare	vbBoolean
vbByte	vbCancel	vbCr	vbCritical
vbCrLf	vbCurrency	vbDate	vbDecimal
vbDefaultButton1	vbDefaultButton2	vbDefaultButton3	vbDirectory
vbDouble	vbEmpty	vbExclamation	vbFalse
vbFirstFourDays	vbFirstFullWeek	vbFirstJan1	vbFormFeed

vbFriday	vbGeneralDate	vbGet	vbHidden
vbHide	vbHiragana	vbIgnore	vbInformation
vbInteger	vbKatakana	vbLet	vbLf
vbLinguisticCasing	vbLong	vbLongDate	vbLongTime
vbLowerCase	vbMaximizedFocus	vbMethod	vbMinimizedFocus
vbMinimizedNoFocus	vbMonday	vbMsgBoxHelp	vbMsgBoxRight
vbMsgBoxRtlReading	vbMsgBoxSetForeground	vbNarrow	vbNewLine
vbNo	vbNormal	vbNormalFocus	vbNormalNoFocus
vbNull	vbNullChar	vbNullString	vbObject
vbObjectError	vbOK	vbOKCancel	vbOKOnly
vbProperCase	vbQuestion	vbReadOnly	vbRetry
vbRetryCancel	vbSaturday	vbSet	vbShortDate
vbShortTime	vbSimplifiedChinese	vbSingle	vbString
vbSunday	vbSystem	vbSystemModal	vbTab
vbTextCompare	vbThursday	vbTraditionalChinese	vbTrue
vbTuesday	vbUpperCase	vbUseDefault	vbUserDefinedType
vbUseSystem	vbUseSystemDayOfWeek	vbVariant	vbVerticalTab
vbVolume	vbWednesday	vbWide	vbYes
vbYesNo	vbYesNoCancel		

Microsoft.VisualBasic.Conversion Module

ErrorToString	Fix	Hex	Int
Oct	Str	Val	

Microsoft.VisualBasic.DateAndTime Module

DateAdd	DateDiff	DatePart	DateSerial

DateString	DateValue	Day	Hour
Minute	Month	MonthName	Now
Second	TimeOfDay	Timer	TimeSerial
TimeString	TimeValue	Today	Weekday
WeekdayName	Year		

Microsoft.VisualBasic.ErrObject Class

Clear	Description	Erl	GetException
HelpContext	HelpFile	LastDllError	Number
Raise	Raise		

Microsoft.VisualBasic.FileSystem Module

ChDir	ChDrive	CurDir	Dir
EOF	FileAttr	FileClose	FileCopy
FileDateTime	FileGet	FileGetObject	FileLen
FileOpen	FilePut	FilePutObject	FileWidth
FreeFile	GetAttr	Input	InputString
Kill	LineInput	Loc	Lock
LOF	MkDir	Print	PrintLine
Rename	Reset	RmDir	Seek
SetAttr	SPC	TAB	Unlock
Write	WriteLine		

Microsoft.VisualBasic.Financial Module

DDB	FV	IPmt	IRR
MIRR	NPer	NPV	Pmt

PPmt	PV	Rate	SLN
SYD			

Microsoft.VisualBasic.Globals Module

ScriptEngine	ScriptEngineBuildVersion	ScriptEngineMajorVersion	ScriptEngineMinorVersion

Microsoft.VisualBasic.HideModuleNameAttribute Class

HideModuleNameAttribute			

Microsoft.VisualBasic.Information Module

Erl	Err	IsArray	IsDate
IsDBNull	IsError	IsNothing	IsNumeric
IsReference	LBound	QBColor	RGB
SystemTypeName	TypeName	UBound	VarType
VbTypeName			

Microsoft.VisualBasic.Interaction Module

AppActivate	Beep	CallByName	Choose
Command	CreateObject	DeleteSetting	Environ
GetAllSettings	GetObject	GetSetting	IIf
InputBox	MsgBox	Partition	SaveSetting
Shell	Switch		

Microsoft.VisualBasic.MyGroupCollectionAttribute Class

CreateMethod	DefaultInstanceAlias	DisposeMethod	MyGroupName

Microsoft.VisualBasic.Strings Module

Asc	Asc	Chr	ChrW
Filter	Format	FormatCurrency	FormatDateTime
FormatNumber	FormatPercent	GetChar	InStr
InStrRev	Join	LCase	Left
Len	LSet	LTrim	Mid
Replace	Right	RSet	RTrim
Space	Split	StrComp	StrConv
StrDup	StrReverse	Trim	UCase

Microsoft.VisualBasic.VBFixedArrayAttribute Class

Bounds	Length		

Microsoft.VisualBasic.VBFixedStringAttribute Class

Length			

Microsoft.VisualBasic.VbMath Module

Randomize	Rnd		

Microsoft.VisualBasic Constants and Enumerations

The `Microsoft.VisualBasic` namespace provides constants and enumerations as part of the Visual Basic run-time library. You can use these constant values in your code. Each enumeration represents a particular category of functionality. For more information, see Constants and Enumerations.

See also

- Constants and Enumerations
- Keywords

Keywords (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

The following tables list all Visual Basic language keywords.

Reserved Keywords

The following keywords are *reserved*, which means that you cannot use them as names for programming elements such as variables or procedures. You can bypass this restriction by enclosing the name in brackets (`[]`). For more information, see "Escaped Names" in Declared Element Names.

> **NOTE**
> We do not recommend that you use escaped names, because it can make your code hard to read, and it can lead to subtle errors that can be difficult to find.

AddHandler	AddressOf	Alias	And
AndAlso	As	Boolean	ByRef
Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar
CDate	CDbl	CDec	Char
CInt	Class Constraint	Class Statement	CLng
CObj	Const	Continue	CSByte
CShort	CSng	CStr	CType
CUInt	CULng	CUShort	Date
Decimal	Declare	Default	Delegate
Dim	DirectCast	Do	Double
Each	Else	ElseIf	End Statement
End <keyword>	`EndIf`	Enum	Erase
Error	Event	Exit	False
Finally	For (in For...Next)	For Each...Next	Friend
Function	Get	GetType	GetXMLNamespace

Global	`GoSub`	GoTo	Handles
If	If()	Implements	Implements Statement
Imports (.NET Namespace and Type)	Imports (XML Namespace)	In	In (Generic Modifier)
Inherits	Integer	Interface	Is
IsNot	Let	Lib	Like
Long	Loop	Me	Mod
Module	Module Statement	MustInherit	MustOverride
MyBase	MyClass	Namespace	Narrowing
New Constraint	New Operator	Next	Next (in Resume)
Not	Nothing	NotInheritable	NotOverridable
Object	Of	On	Operator
Option	Optional	Or	OrElse
Out (Generic Modifier)	Overloads	Overridable	Overrides
ParamArray	Partial	Private	Property
Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	RemoveHandler	Resume
Return	SByte	Select	Set
Shadows	Shared	Short	Single
Static	Step	Stop	String
Structure Constraint	Structure Statement	Sub	SyncLock
Then	Throw	To	True
Try	TryCast	TypeOf...Is	UInteger
ULong	UShort	Using	`Variant`
`Wend`	When	While	Widening
With	WithEvents	WriteOnly	Xor

#Const	#Else	#ElseIf	#End
#If	=	&	&=
*	*=	/	/=
\	\=	^	^=
+	+=	-	-=
>> Operator	>>= Operator	<<	<<=

NOTE

`EndIf`, `GoSub`, `Variant`, and `Wend` are retained as reserved keywords, although they are no longer used in Visual Basic. The meaning of the `Let` keyword has changed. `Let` is now used in LINQ queries. For more information, see Let Clause.

Unreserved Keywords

The following keywords are not reserved, which means you can use them as names for your programming elements. However, doing this is not recommended, because it can make your code hard to read and can lead to subtle errors that can be difficult to find.

Aggregate	Ansi	Assembly	Async
Auto	Await	Binary	Compare
Custom	Distinct	Equals	Explicit
From	Group By	Group Join	Into
IsFalse	IsTrue	Iterator	Join
Key	Mid	Off	Order By
Preserve	Skip	Skip While	Strict
Take	Take While	Text	Unicode
Until	Where	Yield	#ExternalSource
#Region			

Related Topics

TITLE	DESCRIPTION
Arrays Summary	Lists language elements that are used to create, define, and use arrays.
Collection Object Summary	Lists language elements that are used for collections.
Control Flow Summary	Lists statements that are used for looping and controlling procedure flow.
Conversion Summary	Lists functions that are used to convert numbers, dates, times, and strings.
Data Types Summary	Lists data types. Also lists functions that are used to convert between data types and verify data types.
Dates and Times Summary	Lists language elements that are used for dates and times.
Declarations and Constants Summary	Lists statements that are used to declare variables, constants, classes, modules, and other programming elements. Also lists language elements that are used to obtain object information, handle events, and implement inheritance.
Directories and Files Summary	Lists functions that are used to control the file system and to process files.
Errors Summary	Lists language elements that are used to catch and return run-time error values.
Financial Summary	Lists functions that are used to perform financial calculations.
Input and Output Summary	Lists functions that are used to read from and write to files, manage files, and print output.
Information and Interaction Summary	Lists functions that are used to run other programs, obtain command-line arguments, manipulate COM objects, retrieve color information, and use control dialog boxes.
Math Summary	Lists functions that are used to perform trigonometric and other mathematical calculations.
My Reference	Lists the objects contained in `My`, a feature that provides access to frequently used methods, properties, and events of the computer on which the application is running, the current application, the application's resources, the application's settings, and so on.
Operators Summary	Lists assignment and comparison expressions and other operators.
Registry Summary	Lists functions that are used to read, save, and delete program settings.
String Manipulation Summary	Lists functions that are used to manipulate strings.

See also

- Visual Basic Runtime Library Members

Arrays Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Verify an array.	IsArray
Declare and initialize an array.	Dim, Private, Public, ReDim
Find the limits of an array.	LBound, UBound
Reinitialize an array	Erase, ReDim

See also

- Keywords
- Visual Basic Runtime Library Members

Collection Object Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Create a `Collection` object.	Collection
Add an item to a collection.	Add
Remove an object from a collection.	Remove
Reference an item in a collection.	Item[String]
Return a reference to an IEnumerator interface.	IEnumerable.GetEnumerator

See also

- Keywords
- Visual Basic Runtime Library Members

Control Flow Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Branch.	GoTo, On Error
Exit or pause the program.	End, Exit, Stop
Loop.	Do...Loop, For...Next, For Each...Next, While...End While, With
Make decisions.	Choose, If...Then...Else, Select Case, Switch
Use procedures.	Call, Function, Property, Sub

See also

- Keywords
- Visual Basic Runtime Library Members

Conversion Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Convert ANSI value to string.	Chr, ChrW
Convert string to lowercase or uppercase.	Format, LCase,UCase
Convert date to serial number.	DateSerial, DateValue
Convert decimal number to other bases.	Hex, Oct
Convert number to string.	Format, Str
Convert one data type to another.	CBool, CByte, CDate, CDbl, CDec, CInt, CLng, CSng, CShort, CStr, CType, Fix, Int
Convert date to day, month, weekday, or year.	Day, Month, Weekday, Year
Convert time to hour, minute, or second.	Hour, Minute, Second
Convert string to ASCII value.	Asc, AscW
Convert string to number.	Val
Convert time to serial number.	TimeSerial, TimeValue

See also

- Keywords
- Visual Basic Runtime Library Members

Data Types Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Convert between data types	CBool, CByte, CChar, CDate, CDbl, CDec, CInt, CLng, CObj, CShort, CSng, CStr, Fix, Int
Set intrinsic data types	Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Object, Short, Single, String
Verify data types	IsArray, IsDate, IsDBNull, IsError, IsNothing, IsNumeric, IsReference

See also

- Keywords
- Visual Basic Runtime Library Members

Dates and Times Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Get the current date or time.	Now, Today, TimeOfDay
Perform date calculations.	DateAdd, DateDiff, DatePart
Return a date.	DateSerial, DateValue, MonthName, WeekdayName
Return a time.	TimeSerial, TimeValue
Set the date or time.	DateString, TimeOfDay, TimeString, Today
Time a process.	Timer

See also

- Keywords
- Visual Basic Runtime Library Members

Declarations and Constants Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Assign a value.	Get, Property
Declare variables or constants.	Const, Dim, Private, Protected, Public, Shadows, Shared, Static
Declare a class, delegate, enumeration, module, namespace, or structure.	Class, Delegate, Enum, Module, Namespace, Structure
Create objects.	CreateObject, GetObject, New
Get information about an object.	GetType, IsArray, IsDate, IsDBNull, IsError, IsNothing, IsNumeric, IsReference, SystemTypeName, TypeName, VarType, VbTypeName
Refer to the current object.	Me
Require explicit variable declarations.	Option Explicit, Option Strict
Handle events.	AddHandler, Event, RaiseEvent, RemoveHandler
Implement inheritance.	Inherits, MustInherit, MustOverride, MyBase, MyClass, New, NotInheritable, NotOverridable, Overloads, Overridable, Overrides

See also

- Keywords
- Visual Basic Runtime Library Members

Directories and Files Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

Using the `My` feature gives you greater productivity and performance in file I/O operations than using these members. For more information, see FileSystem.

ACTION	LANGUAGE ELEMENT
Change a directory or folder.	ChDir
Change the drive.	ChDrive
Copy a file.	FileCopy
Make a directory or folder.	MkDir
Remove a directory or folder.	RmDir
Rename a file, directory, or folder.	Rename
Return the current path.	CurDir
Return a file's date/time stamp.	FileDateTime
Return file, directory, or label attributes.	GetAttr
Return a file's length.	FileLen
Return a file's name or volume label.	Dir
Set attribute information for a file.	SetAttr

See also

- Keywords
- Visual Basic Runtime Library Members
- Reading from Files
- Writing to Files
- Creating, Deleting, and Moving Files and Directories
- Parsing Text Files with the TextFieldParser Object

Errors Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Generate run-time errors.	Clear, Error, Raise
Get exceptions.	GetException
Provide error information.	Err
Trap errors during run time.	On Error, Resume, Try...Catch...Finally
Provide line number of error.	Erl
Provide system error code.	LastDllError

See also

- Keywords
- Visual Basic Runtime Library Members

Financial Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Calculate depreciation.	DDB, SLN, SYD
Calculate future value.	FV
Calculate interest rate.	Rate
Calculate internal rate of return.	IRR, MIRR
Calculate number of periods.	NPer
Calculate payments.	IPmt, Pmt, PPmt
Calculate present value.	NPV, PV

See also

- Keywords
- Visual Basic Runtime Library Members

Information and Interaction Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Run other programs.	AppActivate, Shell
Call a method or property.	CallByName
Sound a beep from computer.	Beep
Provide a command-line string.	Command
Manipulate COM objects.	CreateObject, GetObject
Retrieve color information.	QBColor, RGB
Control dialog boxes	InputBox, MsgBox

See also

- Keywords
- Visual Basic Runtime Library Members

Input and Output Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Access or create a file.	FileOpen
Close files.	FileClose, Reset
Control output appearance.	Format, Print, SPC, TAB, FileWidth
Copy a file.	FileCopy
Get information about a file.	EOF, FileAttr, FileDateTime, FileLen, FreeFile, GetAttr, Loc, LOF, Seek
Get or provide information from/to the user by means of a control dialog box.	InputBox, MsgBox
Manage files.	Dir, Kill, Lock, Unlock
Read from a file.	FileGet, FileGetObject, Input, InputString, LineInput
Return length of a file.	FileLen
Set or get file attributes.	FileAttr, GetAttr, SetAttr
Set read-write position in a file.	Seek
Write to a file.	FilePut, FilePutObject, Print, Write, WriteLine

See also

- Keywords
- Visual Basic Runtime Library Members

Math Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Derive trigonometric functions.	Atan, Cos, Sin, Tan
General calculations.	Exp, Log, Sqrt
Generate random numbers.	Randomize, Rnd
Get absolute value.	Abs
Get the sign of an expression.	Sign
Perform numeric conversions.	Fix, Int

See also

- Derived Math Functions
- Keywords
- Visual Basic Runtime Library Members

Derived Math Functions (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following table shows non-intrinsic math functions that can be derived from the intrinsic math functions of the System.Math object. You can access the intrinsic math functions by adding `Imports System.Math` to your file or project.

FUNCTION	DERIVED EQUIVALENTS
Secant (Sec(x))	1 / Cos(x)
Cosecant (Csc(x))	1 / Sin(x)
Cotangent (Ctan(x))	1 / Tan(x)
Inverse sine (Asin(x))	Atan(x / Sqrt(-x * x + 1))
Inverse cosine (Acos(x))	Atan(-x / Sqrt(-x * x + 1)) + 2 * Atan(1)
Inverse secant (Asec(x))	2 * Atan(1) – Atan(Sign(x) / Sqrt(x * x – 1))
Inverse cosecant (Acsc(x))	Atan(Sign(x) / Sqrt(x * x – 1))
Inverse cotangent (Acot(x))	2 * Atan(1) - Atan(x)
Hyperbolic sine (Sinh(x))	(Exp(x) – Exp(-x)) / 2
Hyperbolic cosine (Cosh(x))	(Exp(x) + Exp(-x)) / 2
Hyperbolic tangent (Tanh(x))	(Exp(x) – Exp(-x)) / (Exp(x) + Exp(-x))
Hyperbolic secant (Sech(x))	2 / (Exp(x) + Exp(-x))
Hyperbolic cosecant (Csch(x))	2 / (Exp(x) – Exp(-x))
Hyperbolic cotangent (Coth(x))	(Exp(x) + Exp(-x)) / (Exp(x) – Exp(-x))
Inverse hyperbolic sine (Asinh(x))	Log(x + Sqrt(x * x + 1))
Inverse hyperbolic cosine (Acosh(x))	Log(x + Sqrt(x * x – 1))
Inverse hyperbolic tangent (Atanh(x))	Log((1 + x) / (1 – x)) / 2
Inverse hyperbolic secant (AsecH(x))	Log((Sqrt(-x * x + 1) + 1) / x)
Inverse hyperbolic cosecant (Acsch(x))	Log((Sign(x) * Sqrt(x * x + 1) + 1) / x)
Inverse hyperbolic cotangent (Acoth(x))	Log((x + 1) / (x – 1)) / 2

See also

- Math Functions

My Reference (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The `My` feature makes programming faster and easier by giving you intuitive access to commonly used methods, properties, and events. This table lists the objects contained in `My`, and the actions that can be performed with each.

ACTION	OBJECT
Accessing application information and services.	The `My.Application` object consists of the following classes: ApplicationBase provides members that are available in all projects. WindowsFormsApplicationBase provides members available in Windows Forms applications. ConsoleApplicationBase provides members available in console applications.
Accessing the host computer and its resources, services, and data.	`My.Computer` (Computer)
Accessing the forms in the current project.	My.Forms Object
Accessing the application log.	`My.Application.Log` (Log)
Accessing the current web request.	My.Request Object
Accessing resource elements.	My.Resources Object
Accessing the current web response.	My.Response Object
Accessing user and application level settings.	My.Settings Object
Accessing the current user's security context.	`My.User` (User)
Accessing XML Web services referenced by the current project.	My.WebServices Object

See also

- Overview of the Visual Basic Application Model
- Development with My

Operators Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Arithmetic	^, –, *, /, \, Mod, +, =
Assignment	=, ^=, *=, /=, \=, +=, -=, &=
Comparison	=, <>, <, >, <=, >=, Like, Is
Concatenation	&, +
Logical/bitwise operations	Not, And, Or, Xor, AndAlso, OrElse
Miscellaneous operations	AddressOf, Await, GetType

See also

- Keywords
- Visual Basic Runtime Library Members

Registry Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Studio language keywords and run-time library members are organized by purpose and use.

Using the `My` feature provides you with greater productivity and performance in registry operations than these elements. For more information, see RegistryProxy.

ACTION	LANGUAGE ELEMENT
Delete program settings.	DeleteSetting
Read program settings.	GetSetting, GetAllSettings
Save program settings.	SaveSetting

See also

- Keywords
- Visual Basic Runtime Library Members
- Reading from and Writing to the Registry

String Manipulation Summary (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic language keywords and run-time library members are organized by purpose and use.

ACTION	LANGUAGE ELEMENT
Compare two strings.	StrComp
Convert strings.	StrConv
Reverse a string.	InStrRev, StrReverse
Convert to lowercase or uppercase.	Format, LCase, UCase
Create a string of repeating characters.	Space, StrDup
Find the length of a string.	Len
Format a string.	Format, FormatCurrency, FormatDateTime, FormatNumber, FormatPercent
Manipulate strings.	InStr, Left, LTrim, Mid, Right, RTrim, Trim
Set string comparison rules.	Option Compare
Work with ASCII and ANSI values.	Asc, AscW, Chr, ChrW
Replace a specified substring.	Replace
Return a filter-based string array.	Filter
Return a specified number of substrings.	Split, Join

See also

- Keywords
- Visual Basic Runtime Library Members

Attributes (Visual Basic)

8/8/2019 • 2 minutes to read • Edit Online

Visual Basic provides several attributes that allow objects to interoperate with unmanaged code and one attribute that enables module members to be accessed without the module name. The following table lists the attributes used by Visual Basic.

ComClassAttribute	Instructs the compiler to add metadata that allows a class to be exposed as a COM object.
HideModuleNameAttribute	Allows the module members to be accessed using only the qualification needed for the module.
VBFixedArrayAttribute	Indicates that an array in a structure or non-local variable should be treated as a fixed-length array.
VBFixedStringAttribute	Indicates that a string should be treated as if it were fixed length.

See also

- Attributes overview

Constants and Enumerations (Visual Basic)

5/9/2019 • 3 minutes to read • Edit Online

Visual Basic supplies a number of predefined constants and enumerations for developers. Constants store values that remain constant throughout the execution of an application. Enumerations provide a convenient way to work with sets of related constants, and to associate constant values with names.

Constants

Conditional Compilation Constants

The following table lists the predefined constants available for conditional compilation.

CONSTANT	DESCRIPTION
`CONFIG`	A string that corresponds to the current setting of the **Active Solution Configuration** box in the **Configuration Manager**.
`DEBUG`	A `Boolean` value that can be set in the **Project Properties** dialog box. By default, the Debug configuration for a project defines `DEBUG`. When `DEBUG` is defined, Debug class methods generate output to the **Output** window. When it is not defined, Debug class methods are not compiled and no Debug output is generated.
`TARGET`	A string representing the output type for the project or the setting of the command-line **/target** option. The possible values of `TARGET` are: - "winexe" for a Windows application. - "exe" for a console application. - "library" for a class library. - "module" for a module. - The **/target** option may be set in the Visual Studio integrated development environment. For more information, see /target (Visual Basic).
`TRACE`	A `Boolean` value that can be set in the **Project Properties** dialog box. By default, all configurations for a project define `TRACE`. When `TRACE` is defined, Trace class methods generate output to the **Output** window. When it is not defined, Trace class methods are not compiled and no `Trace` output is generated.
`VBC_VER`	A number representing the Visual Basic version, in *major.minor* format.

Print and Display Constants

When you call print and display functions, you can use the following constants in your code in place of the actual values.

CONSTANT	DESCRIPTION
`vbCrLf`	Carriage return/linefeed character combination.
`vbCr`	Carriage return character.
`vbLf`	Linefeed character.
`vbNewLine`	Newline character.
`vbNullChar`	Null character.
`vbNullString`	Not the same as a zero-length string (""); used for calling external procedures.
`vbObjectError`	Error number. User-defined error numbers should be greater than this value. For example: `Err.Raise(Number) = vbObjectError + 1000`
`vbTab`	Tab character.
`vbBack`	Backspace character.
`vbFormFeed`	Not used in Microsoft Windows.
`vbVerticalTab`	Not useful in Microsoft Windows.

Enumerations

The following table lists and describes the enumerations provided by Visual Basic.

ENUMERATION	DESCRIPTION
AppWinStyle	Indicates the window style to use for the invoked program when calling the Shell function.
AudioPlayMode	Indicates how to play sounds when calling audio methods.
BuiltInRole	Indicates the type of role to check when calling the IsInRole method.
CallType	Indicates the type of procedure being invoked when calling the CallByName function.
CompareMethod	Indicates how to compare strings when calling comparison functions.
DateFormat	Indicates how to display dates when calling the FormatDateTime function.
DateInterval	Indicates how to determine and format date intervals when calling date-related functions.

ENUMERATION	DESCRIPTION
DeleteDirectoryOption	Specifies what should be done when a directory that is to be deleted contains files or directories.
DueDate	Indicates when payments are due when calling financial methods.
FieldType	Indicates whether text fields are delimited or fixed-width.
FileAttribute	Indicates the file attributes to use when calling file-access functions.
FirstDayOfWeek	Indicates the first day of the week to use when calling date-related functions.
FirstWeekOfYear	Indicates the first week of the year to use when calling date-related functions.
MsgBoxResult	Indicates which button was pressed on a message box, returned by the MsgBox function.
MsgBoxStyle	Indicates which buttons to display when calling the MsgBox function.
OpenAccess	Indicates how to open a file when calling file-access functions.
OpenMode	Indicates how to open a file when calling file-access functions.
OpenShare	Indicates how to open a file when calling file-access functions.
RecycleOption	Specifies whether a file should be deleted permanently or placed in the Recycle Bin.
SearchOption	Specifies whether to search all or only top-level directories.
TriState	Indicates a `Boolean` value or whether the default should be used when calling number-formatting functions.
UICancelOption	Specifies what should be done if the user clicks **Cancel** during an operation.
UIOption	Specifies whether or not to show a progress dialog when copying, deleting, or moving files or directories.
VariantType	Indicates the type of a variant object, returned by the VarType function.
VbStrConv	Indicates which type of conversion to perform when calling the StrConv function.

See also

- Visual Basic Language Reference

- Visual Basic
- Constants Overview
- Enumerations Overview

Data Type Summary (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

The following table shows the Visual Basic data types, their supporting common language runtime types, their nominal storage allocation, and their value ranges.

VISUAL BASIC TYPE	COMMON LANGUAGE RUNTIME TYPE STRUCTURE	NOMINAL STORAGE ALLOCATION	VALUE RANGE
Boolean	Boolean	Depends on implementing platform	`True` or `False`
Byte	Byte	1 byte	0 through 255 (unsigned)
Char (single character)	Char	2 bytes	0 through 65535 (unsigned)
Date	DateTime	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	Decimal	16 bytes	0 through +/- 79,228,162,514,264,337,593,543,950,335 (+/- 7.9...E+28) † with no decimal point; 0 through +/- 7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/- 0.0000000000000000000000000001 (+/-1E-28) †
Double (double-precision floating-point)	Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 † for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 † for positive values
Integer	Int32	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)

VISUAL BASIC TYPE	COMMON LANGUAGE RUNTIME TYPE STRUCTURE	NOMINAL STORAGE ALLOCATION	VALUE RANGE
Long (long integer)	Int64	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18 †) (signed)
Object	Object (class)	4 bytes on 32-bit platform 8 bytes on 64-bit platform	Any type can be stored in a variable of type `Object`
SByte	SByte	1 byte	-128 through 127 (signed)
Short (short integer)	Int16	2 bytes	-32,768 through 32,767 (signed)
Single (single-precision floating-point)	Single	4 bytes	-3.4028235E+38 through -1.401298E-45 † for negative values; 1.401298E-45 through 3.4028235E+38 † for positive values
String (variable-length)	String (class)	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	UInt32	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	UInt64	8 bytes	0 through 18,446,744,073,709,551,615 (1.8...E+19 †) (unsigned)
User-Defined (structure)	(inherits from ValueType)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	UInt16	2 bytes	0 through 65,535 (unsigned)

† In *scientific notation*, "E" refers to a power of 10. So 3.56E+2 signifies 3.56 x 10^2 or 356, and 3.56E-2 signifies 3.56 / 10^2 or 0.0356.

> **NOTE**
>
> For strings containing text, use the StrConv function to convert from one text format to another.

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements by using a type character. See Type Characters.

Memory Consumption

When you declare an elementary data type, it is not safe to assume that its memory consumption is the same as its nominal storage allocation. This is due to the following considerations:

- **Storage Assignment.** The common language runtime can assign storage based on the current characteristics of the platform on which your application is executing. If memory is nearly full, it might pack your declared elements as closely together as possible. In other cases it might align their memory addresses to natural hardware boundaries to optimize performance.
- **Platform Width.** Storage assignment on a 64-bit platform is different from assignment on a 32-bit platform.

Composite Data Types

The same considerations apply to each member of a composite data type, such as a structure or an array. You cannot rely on simply adding together the nominal storage allocations of the type's members. Furthermore, there are other considerations, such as the following:

- **Overhead.** Some composite types have additional memory requirements. For example, an array uses extra memory for the array itself and also for each dimension. On a 32-bit platform, this overhead is currently 12 bytes plus 8 bytes for each dimension. On a 64-bit platform this requirement is doubled.
- **Storage Layout.** You cannot safely assume that the order of storage in memory is the same as your order of declaration. You cannot even make assumptions about byte alignment, such as a 2-byte or 4-byte boundary. If you are defining a class or structure and you need to control the storage layout of its members, you can apply the StructLayoutAttribute attribute to the class or structure.

Object Overhead

An `Object` referring to any elementary or composite data type uses 4 bytes in addition to the data contained in the data type.

See also

- StrConv
- StructLayoutAttribute
- Type Conversion Functions
- Conversion Summary
- Type Characters
- Efficient Use of Data Types

Boolean Data Type (Visual Basic)

6/20/2019 • 2 minutes to read • Edit Online

Holds values that can be only `True` or `False`. The keywords `True` and `False` correspond to the two states of `Boolean` variables.

Remarks

Use the Boolean Data Type (Visual Basic) to contain two-state values such as true/false, yes/no, or on/off.

The default value of `Boolean` is `False`.

`Boolean` values are not stored as numbers, and the stored values are not intended to be equivalent to numbers. You should never write code that relies on equivalent numeric values for `True` and `False`. Whenever possible, you should restrict usage of `Boolean` variables to the logical values for which they are designed.

Type Conversions

When Visual Basic converts numeric data type values to `Boolean`, 0 becomes `False` and all other values become `True`. When Visual Basic converts `Boolean` values to numeric types, `False` becomes 0 and `True` becomes -1.

When you convert between `Boolean` values and numeric data types, keep in mind that the .NET Framework conversion methods do not always produce the same results as the Visual Basic conversion keywords. This is because the Visual Basic conversion retains behavior compatible with previous versions. For more information, see "Boolean Type Does Not Convert to Numeric Type Accurately" in Troubleshooting Data Types.

Programming Tips

- **Negative Numbers.** `Boolean` is not a numeric type and cannot represent a negative value. In any case, you should not use `Boolean` to hold numeric values.
- **Type Characters.** `Boolean` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Boolean structure.

Example

In the following example, `runningVB` is a `Boolean` variable, which stores a simple yes/no setting.

```
Dim runningVB As Boolean
' Check to see if program is running on Visual Basic engine.
If scriptEngine = "VB" Then
    runningVB = True
End If
```

See also

- System.Boolean
- Data Types

- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types
- Troubleshooting Data Types
- CType Function

Byte data type (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Holds unsigned 8-bit (1-byte) integers that range in value from 0 through 255.

Remarks

Use the `Byte` data type to contain binary data.

The default value of `Byte` is 0.

Literal assignments

You can declare and initialize a `Byte` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integral literal is outside the range of a `Byte` (that is, if it is less than Byte.MinValue or greater than Byte.MaxValue), a compilation error occurs.

In the following example, integers equal to 201 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from Integer to `byte` values.

```
Dim byteValue1 As Byte = 201
Console.WriteLine(byteValue1)

Dim byteValue2 As Byte = &H00C9
Console.WriteLine(byteValue2)

Dim byteValue3 As Byte = &B1100_1001
Console.WriteLine(byteValue3)
' The example displays the following output:
'          201
'          201
'          201
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim byteValue3 As Byte = &B1100_1001
Console.WriteLine(byteValue3)
' The example displays the following output:
'          201
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Byte = &H_6A
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Programming tips

- **Negative Numbers.** Because `Byte` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `Byte`, Visual Basic converts the expression to `Short` first.
- **Format Conversions.** When Visual Basic reads or writes files, or when it calls DLLs, methods, and properties, it can automatically convert between data formats. Binary data stored in `Byte` variables and arrays is preserved during such format conversions. You should not use a `String` variable for binary data, because its contents can be corrupted during conversion between ANSI and Unicode formats.
- **Widening.** The `Byte` data type widens to `Short`, `UShort`, `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, or `Double`. This means you can convert `Byte` to any of these types without encountering a System.OverflowException error.
- **Type Characters.** `Byte` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Byte structure.

Example

In the following example, `b` is a `Byte` variable. The statements demonstrate the range of the variable and the application of bit-shift operators to it.

```
' The valid range of a Byte variable is 0 through 255.
Dim b As Byte
b = 30
' The following statement causes an error because the value is too large.
'b = 256
' The following statement causes an error because the value is negative.
'b = -5
' The following statement sets b to 6.
b = CByte(5.7)

' The following statements apply bit-shift operators to b.
' The initial value of b is 6.
Console.WriteLine(b)
' Bit shift to the right divides the number in half. In this
' example, binary 110 becomes 11.
b >>= 1
' The following statement displays 3.
Console.WriteLine(b)
' Now shift back to the original position, and then one more bit
' to the left. Each shift to the left doubles the value. In this
' example, binary 11 becomes 1100.
b <<= 2
' The following statement displays 12.
Console.WriteLine(b)
```

See also

- System.Byte
- Data Types
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types

Char Data Type (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

Holds unsigned 16-bit (2-byte) code points ranging in value from 0 through 65535. Each *code point*, or character code, represents a single Unicode character.

Remarks

Use the `Char` data type when you need to hold only a single character and do not need the overhead of `String`. In some cases you can use `Char()`, an array of `Char` elements, to hold multiple characters.

The default value of `Char` is the character with a code point of 0.

Unicode Characters

The first 128 code points (0–127) of Unicode correspond to the letters and symbols on a standard U.S. keyboard. These first 128 code points are the same as those the ASCII character set defines. The second 128 code points (128–255) represent special characters, such as Latin-based alphabet letters, accents, currency symbols, and fractions. Unicode uses the remaining code points (256-65535) for a wide variety of symbols, including worldwide textual characters, diacritics, and mathematical and technical symbols.

You can use methods like IsDigit and IsPunctuation on a `Char` variable to determine its Unicode classification.

Type Conversions

Visual Basic does not convert directly between `Char` and the numeric types. You can use the Asc or AscW function to convert a `Char` value to an `Integer` that represents its code point. You can use the Chr or ChrW function to convert an `Integer` value to a `Char` that has that code point.

If the type checking switch (the Option Strict Statement) is on, you must append the literal type character to a single-character string literal to identify it as the `Char` data type. The following example illustrates this. The first assignment to the `charVar` variable generates compiler error BC30512 because `Option Strict` is on. The second compiles successfully because the `c` literal type character identifies the literal as a `Char` value.

```
Option Strict On

Module CharType
    Public Sub Main()
        Dim charVar As Char

        ' This statement generates compiler error BC30512 because Option Strict is On.
        charVar = "Z"

        ' The following statement succeeds because it specifies a Char literal.
        charVar = "Z"c
    End Sub
End Module
```

Programming Tips

- **Negative Numbers.** `Char` is an unsigned type and cannot represent a negative value. In any case, you should not use `Char` to hold numeric values.

- **Interop Considerations.** If you interface with components not written for the .NET Framework, for example Automation or COM objects, remember that character types have a different data width (8 bits) in other environments. If you pass an 8-bit argument to such a component, declare it as `Byte` instead of `Char` in your new Visual Basic code.
- **Widening.** The `Char` data type widens to `String`. This means you can convert `Char` to `String` and will not encounter a System.OverflowException.
- **Type Characters.** Appending the literal type character `C` to a single-character string literal forces it to the `Char` data type. `Char` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Char structure.

See also

- System.Char
- Asc
- AscW
- Chr
- ChrW
- Data Types
- String Data Type
- Type Conversion Functions
- Conversion Summary
- How to: Call a Windows Function that Takes Unsigned Types
- Efficient Use of Data Types

Date Data Type (Visual Basic)

7/30/2019 • 3 minutes to read • Edit Online

Holds IEEE 64-bit (8-byte) values that represent dates ranging from January 1 of the year 0001 through December 31 of the year 9999, and times from 12:00:00 AM (midnight) through 11:59:59.9999999 PM. Each increment represents 100 nanoseconds of elapsed time since the beginning of January 1 of the year 1 in the Gregorian calendar. The maximum value represents 100 nanoseconds before the beginning of January 1 of the year 10000.

Remarks

Use the `Date` data type to contain date values, time values, or date and time values.

The default value of `Date` is 0:00:00 (midnight) on January 1, 0001.

You can get the current date and time from the DateAndTime class.

Format Requirements

You must enclose a `Date` literal within number signs (`# #`). You must specify the date value in the format M/d/yyyy, for example `#5/31/1993#`, or yyyy-MM-dd, for example `#1993-5-31#`. You can use slashes when specifying the year first. This requirement is independent of your locale and your computer's date and time format settings.

The reason for this restriction is that the meaning of your code should never change depending on the locale in which your application is running. Suppose you hard-code a `Date` literal of `#3/4/1998#` and intend it to mean March 4, 1998. In a locale that uses mm/dd/yyyy, 3/4/1998 compiles as you intend. But suppose you deploy your application in many countries/regions. In a locale that uses dd/mm/yyyy, your hard-coded literal would compile to April 3, 1998. In a locale that uses yyyy/mm/dd, the literal would be invalid (April 1998, 0003) and cause a compiler error.

Workarounds

To convert a `Date` literal to the format of your locale, or to a custom format, supply the literal to the Format function, specifying either a predefined or user-defined date format. The following example demonstrates this.

```
MsgBox("The formatted date is " & Format(#5/31/1993#, "dddd, d MMM yyyy"))
```

Alternatively, you can use one of the overloaded constructors of the DateTime structure to assemble a date and time value. The following example creates a value to represent May 31, 1993 at 12:14 in the afternoon.

```
Dim dateInMay As New System.DateTime(1993, 5, 31, 12, 14, 0)
```

Hour Format

You can specify the time value in either 12-hour or 24-hour format, for example `#1:15:30 PM#` or `#13:15:30#`. However, if you do not specify either the minutes or the seconds, you must specify AM or PM.

Date and Time Defaults

If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to January 1, 0001. If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to the start of the day, that is, midnight (0:00:00).

Type Conversions

If you convert a `Date` value to the `String` type, Visual Basic renders the date according to the short date format specified by the run-time locale, and it renders the time according to the time format (either 12-hour or 24-hour) specified by the run-time locale.

Programming Tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that date/time types in other environments are not compatible with the Visual Basic `Date` type. If you are passing a date/time argument to such a component, declare it as `Double` instead of `Date` in your new Visual Basic code, and use the conversion methods DateTime.FromOADate and DateTime.ToOADate.
- **Type Characters.** `Date` has no literal type character or identifier type character. However, the compiler treats literals enclosed within number signs (`# #`) as `Date`.
- **Framework Type.** The corresponding type in the .NET Framework is the System.DateTime structure.

Example

A variable or constant of the `Date` data type holds both the date and the time. The following example illustrates this.

```
Dim someDateAndTime As Date = #8/13/2002 12:14 PM#
```

See also

- System.DateTime
- Data Types
- Standard Date and Time Format Strings
- Custom Date and Time Format Strings
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types

Decimal Data Type (Visual Basic)

7/26/2019 • 2 minutes to read • Edit Online

Holds signed 128-bit (16-byte) values representing 96-bit (12-byte) integer numbers scaled by a variable power of 10. The scaling factor specifies the number of digits to the right of the decimal point; it ranges from 0 through 28. With a scale of 0 (no decimal places), the largest possible value is +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9228162514264337593543950335E+28). With 28 decimal places, the largest value is +/-7.9228162514264337593543950335, and the smallest nonzero value is +/-0.0000000000000000000000000001 (+/-1E-28).

Remarks

The `Decimal` data type provides the greatest number of significant digits for a number. It supports up to 29 significant digits and can represent values in excess of 7.9228 x 10^28. It is particularly suitable for calculations, such as financial, that require a large number of digits but cannot tolerate rounding errors.

The default value of `Decimal` is 0.

Programming Tips

- **Precision.** `Decimal` is not a floating-point data type. The `Decimal` structure holds a binary integer value, together with a sign bit and an integer scaling factor that specifies what portion of the value is a decimal fraction. Because of this, `Decimal` numbers have a more precise representation in memory than floating-point types (`Single` and `Double`).
- **Performance.** The `Decimal` data type is the slowest of all the numeric types. You should weigh the importance of precision against performance before choosing a data type.
- **Widening.** The `Decimal` data type widens to `Single` or `Double` . This means you can convert `Decimal` to either of these types without encountering a System.OverflowException error.
- **Trailing Zeros.** Visual Basic does not store trailing zeros in a `Decimal` literal. However, a `Decimal` variable preserves any trailing zeros acquired computationally. The following example illustrates this.

  ```
  Dim d1, d2, d3, d4 As Decimal
  d1 = 2.375D
  d2 = 1.625D
  d3 = d1 + d2
  d4 = 4.000D
  MsgBox("d1 = " & CStr(d1) & ", d2 = " & CStr(d2) &
        ", d3 = " & CStr(d3) & ", d4 = " & CStr(d4))
  ```

  The output of `MsgBox` in the preceding example is as follows:

  ```
  d1 = 2.375, d2 = 1.625, d3 = 4.000, d4 = 4
  ```

- **Type Characters.** Appending the literal type character `D` to a literal forces it to the `Decimal` data type. Appending the identifier type character `@` to any identifier forces it to `Decimal` .
- **Framework Type.** The corresponding type in the .NET Framework is the System.Decimal structure.

Range

You might need to use the `D` type character to assign a large value to a `Decimal` variable or constant. This requirement is because the compiler interprets a literal as `Long` unless a literal type character follows the literal, as the following example shows.

```
Dim bigDec1 As Decimal = 9223372036854775807   ' No overflow.
Dim bigDec2 As Decimal = 9223372036854775808   ' Overflow.
Dim bigDec3 As Decimal = 92233720368547758080  ' No overflow.
```

The declaration for `bigDec1` doesn't produce an overflow because the value that's assigned to it falls within the range for `Long`. The `Long` value can be assigned to the `Decimal` variable.

The declaration for `bigDec2` generates an overflow error because the value that's assigned to it is too large for `Long`. Because the numeric literal can't first be interpreted as a `Long`, it can't be assigned to the `Decimal` variable.

For `bigDec3`, the literal type character `D` solves the problem by forcing the compiler to interpret the literal as a `Decimal` instead of as a `Long`.

See also

- System.Decimal
- Decimal.Decimal
- Math.Round
- Data Types
- Single Data Type
- Double Data Type
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types

Double Data Type (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

Holds signed IEEE 64-bit (8-byte) double-precision floating-point numbers that range in value from -1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values and from 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values. Double-precision numbers store an approximation of a real number.

Remarks

The `Double` data type provides the largest and smallest possible magnitudes for a number.

The default value of `Double` is 0.

Programming Tips

- **Precision.** When you work with floating-point numbers, remember that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the `Mod` operator. For more information, see Troubleshooting Data Types.
- **Trailing Zeros.** The floating-point data types do not have any internal representation of trailing zero characters. For example, they do not distinguish between 4.2000 and 4.2. Consequently, trailing zero characters do not appear when you display or print floating-point values.
- **Type Characters.** Appending the literal type character `R` to a literal forces it to the `Double` data type. For example, if an integer value is followed by `R`, the value is changed to a `Double`.

  ```
  ' Visual Basic expands the 4 in the statement Dim dub As Double = 4R to 4.0:
  Dim dub As Double = 4.0R
  ```

  Appending the identifier type character `#` to any identifier forces it to `Double`. In the following example, the variable `num` is typed as a `Double`:

  ```
  Dim num# = 3
  ```

- **Framework Type.** The corresponding type in the .NET Framework is the System.Double structure.

See also

- System.Double
- Data Types
- Decimal Data Type
- Single Data Type
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types
- Troubleshooting Data Types
- Type Characters

Integer data type (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Holds signed 32-bit (4-byte) integers that range in value from -2,147,483,648 through 2,147,483,647.

Remarks

The `Integer` data type provides optimal performance on a 32-bit processor. The other integral types are slower to load and store from and to memory.

The default value of `Integer` is 0.

Literal assignments

You can declare and initialize an `Integer` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Integer` (that is, if it is less than Int32.MinValue or greater than Int32.MaxValue, a compilation error occurs.

In the following example, integers equal to 90,946 that are represented as decimal, hexadecimal, and binary literals are assigned to `Integer` values.

```
Dim intValue1 As Integer = 90946
Console.WriteLine(intValue1)
Dim intValue2 As Integer = &H16342
Console.WriteLine(intValue2)

Dim intValue3 As Integer = &B0001_0110_0011_0100_0010
Console.WriteLine(intValue3)
' The example displays the following output:
'          90946
'          90946
'          90946
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim intValue1 As Integer = 90_946
Console.WriteLine(intValue1)

Dim intValue2 As Integer = &H0001_6342
Console.WriteLine(intValue2)

Dim intValue3 As Integer = &B0001_0110_0011_0100_0010
Console.WriteLine(intValue3)
' The example displays the following output:
'          90946
'          90946
'          90946
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Integer = &H_C305_F860
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `I` type character to denote the `Integer` data type, as the following example shows.

```
Dim number = &H_035826I
```

Programming tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, such as Automation or COM objects, remember that `Integer` has a different data width (16 bits) in other environments. If you are passing a 16-bit argument to such a component, declare it as `Short` instead of `Integer` in your new Visual Basic code.
- **Widening.** The `Integer` data type widens to `Long`, `Decimal`, `Single`, or `Double`. This means you can convert `Integer` to any one of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type character `I` to a literal forces it to the `Integer` data type. Appending the identifier type character `%` to any identifier forces it to `Integer`.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Int32 structure.

Range

If you try to set a variable of an integral type to a number outside the range for that type, an error occurs. If you try to set it to a fraction, the number is rounded up or down to the nearest integer value. If the number is equally close to two integer values, the value is rounded to the nearest even integer. This behavior minimizes rounding errors that result from consistently rounding a midpoint value in a single direction. The following code shows examples of rounding.

```
' The valid range of an Integer variable is -2147483648 through +2147483647.
Dim k As Integer
' The following statement causes an error because the value is too large.
k = 2147483648
' The following statement sets k to 6.
k = 5.9
' The following statement sets k to 4
k = 4.5
' The following statement sets k to 6
' Note, Visual Basic uses banker's rounding (toward nearest even number)
k = 5.5
```

See also

- System.Int32
- Data Types
- Long Data Type
- Short Data Type
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types

Long data type (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Holds signed 64-bit (8-byte) integers ranging in value from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18).

Remarks

Use the `Long` data type to contain integer numbers that are too large to fit in the `Integer` data type.

The default value of `Long` is 0.

Literal assignments

You can declare and initialize a `Long` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Long` (that is, if it is less than Int64.MinValue or greater than Int64.MaxValue, a compilation error occurs.

In the following example, integers equal to 4,294,967,296 that are represented as decimal, hexadecimal, and binary literals are assigned to `Long` values.

```
Dim longValue1 As Long = 4294967296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H100000000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'          4294967296
'          4294967296
'          4294967296
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim longValue1 As Long = 4_294_967_296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H1_0000_0000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'          4294967296
'          4294967296
'          4294967296
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Long = &H_0FAC_0326_1489_D68C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `L` type character to denote the `Long` data type, as the following example shows.

```
Dim number = &H_0FAC_0326_1489_D68CL
```

Programming tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, remember that `Long` has a different data width (32 bits) in other environments. If you are passing a 32-bit argument to such a component, declare it as `Integer` instead of `Long` in your new Visual Basic code.
- **Widening.** The `Long` data type widens to `Decimal`, `Single`, or `Double`. This means you can convert `Long` to any one of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type character `L` to a literal forces it to the `Long` data type. Appending the identifier type character `&` to any identifier forces it to `Long`.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Int64 structure.

See also

- Int64
- Data Types
- Integer Data Type
- Short Data Type
- Type Conversion Functions

- Conversion Summary
- Efficient Use of Data Types

Object Data Type

7/26/2019 • 2 minutes to read • Edit Online

Holds addresses that refer to objects. You can assign any reference type (string, array, class, or interface) to an `Object` variable. An `Object` variable can also refer to data of any value type (numeric, `Boolean`, `Char`, `Date`, structure, or enumeration).

Remarks

The `Object` data type can point to data of any data type, including any object instance your application recognizes. Use `Object` when you do not know at compile time what data type the variable might point to.

The default value of `Object` is `Nothing` (a null reference).

Data Types

You can assign a variable, constant, or expression of any data type to an `Object` variable. To determine the data type an `Object` variable currently refers to, you can use the GetTypeCode method of the System.Type class. The following example illustrates this.

```
Dim myObject As Object
' Suppose myObject has now had something assigned to it.
Dim datTyp As Integer
datTyp = Type.GetTypeCode(myObject.GetType())
```

The `Object` data type is a reference type. However, Visual Basic treats an `Object` variable as a value type when it refers to data of a value type.

Storage

Whatever data type it refers to, an `Object` variable does not contain the data value itself, but rather a pointer to the value. It always uses four bytes in computer memory, but this does not include the storage for the data representing the value of the variable. Because of the code that uses the pointer to locate the data, `Object` variables holding value types are slightly slower to access than explicitly typed variables.

Programming Tips

- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that pointer types in other environments are not compatible with the Visual Basic `Object` type.
- **Performance.** A variable you declare with the `Object` type is flexible enough to contain a reference to any object. However, when you invoke a method or property on such a variable, you always incur *late binding* (at run time). To force *early binding* (at compile time) and better performance, declare the variable with a specific class name, or cast it to the specific data type.

  When you declare an object variable, try to use a specific class type, for example OperatingSystem, instead of the generalized `Object` type. You should also use the most specific class available, such as TextBox instead of Control, so that you can access its properties and methods. You can usually use the **Classes** list in the **Object Browser** to find available class names.

- **Widening.** All data types and all reference types widen to the `Object` data type. This means you can convert any type to `Object` without encountering a System.OverflowException error.

  However, if you convert between value types and `Object`, Visual Basic performs operations called *boxing* and *unboxing*, which make execution slower.

- **Type Characters.** `Object` has no literal type character or identifier type character.

- **Framework Type.** The corresponding type in the .NET Framework is the System.Object class.

Example

The following example illustrates an `Object` variable pointing to an object instance.

```
Dim objDb As Object
Dim myCollection As New Collection()
' Suppose myCollection has now been populated.
objDb = myCollection.Item(1)
```

See also

- Object
- Data Types
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types
- How to: Determine Whether Two Objects Are Related
- How to: Determine Whether Two Objects Are Identical

SByte data type (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Holds signed 8-bit (1-byte) integers that range in value from -128 through 127.

Remarks

Use the `SByte` data type to contain integer values that do not require the full data width of `Integer` or even the half data width of `Short`. In some cases, the common language runtime might be able to pack your `SByte` variables closely together and save memory consumption.

The default value of `SByte` is 0.

Literal assignments

You can declare and initialize an `SByte` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal.

In the following example, integers equal to -102 that are represented as decimal, hexadecimal, and binary literals are assigned to `SByte` values. This example requires that you compile with the `/removeintchecks` compiler switch.

```
Dim sbyteValue1 As SByte = -102
Console.WriteLine(sbyteValue1)

Dim sbyteValue4 As SByte = &H9A
Console.WriteLine(sbyteValue4)

Dim sbyteValue5 As SByte = &B1001_1010
Console.WriteLine(sbyteValue5)
' The example displays the following output:
'          -102
'          -102
'          -102
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim sbyteValue3 As SByte = &B1001_1010
Console.WriteLine(sbyteValue3)
' The example displays the following output:
'          -102
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As SByte = &H_F9
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

If the integer literal is outside the range of `SByte` (that is, if it is less than SByte.MinValue or greater than SByte.MaxValue, a compilation error occurs. When an integer literal has no suffix, an Integer is inferred. If the integer literal is outside the range of the `Integer` type, a Long is inferred. This means that, in the previous examples, the numeric literals `0x9A` and `0b10011010` are interpreted as 32-bit signed integers with a value of 156, which exceeds SByte.MaxValue. To successfully compile code like this that assigns a non-decimal integer to an `SByte`, you can do either of the following:

- Disable integer bounds checks by compiling with the `/removeintchecks` compiler switch.
- Use a type character to explicitly define the literal value that you want to assign to the `SByte`. The following example assigns a negative literal `Short` value to an `SByte`. Note that, for negative numbers, the high-order bit of the high-order word of the numeric literal must be set. In the case of our example, this is bit 15 of the literal `Short` value.

  ```
  Dim sByteValue1 As SByte = &HFF_9As
  Dim sByteValue2 As SByte = &B1111_1111_1001_1010s
  Console.WriteLine(sByteValue1)
  Console.WriteLine(sByteValue2)
  ```

Programming tips

- **CLS Compliance.** The `SByte` data type is not part of the Common Language Specification (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Widening.** The `SByte` data type widens to `Short`, `Integer`, `Long`, `Decimal`, `Single`, and `Double`. This means you can convert `SByte` to any of these types without encountering a System.OverflowException error.
- **Type Characters.** `SByte` has no literal type character or identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.SByte structure.

See also

- System.SByte
- Data Types
- Type Conversion Functions
- Conversion Summary
- Short Data Type
- Integer Data Type
- Long Data Type
- Efficient Use of Data Types

Short data type (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Holds signed 16-bit (2-byte) integers that range in value from -32,768 through 32,767.

Remarks

Use the `Short` data type to contain integer values that do not require the full data width of `Integer`. In some cases, the common language runtime can pack your `Short` variables closely together and save memory consumption.

The default value of `Short` is 0.

Literal assignments

You can declare and initialize a `Short` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `Short` (that is, if it is less than Int16.MinValue or greater than Int16.MaxValue, a compilation error occurs.

In the following example, integers equal to 1,034 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from Integer to `Short` values.

```
Dim shortValue1 As Short = 1034
Console.WriteLine(shortValue1)

Dim shortValue2 As Short = &H040A
Console.WriteLine(shortValue2)

Dim shortValue3 As Short = &B0100_00001010
Console.WriteLine(shortValue3)
' The example displays the following output:
'          1034
'          1034
'          1034
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim shortValue1 As Short = 1_034
Console.WriteLine(shortValue1)

Dim shortValue3 As Short = &B00000100_00001010
Console.WriteLine(shortValue3)
' The example displays the following output:
'          1034
'          1034
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As Short = &H_3264
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `S` type character to denote the `Short` data type, as the following example shows.

```
Dim number = &H_3264S
```

Programming tips

- **Widening.** The `Short` data type widens to `Integer` , `Long` , `Decimal` , `Single` , or `Double` . This means you can convert `Short` to any one of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type character `S` to a literal forces it to the `Short` data type. `Short` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Int16 structure.

See also

- System.Int16
- Data Types
- Type Conversion Functions
- Conversion Summary
- Integer Data Type
- Long Data Type
- Efficient Use of Data Types

Single Data Type (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Holds signed IEEE 32-bit (4-byte) single-precision floating-point numbers ranging in value from -3.4028235E+38 through -1.401298E-45 for negative values and from 1.401298E-45 through 3.4028235E+38 for positive values. Single-precision numbers store an approximation of a real number.

Remarks

Use the `Single` data type to contain floating-point values that do not require the full data width of `Double`. In some cases the common language runtime might be able to pack your `Single` variables closely together and save memory consumption.

The default value of `Single` is 0.

Programming Tips

- **Precision.** When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the `Mod` operator. For more information, see Troubleshooting Data Types.
- **Widening.** The `Single` data type widens to `Double`. This means you can convert `Single` to `Double` without encountering a System.OverflowException error.
- **Trailing Zeros.** The floating-point data types do not have any internal representation of trailing 0 characters. For example, they do not distinguish between 4.2000 and 4.2. Consequently, trailing 0 characters do not appear when you display or print floating-point values.
- **Type Characters.** Appending the literal type character `F` to a literal forces it to the `Single` data type. Appending the identifier type character `!` to any identifier forces it to `Single`.
- **Framework Type.** The corresponding type in the .NET Framework is the System.Single structure.

See also

- System.Single
- Data Types
- Decimal Data Type
- Double Data Type
- Type Conversion Functions
- Conversion Summary
- Efficient Use of Data Types
- Troubleshooting Data Types

String Data Type (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Holds sequences of unsigned 16-bit (2-byte) code points that range in value from 0 through 65535. Each *code point*, or character code, represents a single Unicode character. A string can contain from 0 to approximately two billion (2 ^ 31) Unicode characters.

Remarks

Use the `String` data type to hold multiple characters without the array management overhead of `Char()`, an array of `Char` elements.

The default value of `String` is `Nothing` (a null reference). Note that this is not the same as the empty string (value `""`).

Unicode Characters

The first 128 code points (0–127) of Unicode correspond to the letters and symbols on a standard U.S. keyboard. These first 128 code points are the same as those the ASCII character set defines. The second 128 code points (128–255) represent special characters, such as Latin-based alphabet letters, accents, currency symbols, and fractions. Unicode uses the remaining code points (256-65535) for a wide variety of symbols. This includes worldwide textual characters, diacritics, and mathematical and technical symbols.

You can use methods such as IsDigit and IsPunctuation on an individual character in a `String` variable to determine its Unicode classification.

Format Requirements

You must enclose a `String` literal within quotation marks (`" "`). If you must include a quotation mark as one of the characters in the string, you use two contiguous quotation marks (`""`). The following example illustrates this.

```
Dim j As String = "Joe said ""Hello"" to me."
Dim h As String = "Hello"
' The following messages all display the same thing:
' "Joe said "Hello" to me."
MsgBox(j)
MsgBox("Joe said " & """ & h & """ & " to me.")
MsgBox("Joe said """ & h & """ to me.")
```

Note that the contiguous quotation marks that represent a quotation mark in the string are independent of the quotation marks that begin and end the `String` literal.

String Manipulations

Once you assign a string to a `String` variable, that string is *immutable*, which means you cannot change its length or contents. When you alter a string in any way, Visual Basic creates a new string and abandons the previous one. The `String` variable then points to the new string.

You can manipulate the contents of a `String` variable by using a variety of string functions. The following example illustrates the Left function

```
Dim S As String = "Database"
' The following statement sets S to a new string containing "Data".
S = Microsoft.VisualBasic.Left(S, 4)
```

A string created by another component might be padded with leading or trailing spaces. If you receive such a string, you can use the Trim, LTrim, and RTrim functions to remove these spaces.

For more information about string manipulations, see Strings.

Programming Tips

- **Negative Numbers.** Remember that the characters held by `String` are unsigned and cannot represent negative values. In any case, you should not use `String` to hold numeric values.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, remember that string characters have a different data width (8 bits) in other environments. If you are passing a string argument of 8-bit characters to such a component, declare it as `Byte()`, an array of `Byte` elements, instead of `String` in your new Visual Basic code.
- **Type Characters.** Appending the identifier type character `$` to any identifier forces it to the `String` data type. `String` has no literal type character. However, the compiler treats literals enclosed in quotation marks (`" "`) as `String`.
- **Framework Type.** The corresponding type in the .NET Framework is the System.String class.

See also

- System.String
- Data Types
- Char Data Type
- Type Conversion Functions
- Conversion Summary
- How to: Call a Windows Function that Takes Unsigned Types
- Efficient Use of Data Types

UInteger data type

4/28/2019 • 2 minutes to read • Edit Online

Holds unsigned 32-bit (4-byte) integers ranging in value from 0 through 4,294,967,295.

Remarks

The `UInteger` data type provides the largest unsigned value in the most efficient data width.

The default value of `UInteger` is 0.

Literal assignments

You can declare and initialize a `UInteger` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `UInteger` (that is, if it is less than UInt32.MinValue or greater than UInt32.MaxValue, a compilation error occurs.

In the following example, integers equal to 3,000,000,000 that are represented as decimal, hexadecimal, and binary literals are assigned to `UInteger` values.

```
Dim uintValue1 As UInteger = 3000000000ui
Console.WriteLine(uintValue1)

Dim uintValue2 As UInteger = &HB2D05E00ui
Console.WriteLine(uintValue2)

Dim uintValue3 As UInteger = &B1011_0010_1101_0000_0101_1110_0000_0000ui
Console.WriteLine(uintValue3)
' The example displays the following output:
'          3000000000
'          3000000000
'          3000000000
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim uintValue1 As UInteger = 3_000_000_000ui
Console.WriteLine(uintValue1)

Dim uintValue2 As UInteger = &HB2D0_5E00ui
Console.WriteLine(uintValue2)

Dim uintValue3 As UInteger = &B1011_0010_1101_0000_0101_1110_0000_0000ui
Console.WriteLine(uintValue3)
' The example displays the following output:
'          3000000000
'          3000000000
'          3000000000
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As UInteger = &H_0F8C_0326
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `UI` or `ui` type character to denote the `UInteger` data type, as the following example shows.

```
Dim number = &H_0FAC_14D7ui
```

Programming tips

The `UInteger` and `Integer` data types provide optimal performance on a 32-bit processor, because the smaller integer types (`UShort` , `Short` , `Byte` , and `SByte`), even though they use fewer bits, take more time to load, store, and fetch.

- **Negative Numbers.** Because `UInteger` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `UInteger` , Visual Basic converts the expression to `Long` first.
- **CLS Compliance.** The `UInteger` data type is not part of the Common Language Specification (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that types such as `uint` can have a different data width (16 bits) in other environments. If you are passing a 16-bit argument to such a component, declare it as `UShort` instead of `UInteger` in your managed Visual Basic code.
- **Widening.** The `UInteger` data type widens to `Long` , `ULong` , `Decimal` , `Single` , and `Double` . This means you can convert `UInteger` to any of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type characters `UI` to a literal forces it to the `UInteger` data type. `UInteger` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.UInt32 structure.

See also

- UInt32
- Data Types
- Type Conversion Functions
- Conversion Summary
- How to: Call a Windows Function that Takes Unsigned Types
- Efficient Use of Data Types

ULong data type (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Holds unsigned 64-bit (8-byte) integers ranging in value from 0 through 18,446,744,073,709,551,615 (more than 1.84 times 10 ^ 19).

Remarks

Use the `ULong` data type to contain binary data too large for `UInteger`, or the largest possible unsigned integer values.

The default value of `ULong` is 0.

Literal assignments

You can declare and initialize a `ULong` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `ULong` (that is, if it is less than UInt64.MinValue or greater than UInt64.MaxValue, a compilation error occurs.

In the following example, integers equal to 7,934,076,125 that are represented as decimal, hexadecimal, and binary literals are assigned to `ULong` values.

```
Dim ulongValue1 As ULong = 7934076125
Console.WriteLine(ulongValue1)

Dim ulongValue2 As ULong = &H0001D8e864DD
Console.WriteLine(ulongValue2)

Dim ulongValue3 As ULong = &B0001_1101_1000_1110_1000_0110_0100_1101_1101
Console.WriteLine(ulongValue3)
' The example displays the following output:
'          7934076125
'          7934076125
'          7934076125
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim longValue1 As Long = 4_294_967_296
Console.WriteLine(longValue1)

Dim longValue2 As Long = &H1_0000_0000
Console.WriteLine(longValue2)

Dim longValue3 As Long = &B1_0000_0000_0000_0000_0000_0000_0000_0000
Console.WriteLine(longValue3)
' The example displays the following output:
'          4294967296
'          4294967296
'          4294967296
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As ULong = &H_F9AC_0326_1489_D68C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `UL` or `ul` type character to denote the `ULong` data type, as the following example shows.

```
Dim number = &H_00_00_0A_96_2F_AC_14_D7ul
```

Programming tips

- **Negative Numbers.** Because `ULong` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `ULong`, Visual Basic converts the expression to `Decimal` first.
- **CLS Compliance.** The `ULong` data type is not part of the Common Language Specification (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that types such as `ulong` can have a different data width (32 bits) in other environments. If you are passing a 32-bit argument to such a component, declare it as `UInteger` instead of `ULong` in your managed Visual Basic code.

  Furthermore, Automation does not support 64-bit integers on Windows 95, Windows 98, Windows ME, or Windows 2000. You cannot pass a Visual Basic `ULong` argument to an Automation component on these platforms.
- **Widening.** The `ULong` data type widens to `Decimal`, `Single`, and `Double`. This means you can convert `ULong` to any of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type characters `UL` to a literal forces it to the `ULong` data type. `ULong` has no identifier type character.

- **Framework Type.** The corresponding type in the .NET Framework is the System.UInt64 structure.

See also

- UInt64
- Data Types
- Type Conversion Functions
- Conversion Summary
- How to: Call a Windows Function that Takes Unsigned Types
- Efficient Use of Data Types

User-Defined Data Type

7/30/2019 • 2 minutes to read • Edit Online

Holds data in a format you define. The `Structure` statement defines the format.

Previous versions of Visual Basic support the user-defined type (UDT). The current version expands the UDT to a *structure*. A structure is a concatenation of one or more *members* of various data types. Visual Basic treats a structure as a single unit, although you can also access its members individually.

Remarks

Define and use a structure data type when you need to combine various data types into a single unit, or when none of the elementary data types serve your needs.

The default value of a structure data type consists of the combination of the default values of each of its members.

Declaration Format

A structure declaration starts with the Structure Statement and ends with the `End Structure` statement. The `Structure` statement supplies the name of the structure, which is also the identifier of the data type the structure is defining. Other parts of the code can use this identifier to declare variables, parameters, and function return values to be of this structure's data type.

The declarations between the `Structure` and `End Structure` statements define the members of the structure.

Member Access Levels

You must declare every member using a Dim Statement or a statement that specifies access level, such as Public, Friend, or Private. If you use a `Dim` statement, the access level defaults to public.

Programming Tips

- **Memory Consumption.** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration. If you need to control the storage layout of a structure, you can apply the StructLayoutAttribute attribute to the `Structure` statement.
- **Interop Considerations.** If you are interfacing with components not written for the .NET Framework, for example Automation or COM objects, keep in mind that user-defined types in other environments are not compatible with Visual Basic structure types.
- **Widening.** There is no automatic conversion to or from any structure data type. You can define conversion operators on your structure using the Operator Statement, and you can declare each conversion operator to be `Widening` or `Narrowing`.
- **Type Characters.** Structure data types have no literal type character or identifier type character.
- **Framework Type.** There is no corresponding type in the .NET Framework. All structures inherit from the .NET Framework class System.ValueType, but no individual structure corresponds to System.ValueType.

Example

The following paradigm shows the outline of the declaration of a structure.

```
[Public | Protected | Friend | Protected Friend | Private] Structure structname
    {Dim | Public | Friend | Private} member1 As datatype1
    ' ...
    {Dim | Public | Friend | Private} memberN As datatypeN
End Structure
```

See also

- ValueType
- StructLayoutAttribute
- Data Types
- Type Conversion Functions
- Conversion Summary
- Structure Statement
- Widening
- Narrowing
- Structures
- Efficient Use of Data Types

UShort data type (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Holds unsigned 16-bit (2-byte) integers ranging in value from 0 through 65,535.

Remarks

Use the `UShort` data type to contain binary data too large for `Byte`.

The default value of `UShort` is 0.

Literal assignments

You can declare and initialize a `UShort` variable by assigning it a decimal literal, a hexadecimal literal, an octal literal, or (starting with Visual Basic 2017) a binary literal. If the integer literal is outside the range of `UShort` (that is, if it is less than UInt16.MinValue or greater than UInt16.MaxValue, a compilation error occurs.

In the following example, integers equal to 65,034 that are represented as decimal, hexadecimal, and binary literals are assigned to `UShort` values.

```
Dim ushortValue1 As UShort = 65034
Console.WriteLine(ushortValue1)

Dim ushortValue2 As UShort = &HFE0A
Console.WriteLine(ushortValue2)

Dim ushortValue3 As UShort = &B1111_1110_0000_1010
Console.WriteLine(ushortValue3)
' The example displays the following output:
'          65034
'          65034
'          65034
```

NOTE

You use the prefix `&h` or `&H` to denote a hexadecimal literal, the prefix `&b` or `&B` to denote a binary literal, and the prefix `&o` or `&O` to denote an octal literal. Decimal literals have no prefix.

Starting with Visual Basic 2017, you can also use the underscore character, `_`, as a digit separator to enhance readability, as the following example shows.

```
Dim ushortValue1 As UShort = 65_034
Console.WriteLine(ushortValue1)

Dim ushortValue3 As UShort = &B11111110_00001010
Console.WriteLine(ushortValue3)
' The example displays the following output:
'          65034
'          65034
```

Starting with Visual Basic 15.5, you can also use the underscore character (`_`) as a leading separator between the prefix and the hexadecimal, binary, or octal digits. For example:

```
Dim number As UShort = &H_FF8C
```

To use the underscore character as a leading separator, you must add the following element to your Visual Basic project (*.vbproj) file:

```
<PropertyGroup>
  <LangVersion>15.5</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

Numeric literals can also include the `US` or `us` type character to denote the `UShort` data type, as the following example shows.

```
Dim number = &H_5826us
```

Programming tips

- **Negative Numbers.** Because `UShort` is an unsigned type, it cannot represent a negative number. If you use the unary minus (`-`) operator on an expression that evaluates to type `UShort`, Visual Basic converts the expression to `Integer` first.
- **CLS Compliance.** The `UShort` data type is not part of the Common Language Specification (CLS), so CLS-compliant code cannot consume a component that uses it.
- **Widening.** The `UShort` data type widens to `Integer`, `UInteger`, `Long`, `ULong`, `Decimal`, `Single`, and `Double`. This means you can convert `UShort` to any of these types without encountering a System.OverflowException error.
- **Type Characters.** Appending the literal type characters `US` to a literal forces it to the `UShort` data type. `UShort` has no identifier type character.
- **Framework Type.** The corresponding type in the .NET Framework is the System.UInt16 structure.

See also

- UInt16
- Data Types
- Type Conversion Functions
- Conversion Summary
- How to: Call a Windows Function that Takes Unsigned Types
- Efficient Use of Data Types

Directives (Visual Basic)

8/24/2018 • 2 minutes to read • Edit Online

The topics in this section document the Visual Basic source code compiler directives.

In This Section

#Const Directive -- Define a compiler constant

#ExternalSource Directive -- Indicate a mapping between source lines and text external to the source

#If...Then...#Else Directives -- Compile selected blocks of code

#Region Directive -- Collapse and hide sections of code in the Visual Studio editor

#Disable, #Enable -- Disable and enable specific warnings for regions of code.

```
#Disable Warning BC42356 ' suppress warning about no awaits in this method
    Async Function TestAsync() As Task
        Console.WriteLine("testing")
    End Function
#Enable Warning BC42356
```

You can disable and enable a comma-separated list of warning codes too.

Related Sections

Visual Basic Language Reference

Visual Basic

#Const Directive

4/2/2019 • 2 minutes to read • Edit Online

Defines conditional compiler constants for Visual Basic.

Syntax

```
#Const constname = expression
```

Parts

`constname`
Required. Name of the constant being defined.

`expression`
Required. Literal, other conditional compiler constant, or any combination that includes any or all arithmetic or logical operators except `Is`.

Remarks

Conditional compiler constants are always private to the file in which they appear. You cannot create public compiler constants using the `#Const` directive; you can create them only in the user interface or with the `/define` compiler option.

You can use only conditional compiler constants and literals in `expression`. Using a standard constant defined with `Const` causes an error. Conversely, you can use constants defined with the `#Const` keyword only for conditional compilation. Constants can also be undefined, in which case they have a value of `Nothing`.

Example

This example uses the `#Const` directive.

```
#Const MyLocation = "USA"
#Const Version = "8.0.0012"
#Const CustomerNumber = 36
```

See also

- /define (Visual Basic)
- #If...Then...#Else Directives
- Const Statement
- Conditional Compilation
- If...Then...Else Statement

#ExternalSource Directive

4/2/2019 • 2 minutes to read • Edit Online

Indicates a mapping between specific lines of source code and text external to the source.

Syntax

```
#ExternalSource( StringLiteral , IntLiteral )
    [ LogicalLine+ ]
#End ExternalSource
```

Parts

`StringLiteral`
The path to the external source.

`IntLiteral`
The line number of the first line of the external source.

`LogicalLine`
The line where the error occurs in the external source.

`#End ExternalSource`
Terminates the `#ExternalSource` block.

Remarks

This directive is used only by the compiler and the debugger.

A source file may include external source directives, which indicate a mapping between specific lines of code in the source file and text external to the source, such as an .aspx file. If errors are encountered in the designated source code during compilation, they are identified as coming from the external source.

External source directives have no effect on compilation and cannot be nested. They are intended for internal use by the application only.

See also

- Conditional Compilation

#If...Then...#Else Directives

8/22/2019 • 2 minutes to read • Edit Online

Conditionally compiles selected blocks of Visual Basic code.

Syntax

```
#If expression Then
   statements
[ #ElseIf expression Then
   [ statements ]
...
#ElseIf expression Then
   [ statements ] ]
[ #Else
   [ statements ] ]
#End If
```

Parts

`expression`
Required for `#If` and `#ElseIf` statements, optional elsewhere. Any expression, consisting exclusively of one or more conditional compiler constants, literals, and operators, that evaluates to `True` or `False`.

`statements`
Required for `#If` statement block, optional elsewhere. Visual Basic program lines or compiler directives that are compiled if the associated expression evaluates to `True`.

`#End If`
Terminates the `#If` statement block.

Remarks

On the surface, the behavior of the `#If...Then...#Else` directives appears the same as that of the `If...Then...Else` statements. However, the `#If...Then...#Else` directives evaluate what is compiled by the compiler, whereas the `If...Then...Else` statements evaluate conditions at run time.

Conditional compilation is typically used to compile the same program for different platforms. It is also used to prevent debugging code from appearing in an executable file. Code excluded during conditional compilation is completely omitted from the final executable file, so it has no effect on size or performance.

Regardless of the outcome of any evaluation, all expressions are evaluated using `Option Compare Binary`. The `Option Compare` statement does not affect expressions in `#If` and `#ElseIf` statements.

> **NOTE**
> No single-line form of the `#If`, `#Else`, `#ElseIf`, and `#End If` directives exists. No other code can appear on the same line as any of the directives.

The statements within a conditional compilation block must be complete logical statements. For example, you cannot conditionally compile only the attributes of a function, but you can conditionally declare the function

along with its attributes:

```
#If DEBUG Then
<WebMethod()>
Public Function SomeFunction() As String
#Else
<WebMethod(CacheDuration:=86400)>
Public Function SomeFunction() As String
#End If
```

Example

This example uses the `#If...Then...#Else` construct to determine whether to compile certain statements.

```
#Const CustomerNumber = 36
#If CustomerNumber = 35 Then
        ' Insert code to be compiled for customer # 35.
#ElseIf CustomerNumber = 36 Then
        ' Insert code to be compiled for customer # 36.
#Else
        ' Insert code to be compiled for all other customers.
#End If
```

See also

- #Const Directive
- If...Then...Else Statement
- Conditional Compilation
- System.Diagnostics.ConditionalAttribute

#Region Directive

4/2/2019 • 2 minutes to read • Edit Online

Collapses and hides sections of code in Visual Basic files.

Syntax

```
#Region "identifier_string"
#End Region
```

Parts

TERM	DEFINITION
`identifier_string`	Required. String that acts as the title of a region when it is collapsed. Regions are collapsed by default.
`#End Region`	Terminates the `#Region` block.

Remarks

Use the `#Region` directive to specify a block of code to expand or collapse when using the outlining feature of the Visual Studio Code Editor. You can place, or *nest*, regions within other regions to group similar regions together.

Example

This example uses the `#Region` directive.

```
#Region "MathFunctions"
    ' Insert code for the Math functions here.
#End Region
```

See also

- #If...Then...#Else Directives
- Outlining
- How to: Collapse and Hide Sections of Code

Functions (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

The topics in this section contain tables of the Visual Basic run-time member functions.

> **NOTE**
> You can also create functions and call them. For more information, see Function Statement and How to: Create a Procedure that Returns a Value.

In This Section

Conversion Functions

Math Functions

String Functions

Type Conversion Functions

CType Function

Related Sections

Visual Basic Language Reference

Visual Basic

Conversion functions (Visual Basic)

1/23/2019 • 2 minutes to read • Edit Online

Asc
AscW
CBool Function
CByte Function
CChar Function
CDate Function
CDbl Function
CDec Function
Chr
ChrW
CInt Function
CLng Function
CObj Function
CSByte Function
CShort Function
CSng Function
CStr Function
CType Function
CUInt Function
CULng Function
CUShort Function
Format
Hex
Oct
Str
Val

See also

- Type Conversion Functions
- Converting Data Types

Math Functions (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

The methods of the System.Math class provide trigonometric, logarithmic, and other common mathematical functions.

Remarks

The following table lists methods of the System.Math class. You can use these in a Visual Basic program.

.NET METHOD	DESCRIPTION
Abs	Returns the absolute value of a number.
Acos	Returns the angle whose cosine is the specified number.
Asin	Returns the angle whose sine is the specified number.
Atan	Returns the angle whose tangent is the specified number.
Atan2	Returns the angle whose tangent is the quotient of two specified numbers.
BigMul	Returns the full product of two 32-bit numbers.
Ceiling	Returns the smallest integral value that's greater than or equal to the specified `Decimal` or `Double`.
Cos	Returns the cosine of the specified angle.
Cosh	Returns the hyperbolic cosine of the specified angle.
DivRem	Returns the quotient of two 32-bit or 64-bit signed integers, and also returns the remainder in an output parameter.
Exp	Returns e (the base of natural logarithms) raised to the specified power.
Floor	Returns the largest integer that's less than or equal to the specified `Decimal` or `Double` number.
IEEERemainder	Returns the remainder that results from the division of a specified number by another specified number.
Log	Returns the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Log10	Returns the base 10 logarithm of a specified number.
Max	Returns the larger of two numbers.

.NET METHOD	DESCRIPTION
Min	Returns the smaller of two numbers.
Pow	Returns a specified number raised to the specified power.
Round	Returns a `Decimal` or `Double` value rounded to the nearest integral value or to a specified number of fractional digits.
Sign	Returns an `Integer` value indicating the sign of a number.
Sin	Returns the sine of the specified angle.
Sinh	Returns the hyperbolic sine of the specified angle.
Sqrt	Returns the square root of a specified number.
Tan	Returns the tangent of the specified angle.
Tanh	Returns the hyperbolic tangent of the specified angle.
Truncate	Calculates the integral part of a specified `Decimal` or `Double` number.

To use these functions without qualification, import the System.Math namespace into your project by adding the following code to the top of your source file:

```
Imports System.Math
```

Example

This example uses the Abs method of the Math class to compute the absolute value of a number.

```
' Returns 50.3.
Dim MyNumber1 As Double = Math.Abs(50.3)
' Returns 50.3.
Dim MyNumber2 As Double = Math.Abs(-50.3)
```

Example

This example uses the Atan method of the Math class to calculate the value of pi.

```
Public Function GetPi() As Double
    ' Calculate the value of pi.
    Return 4.0 * Math.Atan(1.0)
End Function
```

Example

This example uses the Cos method of the Math class to return the cosine of an angle.

```
Public Function Sec(ByVal angle As Double) As Double
    ' Calculate the secant of angle, in radians.
    Return 1.0 / Math.Cos(angle)
End Function
```

Example

This example uses the Exp method of the Math class to return e raised to a power.

```
Public Function Sinh(ByVal angle As Double) As Double
    ' Calculate hyperbolic sine of an angle, in radians.
    Return (Math.Exp(angle) - Math.Exp(-angle)) / 2.0
End Function
```

Example

This example uses the Log method of the Math class to return the natural logarithm of a number.

```
Public Function Asinh(ByVal value As Double) As Double
    ' Calculate inverse hyperbolic sine, in radians.
    Return Math.Log(value + Math.Sqrt(value * value + 1.0))
End Function
```

Example

This example uses the Round method of the Math class to round a number to the nearest integer.

```
' Returns 3.
Dim MyVar2 As Double = Math.Round(2.8)
```

Example

This example uses the Sign method of the Math class to determine the sign of a number.

```
' Returns 1.
Dim MySign1 As Integer = Math.Sign(12)
' Returns -1.
Dim MySign2 As Integer = Math.Sign(-2.4)
' Returns 0.
Dim MySign3 As Integer = Math.Sign(0)
```

Example

This example uses the Sin method of the Math class to return the sine of an angle.

```
Public Function Csc(ByVal angle As Double) As Double
    ' Calculate cosecant of an angle, in radians.
    Return 1.0 / Math.Sin(angle)
End Function
```

Example

This example uses the Sqrt method of the Math class to calculate the square root of a number.

```
' Returns 2.
Dim MySqr1 As Double = Math.Sqrt(4)
' Returns 4.79583152331272.
Dim MySqr2 As Double = Math.Sqrt(23)
' Returns 0.
Dim MySqr3 As Double = Math.Sqrt(0)
' Returns NaN (not a number).
Dim MySqr4 As Double = Math.Sqrt(-4)
```

Example

This example uses the Tan method of the Math class to return the tangent of an angle.

```
Public Function Ctan(ByVal angle As Double) As Double
    ' Calculate cotangent of an angle, in radians.
    Return 1.0 / Math.Tan(angle)
End Function
```

Requirements

Class: Math

Namespace: System

Assembly: mscorlib (in mscorlib.dll)

See also

- Rnd
- Randomize
- NaN
- Derived Math Functions
- Arithmetic Operators

String Functions (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following table lists the functions that Visual Basic provides to search and manipulate strings.

.NET FRAMEWORK METHOD	DESCRIPTION
Asc, AscW	Returns an `Integer` value representing the character code corresponding to a character.
Chr, ChrW	Returns the character associated with the specified character code.
Filter	Returns a zero-based array containing a subset of a `String` array based on specified filter criteria.
Format	Returns a string formatted according to instructions contained in a format `String` expression.
FormatCurrency	Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.
FormatDateTime	Returns a string expression representing a date/time value.
FormatNumber	Returns an expression formatted as a number.
FormatPercent	Returns an expression formatted as a percentage (that is, multiplied by 100) with a trailing % character.
InStr	Returns an integer specifying the start position of the first occurrence of one string within another.
InStrRev	Returns the position of the first occurrence of one string within another, starting from the right side of the string.
Join	Returns a string created by joining a number of substrings contained in an array.
LCase	Returns a string or character converted to lowercase.
Left	Returns a string containing a specified number of characters from the left side of a string.
Len	Returns an integer that contains the number of characters in a string.
LSet	Returns a left-aligned string containing the specified string adjusted to the specified length.
LTrim	Returns a string containing a copy of a specified string with no leading spaces.

.NET FRAMEWORK METHOD	DESCRIPTION
Mid	Returns a string containing a specified number of characters from a string.
Replace	Returns a string in which a specified substring has been replaced with another substring a specified number of times.
Right	Returns a string containing a specified number of characters from the right side of a string.
RSet	Returns a right-aligned string containing the specified string adjusted to the specified length.
RTrim	Returns a string containing a copy of a specified string with no trailing spaces.
Space	Returns a string consisting of the specified number of spaces.
Split	Returns a zero-based, one-dimensional array containing a specified number of substrings.
StrComp	Returns -1, 0, or 1, based on the result of a string comparison.
StrConv	Returns a string converted as specified.
StrDup	Returns a string or object consisting of the specified character repeated the specified number of times.
StrReverse	Returns a string in which the character order of a specified string is reversed.
Trim	Returns a string containing a copy of a specified string with no leading or trailing spaces.
UCase	Returns a string or character containing the specified string converted to uppercase.

You can use the Option Compare statement to set whether strings are compared using a case-insensitive text sort order determined by your system's locale (`Text`) or by the internal binary representations of the characters (`Binary`). The default text comparison method is `Binary` .

Example

This example uses the `UCase` function to return an uppercase version of a string.

```
' String to convert.
Dim lowerCase As String = "Hello World 1234"
' Returns "HELLO WORLD 1234".
Dim upperCase As String = UCase(lowerCase)
```

Example

This example uses the `LTrim` function to strip leading spaces and the `RTrim` function to strip trailing spaces from

a string variable. It uses the `Trim` function to strip both types of spaces.

```
' Initializes string.
Dim testString As String = "  <-Trim->  "
Dim trimString As String
' Returns "<-Trim->  ".
trimString = LTrim(testString)
' Returns "  <-Trim->".
trimString = RTrim(testString)
' Returns "<-Trim->".
trimString = LTrim(RTrim(testString))
' Using the Trim function alone achieves the same result.
' Returns "<-Trim->".
trimString = Trim(testString)
```

Example

This example uses the `Mid` function to return a specified number of characters from a string.

```
' Creates text string.
Dim testString As String = "Mid Function Demo"
' Returns "Mid".
Dim firstWord As String = Mid(testString, 1, 3)
' Returns "Demo".
Dim lastWord As String = Mid(testString, 14, 4)
' Returns "Function Demo".
Dim midWords As String = Mid(testString, 5)
```

Example

This example uses `Len` to return the number of characters in a string.

```
' Initializes variable.
Dim testString As String = "Hello World"
' Returns 11.
Dim testLen As Integer = Len(testString)
```

Example

This example uses the `InStr` function to return the position of the first occurrence of one string within another.

```
' String to search in.
Dim searchString As String = "XXpXXpXXPXXP"
' Search for "P".
Dim searchChar As String = "P"

Dim testPos As Integer
' A textual comparison starting at position 4. Returns 6.
testPos = InStr(4, searchString, searchChar, CompareMethod.Text)

' A binary comparison starting at position 1. Returns 9.
testPos = InStr(1, SearchString, SearchChar, CompareMethod.Binary)

' If Option Compare is not set, or set to Binary, return 9.
' If Option Compare is set to Text, returns 3.
testPos = InStr(searchString, searchChar)

' Returns 0.
testPos = InStr(1, searchString, "W")
```

Example

This example shows various uses of the `Format` function to format values using both `String` formats and user-defined formats. For the date separator (`/`), time separator (`:`), and the AM/PM indicators (`t` and `tt`), the actual formatted output displayed by your system depends on the locale settings the code is using. When times and dates are displayed in the development environment, the short time format and short date format of the code locale are used.

> **NOTE**
>
> For locales that use a 24-hour clock, the AM/PM indicators (`t` and `tt`) display nothing.

```
Dim testDateTime As Date = #1/27/2001 5:04:23 PM#
Dim testStr As String
' Returns current system time in the system-defined long time format.
testStr = Format(Now(), "Long Time")
' Returns current system date in the system-defined long date format.
testStr = Format(Now(), "Long Date")
' Also returns current system date in the system-defined long date
' format, using the single letter code for the format.
testStr = Format(Now(), "D")

' Returns the value of testDateTime in user-defined date/time formats.
' Returns "5:4:23".
testStr = Format(testDateTime, "h:m:s")
' Returns "05:04:23 PM".
testStr = Format(testDateTime, "hh:mm:ss tt")
' Returns "Saturday, Jan 27 2001".
testStr = Format(testDateTime, "dddd, MMM d yyyy")
' Returns "17:04:23".
testStr = Format(testDateTime, "HH:mm:ss")
' Returns "23".
testStr = Format(23)

' User-defined numeric formats.
' Returns "5,459.40".
testStr = Format(5459.4, "##,##0.00")
' Returns "334.90".
testStr = Format(334.9, "###0.00")
' Returns "500.00%".
testStr = Format(5, "0.00%")
```

See also

- Keywords
- Visual Basic Runtime Library Members
- String Manipulation Summary

Type Conversion Functions (Visual Basic)

4/28/2019 • 12 minutes to read • Edit Online

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Sometimes there is no call to a procedure to accomplish the conversion, which improves performance. Each function coerces an expression to a specific data type.

Syntax

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CSByte(expression)
CShort(expression)
CSng(expression)
CStr(expression)
CUInt(expression)
CULng(expression)
CUShort(expression)
```

Part

`expression`

Required. Any expression of the source data type.

Return Value Data Type

The function name determines the data type of the value it returns, as shown in the following table.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR `EXPRESSION` ARGUMENT
`CBool`	Boolean Data Type	Any valid `Char` or `String` or numeric expression.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR `expression` ARGUMENT
`CByte`	Byte Data Type	Byte.MinValue (0) through Byte.MaxValue (255) (unsigned); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to byte conversion with the `CByte` function; see the Remarks section for more information. See the CInt Example section for an example.
`CChar`	Char Data Type	Any valid `Char` or `String` expression; only first character of a `String` is converted; value can be 0 through 65535 (unsigned).
`CDate`	Date Data Type	Any valid representation of a date and time.
`CDbl`	Double Data Type	-1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.
`CDec`	Decimal Data Type	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001 (+/-1E-28).

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR `expression` ARGUMENT
`CInt`	Integer Data Type	Int32.MinValue (-2,147,483,648) through Int32.MaxValue (2,147,483,647); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to integer conversion with the `CInt` function; see the Remarks section for more information. See the CInt Example section for an example.
`CLng`	Long Data Type	Int64.MinValue (-9,223,372,036,854,775,808) through Int64.MaxValue (9,223,372,036,854,775,807); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to 64-bit integer conversion with the `CLng` function; see the Remarks section for more information. See the CInt Example section for an example.
`CObj`	Object Data Type	Any valid expression.
`CSByte`	SByte Data Type	SByte.MinValue (-128) through SByte.MaxValue (127); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to signed byte conversion with the `CSByte` function; see the Remarks section for more information. See the CInt Example section for an example.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR `EXPRESSION` ARGUMENT
`CShort`	Short Data Type	Int16.MinValue (-32,768) through Int16.MaxValue (32,767); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to 16-bit integer conversion with the `CShort` function; see the Remarks section for more information. See the CInt Example section for an example.
`CSng`	Single Data Type	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
`CStr`	String Data Type	Returns for `CStr` depend on the `expression` argument. See Return Values for the CStr Function.
`CUInt`	UInteger Data Type	UInt32.MinValue (0) through UInt32.MaxValue (4,294,967,295) (unsigned); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned integer conversion with the `CUInt` function; see the Remarks section for more information. See the CInt Example section for an example.
`CULng`	ULong Data Type	UInt64.MinValue (0) through UInt64.MaxValue (18,446,744,073,709,551,615) (unsigned); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned long integer conversion with the `CULng` function; see the Remarks section for more information. See the CInt Example section for an example.

FUNCTION NAME	RETURN DATA TYPE	RANGE FOR `expression` ARGUMENT
`CUShort`	UShort Data Type	UInt16.MinValue (0) through UInt16.MaxValue (65,535) (unsigned); fractional parts are rounded.[1] Starting with Visual Basic 15.8, Visual Basic optimizes the performance of floating-point to unsigned 16-bit integer conversion with the `CUShort` function; see the Remarks section for more information. See the CInt Example section for an example.

[1] Fractional parts can be subject to a special type of rounding called *banker's rounding*. See "Remarks" for more information.

Remarks

As a rule, you should use the Visual Basic type conversion functions in preference to the .NET Framework methods such as `ToString()`, either on the Convert class or on an individual type structure or class. The Visual Basic functions are designed for optimal interaction with Visual Basic code, and they also make your source code shorter and easier to read. In addition, the .NET Framework conversion methods do not always produce the same results as the Visual Basic functions, for example when converting `Boolean` to `Integer`. For more information, see Troubleshooting Data Types.

Starting with Visual Basic 15.8, the performance of floating-point-to-integer conversion is optimized when you pass the Single or Double value returned by the following methods to one of the integer conversion functions (`CByte`, `CShort`, `CInt`, `CLng`, `CSByte`, `CUShort`, `CUInt`, `CULng`):

- Conversion.Fix(Double)
- Conversion.Fix(Object)
- Conversion.Fix(Single)
- Conversion.Int(Double)
- Conversion.Int(Object)
- Conversion.Int(Single)
- Math.Ceiling(Double)
- Math.Floor(Double)
- Math.Round(Double)
- Math.Truncate(Double)

This optimization allows code that does a large number of integer conversions to run up to twice as fast. The following example illustrates these optimized floating-point-to-integer conversions:

```
Dim s As Single = 173.7619
Dim d As Double = s

Dim i1 As Integer = CInt(Fix(s))              ' Result: 173
Dim b1 As Byte = CByte(Int(d))                ' Result: 173
Dim s1 AS Short = CShort(Math.Truncate(s))    ' Result: 173
Dim i2 As Integer = CInt(Math.Ceiling(d))     ' Result: 174
Dim i3 As Integer = CInt(Math.Round(s))       ' Result: 174
```

Behavior

- **Coercion.** In general, you can use the data type conversion functions to coerce the result of an operation to a particular data type rather than the default data type. For example, use `CDec` to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic would normally take place.

- **Failed Conversions.** If the `expression` passed to the function is outside the range of the data type to which it is to be converted, an OverflowException occurs.

- **Fractional Parts.** When you convert a nonintegral value to an integral type, the integer conversion functions (`CByte` , `CInt` , `CLng` , `CSByte` , `CShort` , `CUInt` , `CULng` , and `CUShort`) remove the fractional part and round the value to the closest integer.

  If the fractional part is exactly 0.5, the integer conversion functions round it to the nearest even integer. For example, 0.5 rounds to 0, and 1.5 and 2.5 both round to 2. This is sometimes called *banker's rounding*, and its purpose is to compensate for a bias that could accumulate when adding many such numbers together.

  `CInt` and `CLng` differ from the Int and Fix functions, which truncate, rather than round, the fractional part of a number. Also, `Fix` and `Int` always return a value of the same data type as you pass in.

- **Date/Time Conversions.** Use the IsDate function to determine if a value can be converted to a date and time. `CDate` recognizes date literals and time literals but not numeric values. To convert a Visual Basic 6.0 `Date` value to a `Date` value in Visual Basic 2005 or later versions, you can use the DateTime.FromOADate method.

- **Neutral Date/Time Values.** The Date Data Type always contains both date and time information. For purposes of type conversion, Visual Basic considers 1/1/0001 (January 1 of the year 1) to be a *neutral value* for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a `Date` value to a string, `CStr` does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original `Date` value and can be recovered with functions such as DatePart function.

- **Culture Sensitivity.** The type conversion functions involving strings perform conversions based on the current culture settings for the application. For example, `CDate` recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date might not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

  If you need to convert to or from a string representation of a value in a format other than the one specified by your locale, you cannot use the Visual Basic type

conversion functions. To do this, use the `ToString(IFormatProvider)` and `Parse(String, IFormatProvider)` methods of that value's type. For example, use Double.Parse when converting a string to a `Double`, and use Double.ToString when converting a value of type `Double` to a string.

CType Function

The CType Function takes a second argument, `typename`, and coerces `expression` to `typename`, where `typename` can be any data type, structure, class, or interface to which there exists a valid conversion.

For a comparison of `CType` with the other type conversion keywords, see DirectCast Operator and TryCast Operator.

CBool Example

The following example uses the `CBool` function to convert expressions to `Boolean` values. If an expression evaluates to a nonzero value, `CBool` returns `True`; otherwise, it returns `False`.

```
Dim a, b, c As Integer
Dim check As Boolean
a = 5
b = 5
' The following line of code sets check to True.
check = CBool(a = b)
c = 0
' The following line of code sets check to False.
check = CBool(c)
```

CByte Example

The following example uses the `CByte` function to convert an expression to a `Byte`.

```
Dim aDouble As Double
Dim aByte As Byte
aDouble = 125.5678
' The following line of code sets aByte to 126.
aByte = CByte(aDouble)
```

CChar Example

The following example uses the `CChar` function to convert the first character of a `String` expression to a `Char` type.

```
Dim aString As String
Dim aChar As Char
' CChar converts only the first character of the string.
aString = "BCD"
' The following line of code sets aChar to "B".
aChar = CChar(aString)
```

The input argument to `CChar` must be of data type `Char` or `String`. You cannot use `CChar` to convert a number to a character, because `CChar` cannot accept a numeric data type. The

following example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses the InputBox function to obtain the string of digits, `CInt` to convert the string to type `Integer`, and `ChrW` to convert the number to type `Char`.

```
Dim someDigits As String
Dim codePoint As Integer
Dim thisChar As Char
someDigits = InputBox("Enter code point of character:")
codePoint = CInt(someDigits)
' The following line of code sets thisChar to the Char value of codePoint.
thisChar = ChrW(codePoint)
```

CDate Example

The following example uses the `CDate` function to convert strings to `Date` values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim aDateString, aTimeString As String
Dim aDate, aTime As Date
aDateString = "February 12, 1969"
aTimeString = "4:35:47 PM"
' The following line of code sets aDate to a Date value.
aDate = CDate(aDateString)
' The following line of code sets aTime to Date value.
aTime = CDate(aTimeString)
```

CDbl Example

```
Dim aDec As Decimal
Dim aDbl As Double
' The following line of code uses the literal type character D to make aDec a Decimal.
aDec = 234.456784D
' The following line of code sets aDbl to 1.9225456288E+1.
aDbl = CDbl(aDec * 8.2D * 0.01D)
```

CDec Example

The following example uses the `CDec` function to convert a numeric value to `Decimal`.

```
Dim aDouble As Double
Dim aDecimal As Decimal
aDouble = 10000000.0587
' The following line of code sets aDecimal to 10000000.0587.
aDecimal = CDec(aDouble)
```

CInt Example

The following example uses the `CInt` function to convert a value to `Integer`.

```
Dim aDbl As Double
Dim anInt As Integer
aDbl = 2345.5678
' The following line of code sets anInt to 2346.
anInt = CInt(aDbl)
```

CLng Example

The following example uses the `CLng` function to convert values to `Long`.

```
Dim aDbl1, aDbl2 As Double
Dim aLng1, aLng2 As Long
aDbl1 = 25427.45
aDbl2 = 25427.55
' The following line of code sets aLng1 to 25427.
aLng1 = CLng(aDbl1)
' The following line of code sets aLng2 to 25428.
aLng2 = CLng(aDbl2)
```

CObj Example

The following example uses the `CObj` function to convert a numeric value to `Object`. The `Object` variable itself contains only a four-byte pointer, which points to the `Double` value assigned to it.

```
Dim aDouble As Double
Dim anObject As Object
aDouble = 2.7182818284
' The following line of code sets anObject to a pointer to aDouble.
anObject = CObj(aDouble)
```

CSByte Example

The following example uses the `CSByte` function to convert a numeric value to `SByte`.

```
Dim aDouble As Double
Dim anSByte As SByte
aDouble = 39.501
' The following line of code sets anSByte to 40.
anSByte = CSByte(aDouble)
```

CShort Example

The following example uses the `CShort` function to convert a numeric value to `Short`.

```
Dim aByte As Byte
Dim aShort As Short
aByte = 100
' The following line of code sets aShort to 100.
aShort = CShort(aByte)
```

CSng Example

The following example uses the `CSng` function to convert values to `Single`.

```
Dim aDouble1, aDouble2 As Double
Dim aSingle1, aSingle2 As Single
aDouble1 = 75.3421105
aDouble2 = 75.3421567
' The following line of code sets aSingle1 to 75.34211.
aSingle1 = CSng(aDouble1)
' The following line of code sets aSingle2 to 75.34216.
aSingle2 = CSng(aDouble2)
```

CStr Example

The following example uses the `CStr` function to convert a numeric value to `String`.

```
Dim aDouble As Double
Dim aString As String
aDouble = 437.324
' The following line of code sets aString to "437.324".
aString = CStr(aDouble)
```

The following example uses the `CStr` function to convert `Date` values to `String` values.

```
Dim aDate As Date
Dim aString As String
' The following line of code generates a COMPILER ERROR because of invalid format.
' aDate = #February 12, 1969 00:00:00#
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' The following line of code sets the time component of aDate to midnight.
aDate = #2/12/1969#
' The following conversion suppresses the neutral time value of 00:00:00.
' The following line of code sets aString to "2/12/1969".
aString = CStr(aDate)
' The following line of code sets the time component of aDate to one second past
midnight.
aDate = #2/12/1969 12:00:01 AM#
' The time component becomes part of the converted value.
' The following line of code sets aString to "2/12/1969 12:00:01 AM".
aString = CStr(aDate)
```

`CStr` always renders a `Date` value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM". However, `CStr` suppresses the *neutral values* of 1/1/0001 for the date and 00:00:00 for the time.

For more detail on the values returned by `CStr`, see Return Values for the CStr Function.

CUInt Example

The following example uses the `CUInt` function to convert a numeric value to `UInteger`.

```
Dim aDouble As Double
Dim aUInteger As UInteger
aDouble = 39.501
' The following line of code sets aUInteger to 40.
aUInteger = CUInt(aDouble)
```

CULng Example

The following example uses the `CULng` function to convert a numeric value to `ULong`.

```
Dim aDouble As Double
Dim aULong As ULong
aDouble = 39.501
' The following line of code sets aULong to 40.
aULong = CULng(aDouble)
```

CUShort Example

The following example uses the `CUShort` function to convert a numeric value to `UShort`.

```
Dim aDouble As Double
Dim aUShort As UShort
aDouble = 39.501
' The following line of code sets aUShort to 40.
aUShort = CUShort(aDouble)
```

See also

- Asc
- AscW
- Chr
- ChrW
- Int
- Fix
- Format
- Hex
- Oct
- Str
- Val
- Conversion Functions
- Type Conversions in Visual Basic

Return Values for the CStr Function (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

The following table describes the return values for `CStr` for different data types of `expression`.

IF `EXPRESSION` TYPE IS	`CSTR` RETURNS
Boolean Data Type	A string containing "True" or "False".
Date Data Type	A string containing a `Date` value (date and time) in the short date format of your system.
Numeric Data Types	A string representing the number.

CStr and Date

The `Date` type always contains both date and time information. For purposes of type conversion, Visual Basic considers 1/1/0001 (January 1 of the year 1) to be a *neutral value* for the date, and 00:00:00 (midnight) to be a neutral value for the time. `CStr` does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original `Date` value and can be recovered with functions such as DatePart.

> **NOTE**
>
> The `CStr` function performs its conversion based on the current culture settings for the application. To get the string representation of a number in a particular culture, use the number's `ToString(IFormatProvider)` method. For example, use Double.ToString when converting a value of type `Double` to a `String`.

See also

- DatePart
- Type Conversion Functions
- Boolean Data Type
- Date Data Type

CType Function (Visual Basic)

3/1/2019 • 2 minutes to read • Edit Online

Returns the result of explicitly converting an expression to a specified data type, object, structure, class, or interface.

Syntax

```
CType(expression, typename)
```

Parts

`expression` Any valid expression. If the value of `expression` is outside the range allowed by `typename`, Visual Basic throws an exception.

`typename` Any expression that is legal within an `As` clause in a `Dim` statement, that is, the name of any data type, object, structure, class, or interface.

Remarks

TIP

You can also use the following functions to perform a type conversion:

- Type conversion functions such as `CByte`, `CDbl`, and `CInt` that perform a conversion to a specific data type. For more information, see Type Conversion Functions.
- DirectCast Operator or TryCast Operator. These operators require that one type inherit from or implement the other type. They can provide somewhat better performance than `CType` when converting to and from the `Object` data type.

`CType` is compiled inline, which means that the conversion code is part of the code that evaluates the expression. In some cases, the code runs faster because no procedures are called to perform the conversion.

If no conversion is defined from `expression` to `typename` (for example, from `Integer` to `Date`), Visual Basic displays a compile-time error message.

If a conversion fails at run time, the appropriate exception is thrown. If a narrowing conversion fails, an OverflowException is the most common result. If the conversion is undefined, an InvalidCastException in thrown. For example, this can happen if `expression` is of type `Object` and its run-time type has no conversion to `typename`.

If the data type of `expression` or `typename` is a class or structure you've defined, you can define `CType` on that class or structure as a conversion operator. This makes `CType` act as an *overloaded operator*. If you do this, you can control the behavior of conversions to and from your class or structure, including the exceptions that can be thrown.

Overloading

The `CType` operator can also be overloaded on a class or structure defined outside your code. If your code

converts to or from such a class or structure, be sure you understand the behavior of its `CType` operator. For more information, see Operator Procedures.

Converting Dynamic Objects

Type conversions of dynamic objects are performed by user-defined dynamic conversions that use the TryConvert or BindConvert methods. If you're working with dynamic objects, use the CTypeDynamic method to convert the dynamic object.

Example

The following example uses the `CType` function to convert an expression to the `Single` data type.

```
Dim testNumber As Long = 1000
' The following line of code sets testNewType to 1000.0.
Dim testNewType As Single = CType(testNumber, Single)
```

For additional examples, see Implicit and Explicit Conversions.

See also

- OverflowException
- InvalidCastException
- Type Conversion Functions
- Conversion Functions
- Operator Statement
- How to: Define a Conversion Operator
- Type Conversion in the .NET Framework

Modifiers (Visual Basic)

10/24/2018 • 2 minutes to read • Edit Online

The topics in this section document Visual Basic run-time modifiers.

In This Section

Ansi

Assembly

Async

Auto

ByRef

ByVal

Default

Friend

In

Iterator

Key

Module <keyword>

MustInherit

MustOverride

Narrowing

NotInheritable

NotOverridable

Optional

Out

Overloads

Overridable

Overrides

ParamArray

Partial

Private

Private Protected

Protected

Protected Friend

Public

ReadOnly

Shadows

Shared

Static

Unicode

Widening

WithEvents

WriteOnly

Related Sections

Visual Basic Language Reference

Visual Basic

Ansi (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that Visual Basic should marshal all strings to American National Standards Institute (ANSI) values regardless of the name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The Declare Statement creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Ansi` modifier specifies that Visual Basic should marshal all strings to ANSI values and should look up the procedure without modifying its name during the search.

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Ansi` modifier can be used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See also

- Auto
- Unicode
- Keywords

Assembly (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that an attribute at the beginning of a source file applies to the entire assembly.

Remarks

Many attributes pertain to an individual programming element, such as a class or property. You apply such an attribute by attaching the attribute block, within angle brackets (`< >`), directly to the declaration statement.

If an attribute pertains not only to the following element but to the entire assembly, you place the attribute block at the beginning of the source file and identify the attribute with the `Assembly` keyword. If it applies to the current assembly module, you use the Module keyword.

You can also apply an attribute to an assembly in the AssemblyInfo.vb file, in which case you do not have to use an attribute block in your main source-code file.

See also

- Module <keyword>
- Attributes overview

Async (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

The `Async` modifier indicates that the method or lambda expression that it modifies is asynchronous. Such methods are referred to as *async methods*.

An async method provides a convenient way to do potentially long-running work without blocking the caller's thread. The caller of an async method can resume its work without waiting for the async method to finish.

> **NOTE**
>
> The `Async` and `Await` keywords were introduced in Visual Studio 2012. For an introduction to async programming, see Asynchronous Programming with Async and Await.

The following example shows the structure of an async method. By convention, async method names end in "Async."

```
Public Async Function ExampleMethodAsync() As Task(Of Integer)
    ' . . .

    ' At the Await expression, execution in this method is suspended and,
    ' if AwaitedProcessAsync has not already finished, control returns
    ' to the caller of ExampleMethodAsync. When the awaited task is
    ' completed, this method resumes execution.
    Dim exampleInt As Integer = Await AwaitedProcessAsync()

    ' . . .

    ' The return statement completes the task. Any method that is
    ' awaiting ExampleMethodAsync can now get the integer result.
    Return exampleInt
End Function
```

Typically, a method modified by the `Async` keyword contains at least one Await expression or statement. The method runs synchronously until it reaches the first `Await`, at which point it suspends until the awaited task completes. In the meantime, control is returned to the caller of the method. If the method doesn't contain an `Await` expression or statement, the method isn't suspended and executes as a synchronous method does. A compiler warning alerts you to any async methods that don't contain `Await` because that situation might indicate an error. For more information, see the compiler error.

The `Async` keyword is an unreserved keyword. It is a keyword when it modifies a method or a lambda expression. In all other contexts, it is interpreted as an identifier.

Return Types

An async method is either a Sub procedure, or a Function procedure that has a return type of Task or Task<TResult>. The method cannot declare any ByRef parameters.

You specify `Task(Of TResult)` for the return type of an async method if the Return statement of the method has an operand of type TResult. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `Await` statement that's awaiting the `Task` doesn't produce a result value.

Async subroutines are used primarily to define event handlers where a `Sub` procedure is required. The caller of an async subroutine can't await it and can't catch exceptions that the method throws.

For more information and examples, see Async Return Types.

Example

The following examples show an async event handler, an async lambda expression, and an async method. For a full example that uses these elements, see Walkthrough: Accessing the Web by Using Async and Await. You can download the walkthrough code from Developer Code Samples.

```
' An event handler must be a Sub procedure.
Async Sub button1_Click(sender As Object, e As RoutedEventArgs) Handles button1.Click
    textBox1.Clear()
    ' SumPageSizesAsync is a method that returns a Task.
    Await SumPageSizesAsync()
    textBox1.Text = vbCrLf & "Control returned to button1_Click."
End Sub

' The following async lambda expression creates an equivalent anonymous
' event handler.
AddHandler button1.Click, Async Sub(sender, e)
                              textBox1.Clear()
                              ' SumPageSizesAsync is a method that returns a Task.
                              Await SumPageSizesAsync()
                              textBox1.Text = vbCrLf & "Control returned to button1_Click."
                          End Sub

' The following async method returns a Task(Of T).
' A typical call awaits the Byte array result:
'      Dim result As Byte() = Await GetURLContents("https://msdn.com")
Private Async Function GetURLContentsAsync(url As String) As Task(Of Byte())

    ' The downloaded resource ends up in the variable named content.
    Dim content = New MemoryStream()

    ' Initialize an HttpWebRequest for the current URL.
    Dim webReq = CType(WebRequest.Create(url), HttpWebRequest)

    ' Send the request to the Internet resource and wait for
    ' the response.
    Using response As WebResponse = Await webReq.GetResponseAsync()
        ' Get the data stream that is associated with the specified URL.
        Using responseStream As Stream = response.GetResponseStream()
            ' Read the bytes in responseStream and copy them to content.
            ' CopyToAsync returns a Task, not a Task<T>.
            Await responseStream.CopyToAsync(content)
        End Using
    End Using

    ' Return the result as a byte array.
    Return content.ToArray()
End Function
```

See also

- AsyncStateMachineAttribute
- Await Operator
- Asynchronous Programming with Async and Await
- Walkthrough: Accessing the Web by Using Async and Await

Auto (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that Visual Basic should marshal strings according to .NET Framework rules based on the external name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it must have to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The Declare Statement creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information for marshaling strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Auto` modifier specifies that Visual Basic should marshal strings according to .NET Framework rules, and that it should determine the base character set of the run-time platform and possibly modify the external procedure name if the initial search fails. For more information, see "Character Sets" in Declare Statement.

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Auto` modifier can be used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See also

- Ansi
- Unicode
- Keywords

ByRef (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that an argument is passed in such a way that the called procedure can change the value of a variable underlying the argument in the calling code.

Remarks

The `ByRef` modifier can be used in these contexts:

Declare Statement

Function Statement

Sub Statement

See also

- Keywords
- Passing Arguments by Value and by Reference

ByVal (Visual Basic)

8/21/2019 • 2 minutes to read • Edit Online

Specifies that an argument is passed by value, so that the called procedure or property cannot change the value of a variable underlying the argument in the calling code. If no modifier is specified, ByVal is the default.

> **NOTE**
> Because it is the default, you do not have to explicitly specify the `ByVal` keyword in method signatures. It tends to produce noisy code and often leads to the non-default `ByRef` keyword being overlooked.

Remarks

The `ByVal` modifier can be used in these contexts:

Declare Statement

Function Statement

Operator Statement

Property Statement

Sub Statement

Example

The following example demonstrates the use of the `ByVal` parameter passing mechanism with a reference type argument. In the example, the argument is `c1`, an instance of class `Class1`. `ByVal` prevents the code in the procedures from changing the underlying value of the reference argument, `c1`, but does not protect the accessible fields and properties of `c1`.

```
Module Module1

    Sub Main()

        ' Declare an instance of the class and assign a value to its field.
        Dim c1 As New Class1()
        c1.Field = 5
        Console.WriteLine(c1.Field)
        ' Output: 5

        ' ByVal does not prevent changing the value of a field or property.
        ChangeFieldValue(c1)
        Console.WriteLine(c1.Field)
        ' Output: 500

        ' ByVal does prevent changing the value of c1 itself.
        ChangeClassReference(c1)
        Console.WriteLine(c1.Field)
        ' Output: 500

        Console.ReadKey()
    End Sub

    Public Sub ChangeFieldValue(ByVal cls As Class1)
        cls.Field = 500
    End Sub

    Public Sub ChangeClassReference(ByVal cls As Class1)
        cls = New Class1()
        cls.Field = 1000
    End Sub

    Public Class Class1
        Public Field As Integer
    End Class

End Module
```

See also

- Keywords
- Passing Arguments by Value and by Reference

Default (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Identifies a property as the default property of its class, structure, or interface.

Remarks

A class, structure, or interface can designate at most one of its properties as the *default property*, provided that property takes at least one parameter. If code makes a reference to a class or structure without specifying a member, Visual Basic resolves that reference to the default property.

Default properties can result in a small reduction in source code-characters, but they can make your code more difficult to read. If the calling code is not familiar with your class or structure, when it makes a reference to the class or structure name it cannot be certain whether that reference accesses the class or structure itself, or a default property. This can lead to compiler errors or subtle run-time logic errors.

You can somewhat reduce the chance of default property errors by always using the Option Strict Statement to set compiler type checking to `On`.

If you are planning to use a predefined class or structure in your code, you must determine whether it has a default property, and if so, what its name is.

Because of these disadvantages, you should consider not defining default properties. For code readability, you should also consider always referring to all properties explicitly, even default properties.

The `Default` modifier can be used in this context:

Property Statement

See also

- How to: Declare and Call a Default Property in Visual Basic
- Keywords

Friend (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies that one or more declared programming elements are accessible only from within the assembly that contains their declaration.

Remarks

In many cases, you want programming elements such as classes and structures to be used by the entire assembly, not only by the component that declares them. However, you might not want them to be accessible by code outside the assembly (for example, if the application is proprietary). If you want to limit access to an element in this way, you can declare it by using the `Friend` modifier.

Code in other classes, structures, and modules that are compiled to the same assembly can access all the `Friend` elements in that assembly.

`Friend` access is often the preferred level for an application's programming elements, and `Friend` is the default access level of an interface, a module, a class, or a structure.

You can use `Friend` only at the module, interface, or namespace level. Therefore, the declaration context for a `Friend` element must be a source file, a namespace, an interface, a module, a class, or a structure; it can't be a procedure.

> **NOTE**
> You can also use the Protected Friend access modifier, which makes a class member accessible from within that class, from derived classes, and from the same assembly in which the class is defined. To restrict access to a member from within its class and from derived classes in the same assembly, you use the Private Protected access modifier.

For a comparison of `Friend` and the other access modifiers, see Access levels in Visual Basic.

> **NOTE**
> You can specify that another assembly is a friend assembly, which allows it to access all types and members that are marked as `Friend`. For more information, see Friend Assemblies.

Example

The following class uses the `Friend` modifier to allow other programming elements within the same assembly to access certain members.

```
Class CustomerInfo

    Private p_CustomerID As Integer

    Public ReadOnly Property CustomerID() As Integer
        Get
            Return p_CustomerID
        End Get
    End Property

    ' Allow friend access to the empty constructor.
    Friend Sub New()

    End Sub

    ' Require that a customer identifier be specified for the public constructor.
    Public Sub New(ByVal customerID As Integer)
        p_CustomerID = customerID
    End Sub

    ' Allow friend programming elements to set the customer identifier.
    Friend Sub SetCustomerID(ByVal customerID As Integer)
        p_CustomerID = customerID
    End Sub
End Class
```

Usage

You can use the `Friend` modifier in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Module Statement

Property Statement

Structure Statement

Sub Statement

See also

- InternalsVisibleToAttribute
- Public
- Protected
- Private

- Private Protected
- Protected Friend
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

In (Generic Modifier) (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

For generic type parameters, the `In` keyword specifies that the type parameter is contravariant.

Remarks

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types.

For more information, see Covariance and Contravariance.

Rules

You can use the `In` keyword in generic interfaces and delegates.

A type parameter can be declared contravariant in a generic interface or delegate if it is used only as a type of method arguments and not used as a method return type. `ByRef` parameters cannot be covariant or contravariant.

Covariance and contravariance are supported for reference types and not supported for value types.

In Visual Basic, you cannot declare events in contravariant interfaces without specifying the delegate type. Also, contravariant interfaces cannot have nested classes, enums, or structures, but they can have nested interfaces.

Behavior

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, because in .NET Framework 4, in the IComparer<T> interface, type T is contravariant, you can assign an object of the `IComparer(Of Person)` type to an object of the `IComparer(Of Employee)` type without using any special conversion methods if `Person` inherits `Employee`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

Example

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

```
' Contravariant interface.
Interface IContravariant(Of In A)
End Interface

' Extending contravariant interface.
Interface IExtContravariant(Of In A)
    Inherits IContravariant(Of A)
End Interface

' Implementing contravariant interface.
Class Sample(Of A)
    Implements IContravariant(Of A)
End Class

Sub Main()
    Dim iobj As IContravariant(Of Object) = New Sample(Of Object)()
    Dim istr As IContravariant(Of String) = New Sample(Of String)()

    ' You can assign iobj to istr, because
    ' the IContravariant interface is contravariant.
    istr = iobj
End Sub
```

Example

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also shows how you can implicitly convert a delegate type.

```
' Contravariant delegate.
Public Delegate Sub DContravariant(Of In A)(ByVal argument As A)

' Methods that match the delegate signature.
Public Shared Sub SampleControl(ByVal control As Control)
End Sub

Public Shared Sub SampleButton(ByVal control As Button)
End Sub

Private Sub Test()

    ' Instantiating the delegates with the methods.
    Dim dControl As DContravariant(Of Control) =
        AddressOf SampleControl
    Dim dButton As DContravariant(Of Button) =
        AddressOf SampleButton

    ' You can assign dControl to dButton
    ' because the DContravariant delegate is contravariant.
    dButton = dControl

    ' Invoke the delegate.
    dButton(New Button())
End Sub
```

See also

- Variance in Generic Interfaces
- Out

Iterator (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that a function or `Get` accessor is an iterator.

Remarks

An *iterator* performs a custom iteration over a collection. An iterator uses the Yield statement to return each element in the collection one at a time. When a `Yield` statement is reached, the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

An iterator can be implemented as a function or as a `Get` accessor of a property definition. The `Iterator` modifier appears in the declaration of the iterator function or `Get` accessor.

You call an iterator from client code by using a For Each...Next Statement.

The return type of an iterator function or `Get` accessor can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

An iterator cannot have any `ByRef` parameters.

An iterator cannot occur in an event, instance constructor, static constructor, or static destructor.

An iterator can be an anonymous function. For more information, see Iterators.

Usage

The `Iterator` modifier can be used in these contexts:

- Function Statement
- Property Statement

Example

The following example demonstrates an iterator function. The iterator function has a `Yield` statement that is inside a For…Next loop. Each iteration of the For Each statement body in `Main` creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `Yield` statement, which occurs during the next iteration of the `For…Next` loop.

```
Sub Main()
    For Each number In Power(2, 8)
        Console.Write(number & " ")
    Next
    ' Output: 2 4 8 16 32 64 128 256
    Console.ReadKey()
End Sub

Private Iterator Function Power(
ByVal base As Integer, ByVal highExponent As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    Dim result = 1

    For counter = 1 To highExponent
        result = result * base
        Yield result
    Next
End Function
```

Example

The following example demonstrates a `Get` accessor that is an iterator. The `Iterator` modifier is in the property declaration.

```
Sub Main()
    Dim theGalaxies As New Galaxies
    For Each theGalaxy In theGalaxies.NextGalaxy
        With theGalaxy
            Console.WriteLine(.Name & "  " & .MegaLightYears)
        End With
    Next
    Console.ReadKey()
End Sub

Public Class Galaxies
    Public ReadOnly Iterator Property NextGalaxy _
    As System.Collections.Generic.IEnumerable(Of Galaxy)
        Get
            Yield New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400}
            Yield New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25}
            Yield New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0}
            Yield New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        End Get
    End Property
End Class

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class
```

For additional examples, see Iterators.

See also

- IteratorStateMachineAttribute
- Iterators
- Yield Statement

Key (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

The `Key` keyword enables you to specify behavior for properties of anonymous types. Only properties you designate as key properties participate in tests of equality between anonymous type instances, or calculation of hash code values. The values of key properties cannot be changed.

You designate a property of an anonymous type as a key property by placing the keyword `Key` in front of its declaration in the initialization list. In the following example, `Airline` and `FlightNo` are key properties, but `Gate` is not.

```
Dim flight1 = New With {Key .Airline = "Blue Yonder Airlines",
                        Key .FlightNo = 3554, .Gate = "C33"}
```

When a new anonymous type is created, it inherits directly from Object. The compiler overrides three inherited members: Equals, GetHashCode, and ToString. The override code that is produced for Equals and GetHashCode is based on key properties. If there are no key properties in the type, GetHashCode and Equals are not overridden.

Equality

Two anonymous type instances are equal if they are instances of the same type and if the values of their key properties are equal. In the following examples, `flight2` is equal to `flight1` from the previous example because they are instances of the same anonymous type and they have matching values for their key properties. However, `flight3` is not equal to `flight1` because it has a different value for a key property, `FlightNo`. Instance `flight4` is not the same type as `flight1` because they designate different properties as key properties.

```
Dim flight2 = New With {Key .Airline = "Blue Yonder Airlines",
                        Key .FlightNo = 3554, .Gate = "D14"}
' The following statement displays True. The values of the non-key
' property, Gate, do not have to be equal.
Console.WriteLine(flight1.Equals(flight2))

Dim flight3 = New With {Key .Airline = "Blue Yonder Airlines",
                        Key .FlightNo = 431, .Gate = "C33"}
' The following statement displays False, because flight3 has a
' different value for key property FlightNo.
Console.WriteLine(flight1.Equals(flight3))

Dim flight4 = New With {Key .Airline = "Blue Yonder Airlines",
                        .FlightNo = 3554, .Gate = "C33"}
' The following statement displays False. Instance flight4 is not the
' same type as flight1 because they have different key properties.
' FlightNo is a key property of flight1 but not of flight4.
Console.WriteLine(flight1.Equals(flight4))
```

If two instances are declared with only non-key properties, identical in name, type, order, and value, the two instances are not equal. An instance without key properties is equal only to itself.

For more information about the conditions under which two anonymous type instances are instances of the same anonymous type, see Anonymous Types.

Hash Code Calculation

Like Equals, the hash function that is defined in GetHashCode for an anonymous type is based on the key properties of the type. The following examples show the interaction between key properties and hash code values.

Instances of an anonymous type that have the same values for all key properties have the same hash code value, even if non-key properties do not have matching values. The following statement returns `True`.

```
Console.WriteLine(flight1.GetHashCode = flight2.GetHashCode)
```

Instances of an anonymous type that have different values for one or more key properties have different hash code values. The following statement returns `False`.

```
Console.WriteLine(flight1.GetHashCode = flight3.GetHashCode)
```

Instances of anonymous types that designate different properties as key properties are not instances of the same type. They have different hash code values even when the names and values of all properties are the same. The following statement returns `False`.

```
Console.WriteLine(flight1.GetHashCode = flight4.GetHashCode)
```

Read-Only Values

The values of key properties cannot be changed. For example, in `flight1` in the earlier examples, the `Airline` and `FlightNo` fields are read-only, but `Gate` can be changed.

```
' The following statement will not compile, because FlightNo is a key
' property and cannot be changed.
' flight1.FlightNo = 1234
'
' Gate is not a key property. Its value can be changed.
flight1.Gate = "C5"
```

See also

- Anonymous Type Definition
- How to: Infer Property Names and Types in Anonymous Type Declarations
- Anonymous Types

Module <keyword> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that an attribute at the beginning of a source file applies to the current assembly module.

Remarks

Many attributes pertain to an individual programming element, such as a class or property. You apply such an attribute by attaching the attribute block, within angle brackets (`< >`), directly to the declaration statement.

If an attribute pertains not only to the following element but to the current assembly module, you place the attribute block at the beginning of the source file and identify the attribute with the `Module` keyword. If it applies to the entire assembly, you use the Assembly keyword.

The `Module` modifier is not the same as the Module Statement.

See also

- Assembly
- Module Statement
- Attributes overview

MustInherit (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that a class can be used only as a base class and that you cannot create an object directly from it.

Remarks

The purpose of a *base class* (also known as an *abstract class*) is to define functionality that is common to all the classes derived from it. This saves the derived classes from having to redefine the common elements. In some cases, this common functionality is not complete enough to make a usable object, and each derived class defines the missing functionality. In such a case, you want the consuming code to create objects only from the derived classes. You use `MustInherit` on the base class to enforce this.

Another use of a `MustInherit` class is to restrict a variable to a set of related classes. You can define a base class and derive all these related classes from it. The base class does not need to provide any functionality common to all the derived classes, but it can serve as a filter for assigning values to variables. If your consuming code declares a variable as the base class, Visual Basic allows you to assign only an object from one of the derived classes to that variable.

The .NET Framework defines several `MustInherit` classes, among them Array, Enum, and ValueType. ValueType is an example of a base class that restricts a variable. All value types derive from ValueType. If you declare a variable as ValueType, you can assign only value types to that variable.

Rules

- **Declaration Context.** You can use `MustInherit` only in a `Class` statement.
- **Combined Modifiers.** You cannot specify `MustInherit` together with `NotInheritable` in the same declaration.

Example

The following example illustrates both forced inheritance and forced overriding. The base class `shape` defines a variable `acrossLine`. The classes `circle` and `square` derive from `shape`. They inherit the definition of `acrossLine`, but they must define the function `area` because that calculation is different for each kind of shape.

```
Public MustInherit Class shape
    Public acrossLine As Double
    Public MustOverride Function area() As Double
End Class
Public Class circle : Inherits shape
    Public Overrides Function area() As Double
        Return Math.PI * acrossLine
    End Function
End Class
Public Class square : Inherits shape
    Public Overrides Function area() As Double
        Return acrossLine * acrossLine
    End Function
End Class
Public Class consumeShapes
    Public Sub makeShapes()
        Dim shape1, shape2 As shape
        shape1 = New circle
        shape2 = New square
    End Sub
End Class
```

You can declare `shape1` and `shape2` to be of type `shape`. However, you cannot create an object from `shape` because it lacks the functionality of the function `area` and is marked `MustInherit`.

Because they are declared as `shape`, the variables `shape1` and `shape2` are restricted to objects from the derived classes `circle` and `square`. Visual Basic does not allow you to assign any other object to these variables, which gives you a high level of type safety.

Usage

The `MustInherit` modifier can be used in this context:

Class Statement

See also

- Inherits Statement
- NotInheritable
- Keywords
- Inheritance Basics

MustOverride (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that a property or procedure is not implemented in this class and must be overridden in a derived class before it can be used.

Remarks

You can use `MustOverride` only in a property or procedure declaration statement. The property or procedure that specifies `MustOverride` must be a member of a class, and the class must be marked MustInherit.

Rules

- **Incomplete Declaration.** When you specify `MustOverride`, you do not supply any additional lines of code for the property or procedure, not even the `End Function`, `End Property`, or `End Sub` statement.
- **Combined Modifiers.** You cannot specify `MustOverride` together with `NotOverridable`, `Overridable`, or `Shared` in the same declaration.
- **Shadowing and Overriding.** Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see Shadowing in Visual Basic.
- **Alternate Terms.** An element that cannot be used except in an override is sometimes called a *pure virtual* element.

The `MustOverride` modifier can be used in these contexts:

Function Statement

Property Statement

Sub Statement

See also

- NotOverridable
- Overridable
- Overrides
- MustInherit
- Keywords
- Shadowing in Visual Basic

Narrowing (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that a conversion operator (`CType`) converts a class or structure to a type that might not be able to hold some of the possible values of the original class or structure.

Converting with the Narrowing Keyword

The conversion procedure must specify `Public Shared` in addition to `Narrowing` .

Narrowing conversions do not always succeed at run time, and can fail or incur data loss. Examples are `Long` to `Integer` , `String` to `Date` , and a base type to a derived type. This last conversion is narrowing because the base type might not contain all the members of the derived type and thus is not an instance of the derived type.

If `Option Strict` is `On` , the consuming code must use `CType` for all narrowing conversions.

The `Narrowing` keyword can be used in this context:

Operator Statement

See also

- Operator Statement
- Widening
- Widening and Narrowing Conversions
- How to: Define an Operator
- CType Function
- Option Strict Statement

NotInheritable (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that a class cannot be used as a base class.

Remarks

Alternate Terms. A class that cannot be inherited is sometimes called a *sealed* class.

The `NotInheritable` modifier can be used in this context:

Class Statement

See also

- Inherits Statement
- MustInherit
- Keywords

NotOverridable (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that a property or procedure cannot be overridden in a derived class.

Remarks

The `NotOverridable` modifier prevents a property or method from being overridden in a derived class. The Overridable modifier allows a property or method in a class to be overridden in a derived class. For more information, see Inheritance Basics.

If the `Overridable` or `NotOverridable` modifier is not specified, the default setting depends on whether the property or method overrides a base class property or method. If the property or method overrides a base class property or method, the default setting is `Overridable`; otherwise, it is `NotOverridable`.

An element that cannot be overridden is sometimes called a *sealed* element.

You can use `NotOverridable` only in a property or procedure declaration statement. You can specify `NotOverridable` only on a property or procedure that overrides another property or procedure, that is, only in combination with `Overrides`.

Combined Modifiers

You cannot specify `Overridable` or `NotOverridable` for a `Private` method.

You cannot specify `NotOverridable` together with `MustOverride`, `Overridable`, or `Shared` in the same declaration.

Usage

The `NotOverridable` modifier can be used in these contexts:

Function Statement

Property Statement

Sub Statement

See also

- Modifiers
- Inheritance Basics
- MustOverride
- Overridable
- Overrides
- Keywords
- Shadowing in Visual Basic

Optional (Visual Basic)

7/30/2019 • 2 minutes to read • Edit Online

Specifies that a procedure argument can be omitted when the procedure is called.

Remarks

For each optional parameter, you must specify a constant expression as the default value of that parameter. If the expression evaluates to Nothing, the default value of the value data type is used as the default value of the parameter.

If the parameter list contains an optional parameter, every parameter that follows it must also be optional.

The `Optional` modifier can be used in these contexts:

- Declare Statement
- Function Statement
- Property Statement
- Sub Statement

> **NOTE**
> When calling a procedure with or without optional parameters, you can pass arguments by position or by name. For more information, see Passing Arguments by Position and by Name.

> **NOTE**
> You can also define a procedure with optional parameters by using overloading. If you have one optional parameter, you can define two overloaded versions of the procedure, one that accepts the parameter and one that doesn't. For more information, see Procedure Overloading.

Example

The following example defines a procedure that has an optional parameter.

```
Public Function FindMatches(ByRef values As List(Of String),
                            ByVal searchString As String,
                            Optional ByVal matchCase As Boolean = False) As List(Of String)

    Dim results As IEnumerable(Of String)

    If matchCase Then
        results = From v In values
                  Where v.Contains(searchString)
    Else
        results = From v In values
                  Where UCase(v).Contains(UCase(searchString))
    End If

    Return results.ToList()
End Function
```

Example

The following example demonstrates how to call a procedure with arguments passed by position and with arguments passed by name. The procedure has two optional parameters.

```
Private Sub TestParameters()
    ' Call the procedure with its arguments passed by position,
    studentInfo("Mary", 19, #9/21/1981#)

    ' Omit one optional argument by holding its place with a comma.
    studentInfo("Mary", , #9/21/1981#)

    ' Call the procedure with its arguments passed by name.
    studentInfo(age:=19, birth:=#9/21/1981#, name:="Mary")

    ' Supply an argument by position and an argument by name.
    studentInfo("Mary", birth:=#9/21/1981#)
End Sub

Private Sub studentInfo(ByVal name As String,
   Optional ByVal age As Short = 0,
   Optional ByVal birth As Date = #1/1/2000#)

    Console.WriteLine("name: " & name)
    Console.WriteLine("age: " & age)
    Console.WriteLine("birth date: " & birth)
    Console.WriteLine()
End Sub
```

See also

- Parameter List
- Optional Parameters
- Keywords

Out (Generic Modifier) (Visual Basic)

3/5/2019 • 2 minutes to read • Edit Online

For generic type parameters, the `Out` keyword specifies that the type is covariant.

Remarks

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types.

For more information, see Covariance and Contravariance.

Rules

You can use the `Out` keyword in generic interfaces and delegates.

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

  > **NOTE**
  > There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see Variance in Delegates and Using Variance for Func and Action Generic Delegates.

- The type parameter is not used as a generic constraint for the interface methods.

In a generic delegate, a type parameter can be declared covariant if it is used only as a method return type and not used for method arguments.

Covariance and contravariance are supported for reference types, but they are not supported for value types.

In Visual Basic, you cannot declare events in covariant interfaces without specifying the delegate type. Also, covariant interfaces cannot have nested classes, enums, or structures, but they can have nested interfaces.

Behavior

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in IEnumerable<T>, type T is covariant, you can assign an object of the `IEnumerable(Of String)` type to an object of the `IEnumerable(Of Object)` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

Example

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

```
' Covariant interface.
Interface ICovariant(Of Out R)
End Interface

' Extending covariant interface.
Interface IExtCovariant(Of Out R)
    Inherits ICovariant(Of R)
End Interface

' Implementing covariant interface.
Class Sample(Of R)
    Implements ICovariant(Of R)
End Class

Sub Main()
    Dim iobj As ICovariant(Of Object) = New Sample(Of Object)()
    Dim istr As ICovariant(Of String) = New Sample(Of String)()

    ' You can assign istr to iobj because
    ' the ICovariant interface is covariant.
    iobj = istr
End Sub
```

Example

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how you can use implicit conversion for delegate types.

```
' Covariant delegate.
Public Delegate Function DCovariant(Of Out R)() As R

' Methods that match the delegate signature.
Public Shared Function SampleControl() As Control
    Return New Control()
End Function

Public Shared Function SampleButton() As Button
    Return New Button()
End Function

Private Sub Test()

    ' Instantiating the delegates with the methods.
    Dim dControl As DCovariant(Of Control) =
        AddressOf SampleControl
    Dim dButton As DCovariant(Of Button) =
        AddressOf SampleButton

    ' You can assign dButton to dControl
    ' because the DCovariant delegate is covariant.
    dControl = dButton

    ' Invoke the delegate.
    dControl()
End Sub
```

See also

- Variance in Generic Interfaces
- In

Overloads (Visual Basic)

7/9/2019 • 2 minutes to read • Edit Online

Specifies that a property or procedure redeclares one or more existing properties or procedures with the same name.

Remarks

Overloading is the practice of supplying more than one definition for a given property or procedure name in the same scope. Redeclaring a property or procedure with a different signature is sometimes called *hiding by signature*.

Rules

- **Declaration Context.** You can use `Overloads` only in a property or procedure declaration statement.
- **Combined Modifiers.** You cannot specify `Overloads` together with Shadows in the same procedure declaration.
- **Required Differences.** The *signature* in this declaration must be different from the signature of every property or procedure that it overloads. The signature comprises the property or procedure name together with the following:
  - the number of parameters
  - the order of the parameters
  - the data types of the parameters
  - the number of type parameters (for a generic procedure)
  - the return type (only for a conversion operator procedure)

  All overloads must have the same name, but each must differ from all the others in one or more of the preceding respects. This allows the compiler to distinguish which version to use when code calls the property or procedure.
- **Disallowed Differences.** Changing one or more of the following is not valid for overloading a property or procedure, because they are not part of the signature:
  - whether or not it returns a value (for a procedure)
  - the data type of the return value (except for a conversion operator)
  - the names of the parameters or type parameters
  - the constraints on the type parameters (for a generic procedure)
  - parameter modifier keywords (such as `ByRef` or `Optional`)
  - property or procedure modifier keywords (such as `Public` or `Shared`)
- **Optional Modifier.** You do not have to use the `Overloads` modifier when you are defining multiple overloaded properties or procedures in the same class. However, if you use `Overloads` in one of the declarations, you must use it in all of them.

- **Shadowing and Overloading.** `Overloads` can also be used to shadow an existing member, or set of overloaded members, in a base class. When you use `Overloads` in this way, you declare the property or method with the same name and the same parameter list as the base class member, and you do not supply the `Shadows` keyword.

If you use `Overrides`, the compiler implicitly adds `Overloads` so that your library APIs work with C# more easily.

The `Overloads` modifier can be used in these contexts:

- Function Statement
- Operator Statement
- Property Statement
- Sub Statement

See also

- Shadows
- Procedure Overloading
- Generic Types in Visual Basic
- Operator Procedures
- How to: Define a Conversion Operator

Overridable (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that a property or procedure can be overridden by an identically named property or procedure in a derived class.

Remarks

The `Overridable` modifier allows a property or method in a class to be overridden in a derived class. The NotOverridable modifier prevents a property or method from being overridden in a derived class. For more information, see Inheritance Basics.

If the `Overridable` or `NotOverridable` modifier is not specified, the default setting depends on whether the property or method overrides a base class property or method. If the property or method overrides a base class property or method, the default setting is `Overridable`; otherwise, it is `NotOverridable`.

You can shadow or override to redefine an inherited element, but there are significant differences between the two approaches. For more information, see Shadowing in Visual Basic.

An element that can be overridden is sometimes referred to as a *virtual* element. If it can be overridden, but does not have to be, it is sometimes also called a *concrete* element.

You can use `Overridable` only in a property or procedure declaration statement.

Combined Modifiers

You cannot specify `Overridable` or `NotOverridable` for a `Private` method.

You cannot specify `Overridable` together with `MustOverride`, `NotOverridable`, or `Shared` in the same declaration.

Because an overriding element is implicitly overridable, you cannot combine `Overridable` with `Overrides`.

Usage

The `Overridable` modifier can be used in these contexts:

Function Statement

Property Statement

Sub Statement

See also

- Modifiers
- Inheritance Basics
- MustOverride
- NotOverridable
- Overrides
- Keywords
- Shadowing in Visual Basic

Overrides (Visual Basic)

4/26/2019 • 2 minutes to read • Edit Online

Specifies that a property or procedure overrides an identically named property or procedure inherited from a base class.

Rules

- **Declaration Context.** You can use `Overrides` only in a property or procedure declaration statement.
- **Combined Modifiers.** You cannot specify `Overrides` together with `Shadows` or `Shared` in the same declaration. Because an overriding element is implicitly overridable, you cannot combine `Overridable` with `Overrides`.
- **Matching Signatures.** The signature of this declaration must exactly match the *signature* of the property or procedure that it overrides. This means the parameter lists must have the same number of parameters, in the same order, with the same data types.

  In addition to the signature, the overriding declaration must also exactly match the following:

  - The access level
  - The return type, if any
- **Generic Signatures.** For a generic procedure, the signature includes the number of type parameters. Therefore, the overriding declaration must match the base class version in that respect as well.
- **Additional Matching.** In addition to matching the signature of the base class version, this declaration must also match it in the following respects:

  - Access-level modifier (such as Public)
  - Passing mechanism of each parameter (ByVal or ByRef)
  - Constraint lists on each type parameter of a generic procedure
- **Shadowing and Overriding.** Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see Shadowing in Visual Basic.

If you use `Overrides`, the compiler implicitly adds `Overloads` so that your library APIs work with C# more easily.

The `Overrides` modifier can be used in these contexts:

- Function Statement
- Property Statement
- Sub Statement

See also

- MustOverride
- NotOverridable

- Overridable
- Keywords
- Shadowing in Visual Basic
- Generic Types in Visual Basic
- Type List

ParamArray (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies that a procedure parameter takes an optional array of elements of the specified type. `ParamArray` can be used only on the last parameter of a parameter list.

Remarks

`ParamArray` allows you to pass an arbitrary number of arguments to the procedure. A `ParamArray` parameter is always declared using ByVal.

You can supply one or more arguments to a `ParamArray` parameter by passing an array of the appropriate data type, a comma-separated list of values, or nothing at all. For details, see "Calling a ParamArray" in Parameter Arrays.

> **IMPORTANT**
> Whenever you deal with an array which can be indefinitely large, there is a risk of overrunning some internal capacity of your application. If you accept a parameter array from the calling code, you should test its length and take appropriate steps if it is too large for your application.

The `ParamArray` modifier can be used in these contexts:

Declare Statement

Function Statement

Property Statement

Sub Statement

See also

- Keywords
- Parameter Arrays

Partial (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Indicates that a type declaration is a partial definition of the type.

You can divide the definition of a type among several declarations by using the `Partial` keyword. You can use as many partial declarations as you want, in as many different source files as you want. However, all the declarations must be in the same assembly and the same namespace.

> **NOTE**
>
> Visual Basic supports *partial methods*, which are typically implemented in partial classes. For more information, see Partial Methods and Sub Statement.

Syntax

```
[ <attrlist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ] _
Partial { Class | Structure | Interface | Module } name [ (Of typelist) ]
    [ Inherits classname ]
    [ Implements interfacenames ]
    [ variabledeclarations ]
    [ proceduredeclarations ]
{ End Class | End Structure }
```

Parts

TERM	DEFINITION
`attrlist`	Optional. List of attributes that apply to this type. You must enclose the Attribute List in angle brackets (`< >`).
`accessmodifier`	Optional. Specifies what code can access this type. See Access levels in Visual Basic.
`Shadows`	Optional. See Shadows.
`MustInherit`	Optional. See MustInherit.
`NotInheritable`	Optional. See NotInheritable.
`name`	Required. Name of this type. Must match the name defined in all other partial declarations of the same type.
`Of`	Optional. Specifies that this is a generic type. See Generic Types in Visual Basic.
`typelist`	Required if you use Of. See Type List.
`Inherits`	Optional. See Inherits Statement.

TERM	DEFINITION
`classname`	Required if you use `Inherits`. The name of the class or interface from which this class derives.
`Implements`	Optional. See Implements Statement.
`interfacenames`	Required if you use `Implements`. The names of the interfaces this type implements.
`variabledeclarations`	Optional. Statements which declare additional variables and events for the type.
`proceduredeclarations`	Optional. Statements which declare and define additional procedures for the type.
`End Class` or `End Structure`	Ends this partial `Class` or `Structure` definition.

Remarks

Visual Basic uses partial-class definitions to separate generated code from user-authored code in separate source files. For example, the **Windows Form Designer** defines partial classes for controls such as Form. You should not modify the generated code in these controls.

All the rules for class, structure, interface, and module creation, such as those for modifier usage and inheritance, apply when creating a partial type.

Best Practices

- Under normal circumstances, you should not split the development of a single type across two or more declarations. Therefore, in most cases you do not need the `Partial` keyword.
- For readability, every partial declaration of a type should include the `Partial` keyword. The compiler allows at most one partial declaration to omit the keyword; if two or more omit it the compiler signals an error.

Behavior

- **Union of Declarations.** The compiler treats the type as the union of all its partial declarations. Every modifier from every partial definition applies to the entire type, and every member from every partial definition is available to the entire type.
- **Type Promotion Not Allowed For Partial Types in Modules.** If a partial definition is inside a module, type promotion of that type is automatically defeated. In such a case, a set of partial definitions can cause unexpected results and even compiler errors. For more information, see Type Promotion.

  The compiler merges partial definitions only when their fully qualified paths are identical.

The `Partial` keyword can be used in these contexts:

Class Statement

Structure Statement

Example

The following example splits the definition of class `sampleClass` into two declarations, each of which defines a different `Sub` procedure.

```
Partial Public Class sampleClass
    Public Sub sub1()
    End Sub
End Class
Partial Public Class sampleClass
    Public Sub sub2()
    End Sub
End Class
```

The two partial definitions in the preceding example could be in the same source file or in two different source files.

See also

- Class Statement
- Structure Statement
- Type Promotion
- Shadows
- Generic Types in Visual Basic
- Partial Methods

Private (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that one or more declared programming elements are accessible only from within their declaration context, including from within any contained types.

Remarks

If a programming element represents proprietary functionality, or contains confidential data, you usually want to limit access to it as strictly as possible. You achieve the maximum limitation by allowing only the module, class, or structure that defines it to access it. To limit access to an element in this way, you can declare it with `Private`.

> **NOTE**
>
> You can also use the Private Protected access modifier, which makes a member accessible from within that class and from derived classes located in its containing assembly.

Rules

- **Declaration Context.** You can use `Private` only at module level. This means the declaration context for a `Private` element must be a module, class, or structure, and cannot be a source file, namespace, interface, or procedure.

Behavior

- **Access Level.** All code within a declaration context can access its `Private` elements. This includes code within a contained type, such as a nested class or an assignment expression in an enumeration. No code outside of the declaration context can access its `Private` elements.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see Access levels in Visual Basic.

The `Private` modifier can be used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Property Statement

Structure Statement

Sub Statement

See also

- Public
- Protected
- Friend
- Private Protected
- Protected Friend Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

Private Protected (Visual Basic)

3/5/2019 • 2 minutes to read • Edit Online

The `Private Protected` keyword combination is a member access modifier. A `Private Protected` member is accessible by all members in its containing class, as well as by types derived from the containing class, but only if they are found in its containing assembly.

You can specify `Private Protected` only on members of classes; you cannot apply `Private Protected` to members of a structure because structures cannot be inherited.

The `Private Protected` access modifier is supported by Visual Basic 15.5 and later. To use it, you can add the following element to your Visual Basic project (*.vbproj) file. As long as Visual Basic 15.5 or later is installed on your system, it lets you take advantage of all the language features supported by the latest version of the Visual Basic compiler:

```
<PropertyGroup>
   <LangVersion>latest</LangVersion>
</PropertyGroup>
```

For more information see setting the Visual Basic language version.

> **NOTE**
> In Visual Studio, selecting F1 help on `private protected` provides help for either private or protected. The IDE picks the single token under the cursor rather than the compound word.

Rules

- **Declaration Context.** You can use `Private Protected` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

Behavior

- **Access Level.** All code in a class can access its elements. Code in any class that derives from a base class and is contained in the same assembly can access all the `Private Protected` elements of the base class. However, code in any class that derives from a base class and is contained in a different assembly can't access the base class `Private Protected` elements.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see Access levels in Visual Basic.

The `Private Protected` modifier can be used in these contexts:

- Class Statement of a nested class
- Const Statement
- Declare Statement
- Delegate Statement of a delegate nested in a class

- Dim Statement
- Enum Statement of an enumeration nested in a class
- Event Statement
- Function Statement
- Interface Statement of an interface nested in a class
- Property Statement
- Structure Statement of a structure nested in a class
- Sub Statement

See also

- Public
- Protected
- Friend
- Private
- Protected Friend
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

Protected (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

A member access modifier that specifies that one or more declared programming elements are accessible only from within their own class or from a derived class.

Remarks

Sometimes a programming element declared in a class contains sensitive data or restricted code, and you want to limit access to the element. However, if the class is inheritable and you expect a hierarchy of derived classes, it might be necessary for these derived classes to access the data or code. In such a case, you want the element to be accessible both from the base class and from all derived classes. To limit access to an element in this manner, you can declare it with `Protected`.

> **NOTE**
>
> The `Protected` access modifier can be combined with two other modifiers:
>
> - The Protected Friend modifier makes a class member accessible from within that class, from derived classes, and from the same assembly in which the class is defined.
> - The Private Protected modifier makes a class member accessible by derived types, but only within its containing assembly.

Rules

- **Declaration Context.** You can use `Protected` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

Behavior

- **Access Level.** All code in a class can access its elements. Code in any class that derives from a base class can access all the `Protected` elements of the base class. This is true for all generations of derivation. This means that a class can access `Protected` elements of the base class of the base class, and so on.

  Protected access is not a superset or subset of friend access.

- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see Access levels in Visual Basic.

The `Protected` modifier can be used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Property Statement

Structure Statement

Sub Statement

See also

- Public
- Friend
- Private
- Private Protected
- Protected Friend
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

Protected Friend (Visual Basic)

4/8/2019 • 2 minutes to read • Edit Online

The `Protected Friend` keyword combination is a member access modifier. It confers both Friend access and Protected access on the declared elements, so they are accessible from anywhere in the same assembly, from their own class, and from derived classes. You can specify `Protected Friend` only on members of classes; you cannot apply `Protected Friend` to members of a structure because structures cannot be inherited.

> **NOTE**
> In Visual Studio, selecting F1 help on `protected friend` provides help for either protected or friend. The IDE picks the single token under the cursor rather than the compound word.

Rules

- **Declaration Context.** You can use `Protected Friend` only at the class level. This means the declaration context for a `Protected` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

See also

- Public
- Protected
- Friend
- Private
- Private Protected
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

Public (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that one or more declared programming elements have no access restrictions.

Remarks

If you are publishing a component or set of components, such as a class library, you usually want the programming elements to be accessible by any code that interoperates with your assembly. To confer such unlimited access on an element, you can declare it with `Public`.

Public access is the normal level for a programming element when you do not need to limit access to it. Note that the access level of an element declared within an interface, module, class, or structure defaults to `Public` if you do not declare it otherwise.

Rules

- **Declaration Context.** You can use `Public` only at module, interface, or namespace level. This means the declaration context for a `Public` element must be a source file, namespace, interface, module, class, or structure, and cannot be a procedure.

Behavior

- **Access Level.** All code that can access a module, class, or structure can access its `Public` elements.
- **Default Access.** Local variables inside a procedure default to public access, and you cannot use any access modifiers on them.
- **Access Modifiers.** The keywords that specify access level are called *access modifiers*. For a comparison of the access modifiers, see Access levels in Visual Basic.

The `Public` modifier can be used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Module Statement

Operator Statement

Property Statement

Structure Statement

Sub Statement

See also

- Protected
- Friend
- Private
- Private Protected
- Protected Friend
- Access levels in Visual Basic
- Procedures
- Structures
- Objects and Classes

ReadOnly (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies that a variable or property can be read but not written.

Remarks

Rules

- **Declaration Context.** You can use `ReadOnly` only at module level. This means the declaration context for a `ReadOnly` element must be a class, structure, or module, and cannot be a source file, namespace, or procedure.
- **Combined Modifiers.** You cannot specify `ReadOnly` together with `Static` in the same declaration.
- **Assigning a Value.** Code consuming a `ReadOnly` property cannot set its value. But code that has access to the underlying storage can assign or change the value at any time.

  You can assign a value to a `ReadOnly` variable only in its declaration or in the constructor of a class or structure in which it is defined.

When to Use a ReadOnly Variable

There are situations in which you cannot use a Const Statement to declare and assign a constant value. For example, the `Const` statement might not accept the data type you want to assign, or you might not be able to compute the value at compile time with a constant expression. You might not even know the value at compile time. In these cases, you can use a `ReadOnly` variable to hold a constant value.

> **IMPORTANT**
> If the data type of the variable is a reference type, such as an array or a class instance, its members can be changed even if the variable itself is `ReadOnly`. The following example illustrates this.

```
ReadOnly characterArray() As Char = {"x"c, "y"c, "z"c}

Sub changeArrayElement()

characterArray(1) = "M"c

End Sub
```

When initialized, the array pointed to by `characterArray()` holds "x", "y", and "z". Because the variable `characterArray` is `ReadOnly`, you cannot change its value once it is initialized; that is, you cannot assign a new array to it. However, you can change the values of one or more of the array members. Following a call to the procedure `changeArrayElement`, the array pointed to by `characterArray()` holds "x", "M", and "z".

Note that this is similar to declaring a procedure parameter to be ByVal, which prevents the procedure from changing the calling argument itself but allows it to change its members.

Example

The following example defines a `ReadOnly` property for the date on which an employee was hired. The class stores the property value internally as a `Private` variable, and only code inside the class can change that value. However, the property is `Public`, and any code that can access the class can read the property.

```
Class employee
    ' Only code inside class employee can change the value of hireDateValue.
    Private hireDateValue As Date
    ' Any code that can access class employee can read property dateHired.
    Public ReadOnly Property dateHired() As Date
        Get
            Return hireDateValue
        End Get
    End Property
End Class
```

The `ReadOnly` modifier can be used in these contexts:

Dim Statement

Property Statement

See also

- WriteOnly
- Keywords

Shadows (Visual Basic)

7/9/2019 • 2 minutes to read • Edit Online

Specifies that a declared programming element redeclares and hides an identically named element, or set of overloaded elements, in a base class.

Remarks

The main purpose of shadowing (which is also known as *hiding by name*) is to preserve the definition of your class members. The base class might undergo a change that creates an element with the same name as one you have already defined. If this happens, the `Shadows` modifier forces references through your class to be resolved to the member you defined, instead of to the new base class element.

Both shadowing and overriding redefine an inherited element, but there are significant differences between the two approaches. For more information, see Shadowing in Visual Basic.

Rules

- **Declaration Context.** You can use `Shadows` only at class level. This means the declaration context for a `Shadows` element must be a class, and cannot be a source file, namespace, interface, module, structure, or procedure.

  You can declare only one shadowing element in a single declaration statement.

- **Combined Modifiers.** You cannot specify `Shadows` together with `Overloads`, `Overrides`, or `Static` in the same declaration.

- **Element Types.** You can shadow any kind of declared element with any other kind. If you shadow a property or procedure with another property or procedure, the parameters and the return type do not have to match those in the base class property or procedure.

- **Accessing.** The shadowed element in the base class is normally unavailable from within the derived class that shadows it. However, the following considerations apply.

  - If the shadowing element is not accessible from the code referring to it, the reference is resolved to the shadowed element. For example, if a `Private` element shadows a base class element, code that does not have permission to access the `Private` element accesses the base class element instead.

  - If you shadow an element, you can still access the shadowed element through an object declared with the type of the base class. You can also access it through `MyBase`.

The `Shadows` modifier can be used in these contexts:

- Class Statement
- Const Statement
- Declare Statement
- Delegate Statement
- Dim Statement
- Enum Statement

- Event Statement
- Function Statement
- Interface Statement
- Property Statement
- Structure Statement
- Sub Statement

See also

- Shared
- Static
- Private
- Me, My, MyBase, and MyClass
- Inheritance Basics
- MustOverride
- NotOverridable
- Overloads
- Overridable
- Overrides
- Shadowing in Visual Basic

Shared (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Specifies that one or more declared programming elements are associated with a class or structure at large, and not with a specific instance of the class or structure.

Remarks

When to Use Shared

Sharing a member of a class or structure makes it available to every instance, rather than *nonshared*, where each instance keeps its own copy. This is useful, for example, if the value of a variable applies to the entire application. If you declare that variable to be `Shared`, then all instances access the same storage location, and if one instance changes the variable's value, all instances access the updated value.

Sharing does not alter the access level of a member. For example, a class member can be shared and private (accessible only from within the class), or nonshared and public. For more information, see Access levels in Visual Basic.

Rules

- **Declaration Context.** You can use `Shared` only at module level. This means the declaration context for a `Shared` element must be a class or structure, and cannot be a source file, namespace, or procedure.
- **Combined Modifiers.** You cannot specify `Shared` together with Overrides, Overridable, NotOverridable, MustOverride, or Static in the same declaration.
- **Accessing.** You access a shared element by qualifying it with its class or structure name, not with the variable name of a specific instance of its class or structure. You do not even have to create an instance of a class or structure to access its shared members.

  The following example calls the shared procedure IsNaN exposed by the Double structure.

  ```
  If Double.IsNaN(result) Then MsgBox("Result is mathematically undefined.")
  ```

- **Implicit Sharing.** You cannot use the `Shared` modifier in a Const Statement, but constants are implicitly shared. Similarly, you cannot declare a member of a module or an interface to be `Shared`, but they are implicitly shared.

Behavior

- **Storage.** A shared variable or event is stored in memory only once, no matter how many or few instances you create of its class or structure. Similarly, a shared procedure or property holds only one set of local variables.
- **Accessing through an Instance Variable.** It is possible to access a shared element by qualifying it with the name of a variable that contains a specific instance of its class or structure. Although this usually works as expected, the compiler generates a warning message and makes the access through the class or structure name instead of the variable.
- **Accessing through an Instance Expression.** If you access a shared element through an expression that returns an instance of its class or structure, the compiler makes the access through the class or

structure name instead of evaluating the expression. This produces unexpected results if you intended the expression to perform other actions as well as returning the instance. The following example illustrates this.

```
Sub main()
    shareTotal.total = 10
    ' The preceding line is the preferred way to access total.
    Dim instanceVar As New shareTotal
    instanceVar.total += 100
    ' The preceding line generates a compiler warning message and
    ' accesses total through class shareTotal instead of through
    ' the variable instanceVar. This works as expected and adds
    ' 100 to total.
    returnClass().total += 1000
    ' The preceding line generates a compiler warning message and
    ' accesses total through class shareTotal instead of calling
    ' returnClass(). This adds 1000 to total but does not work as
    ' expected, because the MsgBox in returnClass() does not run.
    MsgBox("Value of total is " & CStr(shareTotal.total))
End Sub
Public Function returnClass() As shareTotal
    MsgBox("Function returnClass() called")
    Return New shareTotal
End Function
Public Class shareTotal
    Public Shared total As Integer
End Class
```

In the preceding example, the compiler generates a warning message both times the code accesses the shared variable `total` through an instance. In each case it makes the access directly through the class `shareTotal` and does not make use of any instance. In the case of the intended call to the procedure `returnClass`, this means it does not even generate a call to `returnClass`, so the additional action of displaying "Function returnClass() called" is not performed.

The `Shared` modifier can be used in these contexts:

Dim Statement

Event Statement

Function Statement

Operator Statement

Property Statement

Sub Statement

See also

- Shadows
- Static
- Lifetime in Visual Basic
- Procedures
- Structures
- Objects and Classes

Static (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies that one or more declared local variables are to continue to exist and retain their latest values after termination of the procedure in which they are declared.

Remarks

Normally, a local variable in a procedure ceases to exist as soon as the procedure stops. A static variable continues to exist and retains its most recent value. The next time your code calls the procedure, the variable is not reinitialized, and it still holds the latest value that you assigned to it. A static variable continues to exist for the lifetime of the class or module that it is defined in.

Rules

- **Declaration Context.** You can use `Static` only on local variables. This means the declaration context for a `Static` variable must be a procedure or a block in a procedure, and it cannot be a source file, namespace, class, structure, or module.

  You cannot use `Static` inside a structure procedure.

- The data types of `Static` local variables cannot be inferred. For more information, see Local Type Inference.

- **Combined Modifiers.** You cannot specify `Static` together with `ReadOnly`, `Shadows`, or `Shared` in the same declaration.

Behavior

When you declare a static variable in a `Shared` procedure, only one copy of the static variable is available for the whole application. You call a `Shared` procedure by using the class name, not a variable that points to an instance of the class.

When you declare a static variable in a procedure that isn't `Shared`, only one copy of the variable is available for each instance of the class. You call a non-shared procedure by using a variable that points to a specific instance of the class.

Example

The following example demonstrates the use of `Static`.

```
Function updateSales(ByVal thisSale As Decimal) As Decimal
    Static totalSales As Decimal = 0
    totalSales += thisSale
    Return totalSales
End Function
```

The `Static` variable `totalSales` is initialized to 0 only one time. Each time that you enter `updateSales`, `totalSales` still has the most recent value that you calculated for it.

The `Static` modifier can be used in this context:

Dim Statement

See also

- Shadows
- Shared
- Lifetime in Visual Basic
- Variable Declaration
- Structures
- Local Type Inference
- Objects and Classes

Unicode (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that Visual Basic should marshal all strings to Unicode values regardless of the name of the external procedure being declared.

When you call a procedure defined outside your project, the Visual Basic compiler does not have access to the information it must have in order to call the procedure correctly. This information includes where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The Declare Statement creates a reference to an external procedure and supplies this necessary information.

The `charsetmodifier` part in the `Declare` statement supplies the character set information to marshal strings during a call to the external procedure. It also affects how Visual Basic searches the external file for the external procedure name. The `Unicode` modifier specifies that Visual Basic should marshal all strings to Unicode values and should look up the procedure without modifying its name during the search.

If no character set modifier is specified, `Ansi` is the default.

Remarks

The `Unicode` modifier can be used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See also

- Ansi
- Auto
- Keywords

Widening (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that a conversion operator (`CType`) converts a class or structure to a type that can hold all possible values of the original class or structure.

Converting with the Widening Keyword

The conversion procedure must specify `Public Shared` in addition to `Widening` .

Widening conversions always succeed at run time and never incur data loss. Examples are `Single` to `Double` , `Char` to `String` , and a derived type to its base type. This last conversion is widening because the derived type contains all the members of the base type and thus is an instance of the base type.

The consuming code does not have to use `CType` for widening conversions, even if `Option Strict` is `On` .

The `Widening` keyword can be used in this context:

Operator Statement

For example definitions of widening and narrowing conversion operators, see How to: Define a Conversion Operator.

See also

- Operator Statement
- Narrowing
- Widening and Narrowing Conversions
- How to: Define an Operator
- CType Function
- Option Strict Statement
- How to: Define a Conversion Operator

WithEvents (Visual Basic)

6/20/2019 • 2 minutes to read • Edit Online

Specifies that one or more declared member variables refer to an instance of a class that can raise events.

Remarks

When a variable is defined using `WithEvents`, you can declaratively specify that a method handles the variable's events using the `Handles` keyword.

You can use `WithEvents` only at class or module level. This means the declaration context for a `WithEvents` variable must be a class or module and cannot be a source file, namespace, structure, or procedure.

You cannot use `WithEvents` on a structure member.

You can declare only individual variables—not arrays—with `WithEvents`.

Rules

- **Element Types.** You must declare `WithEvents` variables to be object variables so that they can accept class instances. However, you cannot declare them as `Object`. You must declare them as the specific class that can raise the events.

The `WithEvents` modifier can be used in this context: Dim Statement

Example

```
Dim WithEvents app As Application
```

See also

- Handles
- Keywords
- Events

WriteOnly (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies that a property can be written but not read.

Remarks

Rules

Declaration Context. You can use `WriteOnly` only at module level. This means the declaration context for a `WriteOnly` property must be a class, structure, or module, and cannot be a source file, namespace, or procedure.

You can declare a property as `WriteOnly`, but not a variable.

When to Use WriteOnly

Sometimes you want the consuming code to be able to set a value but not discover what it is. For example, sensitive data, such as a social registration number or a password, needs to be protected from access by any component that did not set it. In these cases, you can use a `WriteOnly` property to set the value.

> **IMPORTANT**
> When you define and use a `WriteOnly` property, consider the following additional protective measures:

- **Overriding.** If the property is a member of a class, allow it to default to NotOverridable, and do not declare it `Overridable` or `MustOverride`. This prevents a derived class from making undesired access through an override.
- **Access Level.** If you hold the property's sensitive data in one or more variables, declare them Private so that no other code can access them.
- **Encryption.** Store all sensitive data in encrypted form rather than in plain text. If malicious code somehow gains access to that area of memory, it is more difficult to make use of the data. Encryption is also useful if it is necessary to serialize the sensitive data.
- **Resetting.** When the class, structure, or module defining the property is being terminated, reset the sensitive data to default values or to other meaningless values. This gives extra protection when that area of memory is freed for general access.
- **Persistence.** Do not persist any sensitive data, for example on disk, if you can avoid it. Also, do not write any sensitive data to the Clipboard.

The `WriteOnly` modifier can be used in this context:

Property Statement

See also

- ReadOnly
- Private
- Keywords

Modules (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic provides several modules that enable you to simplify common tasks in your code, including manipulating strings, performing mathematical calculations, getting system information, performing file and directory operations, and so on. The following table lists the modules provided by Visual Basic.

Constants	Contains miscellaneous constants. These constants can be used anywhere in your code.
ControlChars	Contains constant control characters for printing and displaying text.
Conversion	Contains members that convert decimal numbers to other bases, numbers to strings, strings to numbers, and one data type to another.
DateAndTime	Contains members that get the current date or time, perform date calculations, return a date or time, set the date or time, or time the duration of a process.
ErrObject	Contains information about run-time errors and methods to raise or clear an error.
FileSystem	Contains members that perform file, directory or folder, and system operations.
Financial	Contains procedures that are used to perform financial calculations.
Globals	Contains information about the current scripting engine version.
Information	Contains the members that return, test for, or verify information such as array size, type names, and so on.
Interaction	Contains members interact with objects, applications, and systems.
Strings	Contains members that perform string operations such as reformatting strings, searching a string, getting the length of a string, and so on.
VBMath	Contains members perform mathematical operations.

See also

- Visual Basic Language Reference
- Visual Basic

Nothing (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Represents the default value of any data type. For reference types, the default value is the `null` reference. For value types, the default value depends on whether the value type is nullable.

> **NOTE**
> For non-nullable value types, `Nothing` in Visual Basic differs from `null` in C#. In Visual Basic, if you set a variable of a non-nullable value type to `Nothing`, the variable is set to the default value for its declared type. In C#, if you assign a variable of a non-nullable value type to `null`, a compile-time error occurs.

Remarks

`Nothing` represents the default value of a data type. The default value depends on whether the variable is of a value type or of a reference type.

A variable of a *value type* directly contains its value. Value types include all numeric data types, `Boolean`, `Char`, `Date`, all structures, and all enumerations. A variable of a *reference type* stores a reference to an instance of the object in memory. Reference types include classes, arrays, delegates, and strings. For more information, see Value Types and Reference Types.

If a variable is of a value type, the behavior of `Nothing` depends on whether the variable is of a nullable data type. To represent a nullable value type, add a `?` modifier to the type name. Assigning `Nothing` to a nullable variable sets the value to `null`. For more information and examples, see Nullable Value Types.

If a variable is of a value type that is not nullable, assigning `Nothing` to it sets it to the default value for its declared type. If that type contains variable members, they are all set to their default values. The following example illustrates this for scalar types.

```
Module Module1

    Sub Main()
        Dim ts As TestStruct
        Dim i As Integer
        Dim b As Boolean

        ' The following statement sets ts.Name to Nothing and ts.Number to 0.
        ts = Nothing

        ' The following statements set i to 0 and b to False.
        i = Nothing
        b = Nothing

        Console.WriteLine("ts.Name: " & ts.Name)
        Console.WriteLine("ts.Number: " & ts.Number)
        Console.WriteLine("i: " & i)
        Console.WriteLine("b: " & b)

        Console.ReadKey()
    End Sub

    Public Structure TestStruct
        Public Name As String
        Public Number As Integer
    End Structure
End Module
```

If a variable is of a reference type, assigning `Nothing` to the variable sets it to a `null` reference of the variable's type. A variable that is set to a `null` reference is not associated with any object. The following example demonstrates this.

```
Module Module1

    Sub Main()

        Dim testObject As Object
        ' The following statement sets testObject so that it does not refer to
        ' any instance.
        testObject = Nothing

        Dim tc As New TestClass
        tc = Nothing
        ' The fields of tc cannot be accessed. The following statement causes
        ' a NullReferenceException at run time. (Compare to the assignment of
        ' Nothing to structure ts in the previous example.)
        'Console.WriteLine(tc.Field1)

    End Sub

    Class TestClass
        Public Field1 As Integer
        ' . . .
    End Class
End Module
```

When checking whether a reference (or nullable value type) variable is `null`, do not use `= Nothing` or `<> Nothing`. Always use `Is Nothing` or `IsNot Nothing`.

For strings in Visual Basic, the empty string equals `Nothing`. Therefore, `"" = Nothing` is true.

The following example shows comparisons that use the `Is` and `IsNot` operators.

```
Module Module1
    Sub Main()

        Dim testObject As Object
        testObject = Nothing
        Console.WriteLine(testObject Is Nothing)
        ' Output: True

        Dim tc As New TestClass
        tc = Nothing
        Console.WriteLine(tc IsNot Nothing)
        ' Output: False

        ' Declare a nullable value type.
        Dim n? As Integer
        Console.WriteLine(n Is Nothing)
        ' Output: True

        n = 4
        Console.WriteLine(n Is Nothing)
        ' Output: False

        n = Nothing
        Console.WriteLine(n IsNot Nothing)
        ' Output: False

        Console.ReadKey()
    End Sub

    Class TestClass
        Public Field1 As Integer
        Private field2 As Boolean
    End Class
End Module
```

If you declare a variable without using an `As` clause and set it to `Nothing`, the variable has a type of `Object`. An example of this is `Dim something = Nothing`. A compile-time error occurs in this case when `Option Strict` is on and `Option Infer` is off.

When you assign `Nothing` to an object variable, it no longer refers to any object instance. If the variable had previously referred to an instance, setting it to `Nothing` does not terminate the instance itself. The instance is terminated, and the memory and system resources associated with it are released, only after the garbage collector (GC) detects that there are no active references remaining.

`Nothing` differs from the DBNull object, which represents an uninitialized variant or a nonexistent database column.

See also

- Dim Statement
- Object Lifetime: How Objects Are Created and Destroyed
- Lifetime in Visual Basic
- Is Operator
- IsNot Operator
- Nullable Value Types

Objects (Visual Basic)

5/18/2019 • 2 minutes to read • Edit Online

This topic provides links to other topics that document the Visual Basic run-time objects and contain tables of their member procedures, properties, and events.

Visual Basic Run-time Objects

Collection	Provides a convenient way to see a related group of items as a single object.
Err	Contains information about run-time errors.
The `My.Application` object consists of the following classes: ApplicationBase provides members that are available in all projects. WindowsFormsApplicationBase provides members available in Windows Forms applications. ConsoleApplicationBase provides members available in console applications.	Provides data that is associated only with the current application or DLL. No system-level information can be altered with `My.Application`. Some members are available only for Windows Forms or console applications.
`My.Application.Info` (Info)	Provides properties for getting the information about an application, such as the version number, description, loaded assemblies, and so on.
`My.Application.Log` (Log)	Provides a property and methods to write event and exception information to the application's log listeners.
`My.Computer` (Computer)	Provides properties for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.
`My.Computer.Audio` (Audio)	Provides methods for playing sounds.
`My.Computer.Clipboard` (Clipboard)	Provides methods for manipulating the Clipboard.
`My.Computer.Clock` (Clock)	Provides properties for accessing the current local time and Universal Coordinated Time (equivalent to Greenwich Mean Time) from the system clock.
`My.Computer.FileSystem` (FileSystem)	Provides properties and methods for working with drives, files, and directories.
`My.Computer.FileSystem.SpecialDirectories` (SpecialDirectories)	Provides properties for accessing commonly referenced directories.

`My.Computer.Info` (ComputerInfo)	Provides properties for getting information about the computer's memory, loaded assemblies, name, and operating system.
`My.Computer.Keyboard` (Keyboard)	Provides properties for accessing the current state of the keyboard, such as what keys are currently pressed, and provides a method to send keystrokes to the active window.
`My.Computer.Mouse` (Mouse)	Provides properties for getting information about the format and configuration of the mouse that is installed on the local computer.
`My.Computer.Network` (Network)	Provides a property, an event, and methods for interacting with the network to which the computer is connected.
`My.Computer.Ports` (Ports)	Provides a property and a method for accessing the computer's serial ports.
`My.Computer.Registry` (RegistryProxy)	Provides properties and methods for manipulating the registry.
My.Forms Object	Provides properties for accessing an instance of each Windows Form declared in the current project.
`My.Log` (AspLog)	Provides a property and methods for writing event and exception information to the application's log listeners for Web applications.
My.Request Object	Gets the HttpRequest object for the requested page. The `My.Request` object contains information about the current HTTP request. The `My.Request` object is available only for ASP.NET applications.
My.Resources Object	Provides properties and classes for accessing an application's resources.
My.Response Object	Gets the HttpResponse object that is associated with the Page. This object allows you to send HTTP response data to a client and contains information about that response. The `My.Response` object is available only for ASP.NET applications.
My.Settings Object	Provides properties and methods for accessing an application's settings.
`My.User` (User)	Provides access to information about the current user.
My.WebServices Object	Provides properties for creating and accessing a single instance of each Web service that is referenced by the current project.

TextFieldParser	Provides methods and properties for parsing structured text files.

See also

- Visual Basic Language Reference
- Visual Basic

My.Application Object

4/28/2019 • 2 minutes to read • Edit Online

Provides properties, methods, and events related to the current application.

Remarks

For information about the methods and properties of the `My.Application` object, see the following resources:

- ApplicationBase for members that are available in all projects.
- WindowsFormsApplicationBase for members that are available in Windows Forms applications.
- ConsoleApplicationBase for members that are available in console applications.

Requirements

Namespace: Microsoft.VisualBasic.ApplicationServices

Class: WindowsFormsApplicationBase (the base class ConsoleApplicationBase provides members available in console applications, and its base class ApplicationBase provides the members that are available in all projects)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Application.Info Object
- My.Application.Log Object

My.Application.Info Object

8/22/2019 • 2 minutes to read • Edit Online

Provides properties for getting the information about the application, such as the version number, description, loaded assemblies, and so on.

Remarks

For information about the methods and properties of the `My.Application.Info` object, see AssemblyInfo.

> **NOTE**
> You can use properties of the System.Diagnostics.FileVersionInfo class to obtain information about a file on disk.

Requirements

Namespace: Microsoft.VisualBasic.ApplicationServices

Class: AssemblyInfo

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Application Object

My.Application.Log Object

8/22/2019 • 2 minutes to read • Edit Online

Provides a property and methods to write event and exception information to the application's log listeners.

Remarks

For information about the methods and properties of the `My.Application.Log` object, see Log.

For more information, see Logging Information from the Application.

> **NOTE**
>
> You can also use classes in the .NET Framework to log information from your application. For more information, see Tracing and Instrumenting Applications.

Requirements

Namespace: Microsoft.VisualBasic.Logging

Class: Log

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Application Object

My.Computer Object

4/2/2019 • 2 minutes to read • Edit Online

Provides properties for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.

Remarks

For information about the methods and properties of the `My.Computer` object, see Computer. The base class ServerComputer provides the members that are available in all projects.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Computer (the base class ServerComputer provides the members that are available in all projects).

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer.Audio Object
- My.Computer.Clipboard Object
- My.Computer.Clock Object
- My.Computer.FileSystem Object
- My.Computer.FileSystem.SpecialDirectories Object
- My.Computer.Info Object
- My.Computer.Keyboard Object
- My.Computer.Mouse Object
- My.Computer.Network Object
- My.Computer.Ports Object
- My.Computer.Registry Object

My.Computer.Audio Object

4/2/2019 • 2 minutes to read • Edit Online

Provides methods for playing sounds.

Remarks

For information about the methods and properties of the `My.Computer.Audio` object, see Audio.

For more information, see Playing Sounds.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Audio

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Clipboard Object

8/22/2019 • 2 minutes to read • Edit Online

Provides methods for manipulating the Clipboard.

Remarks

For information about the methods and properties of the `My.Computer.Clipboard` object, see ClipboardProxy.

For more information, see Storing Data to and Reading from the Clipboard.

> **NOTE**
>
> You can also use methods of the System.Windows.Forms.Clipboard class to manipulate the Clipboard.

Requirements

Namespace: Microsoft.VisualBasic.MyServices

Class: ClipboardProxy (provides access to Clipboard)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- Clipboard
- My.Computer Object

My.Computer.Clock Object

4/2/2019 • 2 minutes to read • Edit Online

Provides properties for accessing the current local time and Universal Coordinated Time (equivalent to Greenwich Mean Time) from the system clock.

Remarks

For information about the methods and properties of the `My.Computer.Clock` object, see Clock.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Clock

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.FileSystem Object

8/22/2019 • 2 minutes to read • Edit Online

Provides properties and methods for working with drives, files, and directories.

Remarks

For information about the methods and properties of the `My.Computer.FileSystem` object, see FileSystem.

For more information, see File Access with Visual Basic.

> **NOTE**
>
> You can also use classes in the System.IO namespace to work with drives, files, and directories.

Requirements

Namespace: Microsoft.VisualBasic.MyServices

Class: FileSystemProxy (provides access to FileSystem)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer.FileSystem.SpecialDirectories Object
- My.Computer Object

My.Computer.FileSystem.SpecialDirectories Object

6/13/2019 • 2 minutes to read • Edit Online

Provides properties for accessing commonly referenced directories.

Remarks

For information about the methods and properties of the `My.Computer.FileSystem.SpecialDirectories` object, see SpecialDirectories.

For more information, see How to: Retrieve the Contents of the My Documents Directory.

Requirements

Namespace: Microsoft.VisualBasic.MyServices

Class: SpecialDirectoriesProxy (provides access to SpecialDirectories)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer.FileSystem Object
- My.Computer Object

My.Computer.Info Object

4/2/2019 • 2 minutes to read • Edit Online

Provides properties for getting information about the computer's memory, loaded assemblies, name, and operating system.

Remarks

For information about the properties of the `My.Computer.Info` object, see ComputerInfo.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: ComputerInfo

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Keyboard Object

4/2/2019 • 2 minutes to read • Edit Online

Provides properties for accessing the current state of the keyboard, such as what keys are currently pressed, and provides a method to send keystrokes to the active window.

Remarks

For information about the methods and properties of the `My.Computer.Keyword` object, see Keyboard.

For more information, see Accessing the Keyboard.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Keyboard

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Mouse Object

4/2/2019 • 2 minutes to read • Edit Online

Provides properties for getting information about the format and configuration of the mouse installed on the local computer.

Remarks

For information about the methods and properties of the `My.Computer.Mouse` object, see Mouse.

For more information, see Accessing the Mouse.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Mouse

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Network Object

4/2/2019 • 2 minutes to read • Edit Online

Provides a property, event, and methods for interacting with the network to which the computer is connected.

Remarks

For information about the methods and properties of the `My.Computer.Network` object, see Network.

For more information, see Performing Network Operations.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Network

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Ports Object

8/22/2019 • 2 minutes to read • Edit Online

Provides a property and a method for accessing the computer's serial ports.

Remarks

For information about the methods and properties of the `My.Computer.Ports` object, see Ports.

For more information, see Accessing the Computer's Ports.

> **NOTE**
>
> You can also use properties and methods of the System.IO.Ports.SerialPort class to access the computer's serial ports.

Requirements

Namespace: Microsoft.VisualBasic.Devices

Class: Ports

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Computer.Registry Object

8/22/2019 • 2 minutes to read • Edit Online

Provides properties and methods for manipulating the registry.

Remarks

For information about the methods and properties of the `My.Computer.Registry` object, see RegistryProxy.

For more information, see Reading from and Writing to the Registry.

> **NOTE**
>
> You can also manipulate the registry by using methods of the Microsoft.Win32.Registry class.

Requirements

Namespace: Microsoft.VisualBasic.MyServices

Class: RegistryProxy (provides access to Registry)

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- My.Computer Object

My.Forms Object

8/22/2019 • 2 minutes to read • Edit Online

Provides properties for accessing an instance of each Windows form declared in the current project.

Remarks

The `My.Forms` object provides an instance of each form in the current project. The name of the property is the same as the name of the form that the property accesses.

You can access the forms provided by the `My.Forms` object by using the name of the form, without qualification. Because the property name is the same as the form's type name, this allows you to access a form as if it had a default instance. For example, `My.Forms.Form1.Show` is equivalent to `Form1.Show`.

The `My.Forms` object exposes only the forms associated with the current project. It does not provide access to forms declared in referenced DLLs. To access a form that a DLL provides, you must use the qualified name of the form, written as *DllName.FormName*.

You can use the OpenForms property to get a collection of all the application's open forms.

The object and its properties are available only for Windows applications.

Properties

Each property of the `My.Forms` object provides access to an instance of a form in the current project. The name of the property is the same as the name of the form that the property accesses, and the property type is the same as the form's type.

> **NOTE**
>
> If there is a name collision, the property name to access a form is *RootNamespace_Namespace_FormName*. For example, consider two forms named `Form1.` If one of these forms is in the root namespace `WindowsApplication1` and in the namespace `Namespace1`, you would access that form through `My.Forms.WindowsApplication1_Namespace1_Form1`.

The `My.Forms` object provides access to the instance of the application's main form that was created on startup. For all other forms, the `My.Forms` object creates a new instance of the form when it is accessed and stores it. Subsequent attempts to access that property return that instance of the form.

You can dispose of a form by assigning `Nothing` to the property for that form. The property setter calls the Close method of the form, and then assigns `Nothing` to the stored value. If you assign any value other than `Nothing` to the property, the setter throws an ArgumentException exception.

You can test whether a property of the `My.Forms` object stores an instance of the form by using the `Is` or `IsNot` operator. You can use those operators to check if the value of the property is `Nothing`.

NOTE

Typically, the `Is` or `IsNot` operator has to read the value of the property to perform the comparison. However, if the property currently stores `Nothing`, the property creates a new instance of the form and then returns that instance. However, the Visual Basic compiler treats the properties of the `My.Forms` object differently and allows the `Is` or `IsNot` operator to check the status of the property without altering its value.

Example

This example changes the title of the default `SidebarMenu` form.

```
Sub ShowSidebarMenu(ByVal newTitle As String)
    If My.Forms.SidebarMenu IsNot Nothing Then
        My.Forms.SidebarMenu.Text = newTitle
    End If
End Sub
```

For this example to work, your project must have a form named `SidebarMenu`.

This code will work only in a Windows Application project.

Requirements

Availability by Project Type

PROJECT TYPE	AVAILABLE
Windows Application	**Yes**
Class Library	No
Console Application	No
Windows Control Library	No
Web Control Library	No
Windows Service	No
Web Site	No

See also

- OpenForms
- Form
- Close
- Objects
- Is Operator
- IsNot Operator
- Accessing Application Forms

My.Log Object

6/13/2019 • 2 minutes to read • Edit Online

Provides a property and methods for writing event and exception information to the application's log listeners.

Remarks

For information about the methods and properties of the `My.Log` object, see AspLog.

The `My.Log` object is available for ASP.NET applications only. For client applications, use My.Application.Log Object.

Requirements

Namespace: Microsoft.VisualBasic.Logging

Class: AspLog

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

My.Request Object

8/17/2019 • 2 minutes to read • Edit Online

Gets the HttpRequest object for the requested page.

Remarks

The `My.Request` object contains information about the current HTTP request.

The `My.Request` object is available only for ASP.NET applications.

Example

The following example gets the header collection from the `My.Request` object and uses the `My.Response` object to write it to the ASP.NET page.

```
<script runat="server">
    Public Sub ShowHeaders()
        ' Load the header collection from the Request object.
        Dim coll As System.Collections.Specialized.NameValueCollection
        coll = My.Request.Headers

        ' Put the names of all keys into a string array.
        For Each key As String In coll.AllKeys
            My.Response.Write("Key: " & key & "<br>")

            ' Get all values under this key.
            For Each value As String In coll.GetValues(key)
                My.Response.Write("Value: " & _
                    Server.HtmlEncode(value) & "<br>")
            Next
        Next
    End Sub
</script>
```

See also

- HttpRequest
- My.Response Object

My.Response Object

8/17/2019 • 2 minutes to read • Edit Online

Gets the HttpResponse object associated with the Page. This object allows you to send HTTP response data to a client and contains information about that response.

Remarks

The `My.Response` object contains the current HttpResponse object associated with the page.

The `My.Response` object is only available for ASP.NET applications.

Example

The following example gets the header collection from the `My.Request` object and uses the `My.Response` object to write it to the ASP.NET page.

```
<script runat="server">
    Public Sub ShowHeaders()
        ' Load the header collection from the Request object.
        Dim coll As System.Collections.Specialized.NameValueCollection
        coll = My.Request.Headers

        ' Put the names of all keys into a string array.
        For Each key As String In coll.AllKeys
            My.Response.Write("Key: " & key & "<br>")

            ' Get all values under this key.
            For Each value As String In coll.GetValues(key)
                My.Response.Write("Value: " & _
                    Server.HtmlEncode(value) & "<br>")
            Next
        Next
    End Sub
</script>
```

See also

- HttpResponse
- My.Request Object

My.Resources Object

4/28/2019 • 3 minutes to read • Edit Online

Provides properties and classes for accessing the application's resources.

Remarks

The `My.Resources` object provides access to the application's resources and lets you dynamically retrieve resources for your application. For more information, see Managing Application Resources (.NET).

The `My.Resources` object exposes only global resources. It does not provide access to resource files associated with forms. You must access the form resources from the form.

You can access the application's culture-specific resource files from the `My.Resources` object. By default, the `My.Resources` object looks up resources from the resource file that matches the culture in the UICulture property. However, you can override this behavior and specify a particular culture to use for the resources. For more information, see Resources in Desktop Apps.

Properties

The properties of the `My.Resources` object provide read-only access to your application's resources. To add or remove resources, use the **Project Designer**. You can access resources added through the **Project Designer** by using `My.Resources.` *resourceName*.

You can also add or remove resource files by selecting your project in **Solution Explorer** and clicking **Add New Item** or **Add Existing Item** from the **Project** menu. You can access resources added in this manner by using `My.Resources.` *resourceFileName* `.` *resourceName*.

Each resource has a name, category, and value, and these resource settings determine how the property to access the resource appears in the `My.Resources` object. For resources added in the **Project Designer**:

- The name determines the name of the property,
- The resource data is the value of the property,
- The category determines the type of the property:

CATEGORY	PROPERTY DATA TYPE
Strings	String
Images	Bitmap
Icons	Icon
Audio	UnmanagedMemoryStream The UnmanagedMemoryStream class derives from the Stream class, so it can be used with methods that take streams, such as the Play method.

CATEGORY	PROPERTY DATA TYPE
Files	- String for text files. - Bitmap for image files. - Icon for icon files. - UnmanagedMemoryStream for sound files.
Other	Determined by the information in the designer's **Type** column.

Classes

The `My.Resources` object exposes each resource file as a class with shared properties. The class name is the same as the name of the resource file. As described in the previous section, the resources in a resource file are exposed as properties in the class.

Example

This example sets the title of a form to the string resource named `Form1Title` in the application resource file. For the example to work, the application must have a string named `Form1Title` in its resource file.

```
Sub SetFormTitle()
    Me.Text = My.Resources.Form1Title
End Sub
```

Example

This example sets the icon of the form to the icon named `Form1Icon` that is stored in the application's resource file. For the example to work, the application must have an icon named `Form1Icon` in its resource file.

```
Sub SetFormIcon()
    Me.Icon = My.Resources.Form1Icon
End Sub
```

Example

This example sets the background image of a form to the image resource named `Form1Background`, which is in the application resource file. For this example to work, the application must have an image resource named `Form1Background` in its resource file.

```
Sub SetFormBackgroundImage()
    Me.BackgroundImage = My.Resources.Form1Background
End Sub
```

Example

This example plays the sound that is stored as an audio resource named `Form1Greeting` in the application's resource file. For the example to work, the application must have an audio resource named `Form1Greeting` in its resource file. The `My.Computer.Audio.Play` method is available only for Windows Forms applications.

```
Sub PlayFormGreeting()
    My.Computer.Audio.Play(My.Resources.Form1Greeting,
        AudioPlayMode.Background)
End Sub
```

Example

This example retrieves the French-culture version of a string resource of the application. The resource is named `Message`. To change the culture that the `My.Resources` object uses, the example uses ChangeUICulture.

For this example to work, the application must have a string named `Message` in its resource file, and the application should have the French-culture version of that resource file, Resources.fr-FR.resx. If the application does not have the French-culture version of the resource file, the `My.Resource` object retrieves the resource from the default-culture resource file.

```
Sub ShowLocalizedMessage()
    Dim culture As String = My.Application.UICulture.Name
    My.Application.ChangeUICulture("fr-FR")
    MsgBox(My.Resources.Message)
    My.Application.ChangeUICulture(culture)
End Sub
```

See also

- Managing Application Resources (.NET)
- Resources in Desktop Apps

My.Settings Object

4/28/2019 • 2 minutes to read • Edit Online

Provides properties and methods for accessing the application's settings.

Remarks

The `My.Settings` object provides access to the application's settings and allows you to dynamically store and retrieve property settings and other information for your application. For more information, see Managing Application Settings (.NET).

Properties

The properties of the `My.Settings` object provide access to your application's settings. To add or remove settings, use the **Settings Designer**.

Each setting has a **Name**, **Type**, **Scope**, and **Value**, and these settings determine how the property to access each setting appears in the `My.Settings` object:

- **Name** determines the name of the property.
- **Type** determines the type of the property.
- **Scope** indicates if the property is read-only. If the value is **Application**, the property is read-only; if the value is **User**, the property is read-write.
- **Value** is the default value of the property.

Methods

METHOD	DESCRIPTION
`Reload`	Reloads the user settings from the last saved values.
`Save`	Saves the current user settings.

The `My.Settings` object also provides advanced properties and methods, inherited from the ApplicationSettingsBase class.

Tasks

The following table lists examples of tasks involving the `My.Settings` object.

TO	SEE
Read an application setting	How to: Read Application Settings in Visual Basic
Change a user setting	How to: Change User Settings in Visual Basic
Persist user settings	How to: Persist User Settings in Visual Basic

TO	SEE
Create a property grid for user settings	How to: Create Property Grids for User Settings in Visual Basic

Example

This example displays the value of the `Nickname` setting.

```
Sub ShowNickname()
    MsgBox("Nickname is " & My.Settings.Nickname)
End Sub
```

For this example to work, your application must have a `Nickname` setting, of type `String`.

See also

- ApplicationSettingsBase
- How to: Read Application Settings in Visual Basic
- How to: Change User Settings in Visual Basic
- How to: Persist User Settings in Visual Basic
- How to: Create Property Grids for User Settings in Visual Basic
- Managing Application Settings (.NET)

My.User Object

4/2/2019 • 2 minutes to read • Edit Online

Provides access to information about the current user.

Remarks

For information about the methods and properties of the `My.User` object, see Microsoft.VisualBasic.ApplicationServices.User.

For more information, see Accessing User Data.

Requirements

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- IPrincipal
- CurrentPrincipal
- User
- Current

My.WebServices Object

8/22/2019 • 2 minutes to read • Edit Online

Provides properties for creating and accessing a single instance of each XML Web service referenced by the current project.

Remarks

The `My.WebServices` object provides an instance of each Web service referenced by the current project. Each instance is instantiated on demand. You can access these Web services through the properties of the `My.WebServices` object. The name of the property is the same as the name of the Web service that the property accesses. Any class that inherits from SoapHttpClientProtocol is a Web service. For information about adding Web services to a project, see Accessing Application Web Services.

The `My.WebServices` object exposes only the Web services associated with the current project. It does not provide access to Web services declared in referenced DLLs. To access a Web service that a DLL provides, you must use the qualified name of the Web service, in the form *DllName*.*WebServiceName*. For more information, see Accessing Application Web Services.

The object and its properties are not available for Web applications.

Properties

Each property of the `My.WebServices` object provides access to an instance of a Web service referenced by the current project. The name of the property is the same as the name of the Web service that the property accesses, and the property type is the same as the Web service's type.

> **NOTE**
>
> If there is a name collision, the property name for accessing a Web service is *RootNamespace_Namespace_ServiceName*. For example, consider two Web services named `Service1`. If one of these services is in the root namespace `WindowsApplication1` and in the namespace `Namespace1`, you would access that service by using `My.WebServices.WindowsApplication1_Namespace1_Service1`.

When you first access one of the `My.WebServices` object's properties, it creates a new instance of the Web service and stores it. Subsequent accesses of that property return that instance of the Web service.

You can dispose of a Web service by assigning `Nothing` to the property for that Web service. The property setter assigns `Nothing` to the stored value. If you assign any value other than `Nothing` to the property, the setter throws an ArgumentException exception.

You can test whether a property of the `My.WebServices` object stores an instance of the Web service by using the `Is` or `IsNot` operator. You can use those operators to check if the value of the property is `Nothing`.

> **NOTE**
>
> Typically, the `Is` or `IsNot` operator has to read the value of the property to perform the comparison. However, if the property currently stores `Nothing`, the property creates a new instance of the Web service and then returns that instance. However, the Visual Basic compiler treats the properties of the `My.WebServices` object specially, and allows the `Is` or `IsNot` operator to check the status of the property without altering its value.

Example

This example calls the `FahrenheitToCelsius` method of the `TemperatureConverter` XML Web service, and returns the result.

```
Function ConvertFromFahrenheitToCelsius(
    ByVal dFahrenheit As Double) As Double

    Return My.WebServices.TemperatureConverter.FahrenheitToCelsius(dFahrenheit)
End Function
```

For this example to work, your project must reference a Web service named `Converter`, and that Web service must expose the `ConvertTemperature` method. For more information, see Accessing Application Web Services.

This code does not work in a Web application project.

Requirements

Availability by Project Type

PROJECT TYPE	AVAILABLE
Windows Application	**Yes**
Class Library	**Yes**
Console Application	**Yes**
Windows Control Library	**Yes**
Web Control Library	**Yes**
Windows Service	**Yes**
Web Site	No

See also

- SoapHttpClientProtocol
- ArgumentException
- Accessing Application Web Services

TextFieldParser Object

6/13/2019 • 2 minutes to read • Edit Online

Provides methods and properties for parsing structured text files.

Syntax

```
Public Class TextFieldParser
```

Remarks

For information about the methods and properties of the `TextFieldParser` object, see TextFieldParser.

For more information, see Reading from Files.

Requirements

Namespace: Microsoft.VisualBasic.FileIO

Class: TextFieldParser

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

Operators (Visual Basic)

10/19/2018 • 2 minutes to read • Edit Online

In this section

Operator precedence in Visual Basic

Operators listed by functionality

Data types of operator results

DirectCast operator

TryCast operator

New operator

Null-conditional operators

Arithmetic operators

Assignment operators

Bit Shift operators

Comparison operators

Concatenation operators

Logical/Bitwise operators

Miscellaneous operators

Related sections

Visual Basic language reference

Visual Basic

Operator Precedence in Visual Basic

8/21/2019 • 3 minutes to read • Edit Online

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called *operator precedence*.

Precedence Rules

When expressions contain operators from more than one category, they are evaluated according to the following rules:

- The arithmetic and concatenation operators have the order of precedence described in the following section, and all have greater precedence than the comparison, logical, and bitwise operators.
- All comparison operators have equal precedence, and all have greater precedence than the logical and bitwise operators, but lower precedence than the arithmetic and concatenation operators.
- The logical and bitwise operators have the order of precedence described in the following section, and all have lower precedence than the arithmetic, concatenation, and comparison operators.
- Operators with equal precedence are evaluated left to right in the order in which they appear in the expression.

Precedence Order

Operators are evaluated in the following order of precedence:

Await Operator

Await

Arithmetic and Concatenation Operators

Exponentiation (`^`)

Unary identity and negation (`+` , `-`)

Multiplication and floating-point division (`*` , `/`)

Integer division (`\`)

Modular arithmetic (`Mod`)

Addition and subtraction (`+` , `-`)

String concatenation (`&`)

Arithmetic bit shift (`<<` , `>>`)

Comparison Operators

All comparison operators (`=` , `<>` , `<` , `<=` , `>` , `>=` , `Is` , `IsNot` , `Like` , `TypeOf` ... `Is`)

Logical and Bitwise Operators

Negation (`Not`)

Conjunction (`And` , `AndAlso`)

Inclusive disjunction (`Or` , `OrElse`)

Exclusive disjunction (`Xor`)

Comments

The `=` operator is only the equality comparison operator, not the assignment operator.

The string concatenation operator (`&`) is not an arithmetic operator, but in precedence it is grouped with the arithmetic operators.

The `Is` and `IsNot` operators are object reference comparison operators. They do not compare the values of two objects; they check only to determine whether two object variables refer to the same object instance.

Associativity

When operators of equal precedence appear together in an expression, for example multiplication and division, the compiler evaluates each operation as it encounters it from left to right. The following example illustrates this.

```
Dim n1 As Integer = 96 / 8 / 4
Dim n2 As Integer = (96 / 8) / 4
Dim n3 As Integer = 96 / (8 / 4)
```

The first expression evaluates the division 96 / 8 (which results in 12) and then the division 12 / 4, which results in three. Because the compiler evaluates the operations for `n1` from left to right, the evaluation is the same when that order is explicitly indicated for `n2` . Both `n1` and `n2` have a result of three. By contrast, `n3` has a result of 48, because the parentheses force the compiler to evaluate 8 / 4 first.

Because of this behavior, operators are said to be *left associative* in Visual Basic.

Overriding Precedence and Associativity

You can use parentheses to force some parts of an expression to be evaluated before others. This can override both the order of precedence and the left associativity. Visual Basic always performs operations that are enclosed in parentheses before those outside. However, within parentheses, it maintains ordinary precedence and associativity, unless you use parentheses within the parentheses. The following example illustrates this.

```
Dim a, b, c, d, e, f, g As Double
a = 8.0
b = 3.0
c = 4.0
d = 2.0
e = 1.0
f = a - b + c / d * e
' The preceding line sets f to 7.0. Because of natural operator
' precedence and associativity, it is exactly equivalent to the
' following line.
f = (a - b) + ((c / d) * e)
' The following line overrides the natural operator precedence
' and left associativity.
g = (a - (b + c)) / (d * e)
' The preceding line sets g to 0.5.
```

See also

- = Operator

- Is Operator
- IsNot Operator
- Like Operator
- TypeOf Operator
- Await Operator
- Operators Listed by Functionality
- Operators and Expressions

Data Types of Operator Results (Visual Basic)

8/22/2019 • 7 minutes to read • Edit Online

Visual Basic determines the result data type of an operation based on the data types of the operands. In some cases this might be a data type with a greater range than that of either operand.

Data Type Ranges

The ranges of the relevant data types, in order from smallest to largest, are as follows:

- Boolean — two possible values
- SByte, Byte — 256 possible integral values
- Short, UShort — 65,536 (6.5...E+4) possible integral values
- Integer, UInteger — 4,294,967,296 (4.2...E+9) possible integral values
- Long, ULong — 18,446,744,073,709,551,615 (1.8...E+19) possible integral values
- Decimal — 1.5...E+29 possible integral values, maximum range 7.9...E+28 (absolute value)
- Single — maximum range 3.4...E+38 (absolute value)
- Double — maximum range 1.7...E+308 (absolute value)

For more information on Visual Basic data types, see Data Types.

If an operand evaluates to Nothing, the Visual Basic arithmetic operators treat it as zero.

Decimal Arithmetic

Note that the Decimal data type is neither floating-point nor integer.

If either operand of a `+`, `-`, `*`, `/`, or `Mod` operation is `Decimal` and the other is not `Single` or `Double`, Visual Basic widens the other operand to `Decimal`. It performs the operation in `Decimal`, and the result data type is `Decimal`.

Floating-Point Arithmetic

Visual Basic performs most floating-point arithmetic in Double, which is the most efficient data type for such operations. However, if one operand is Single and the other is not `Double`, Visual Basic performs the operation in `Single`. It widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

/ and ^ Operators

The `/` operator is defined only for the Decimal, Single, and Double data types. Visual Basic widens each operand as necessary to the appropriate data type before the operation, and the result has that data type.

The following table shows the result data types for the `/` operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	`Decimal`	`Single`	`Double`	Any integer type
`Decimal`	Decimal	Single	Double	Decimal
`Single`	Single	Single	Double	Single
`Double`	Double	Double	Double	Double
Any integer type	Decimal	Single	Double	Double

The `^` operator is defined only for the `Double` data type. Visual Basic widens each operand as necessary to `Double` before the operation, and the result data type is always `Double`.

Integer Arithmetic

The result data type of an integer operation depends on the data types of the operands. In general, Visual Basic uses the following policies for determining the result data type:

- If both operands of a binary operator have the same data type, the result has that data type. An exception is `Boolean`, which is forced to `Short`.
- If an unsigned operand participates with a signed operand, the result has a signed type with at least as large a range as either operand.
- Otherwise, the result usually has the larger of the two operand data types.

Note that the result data type might not be the same as either operand data type.

> **NOTE**
> The result data type is not always large enough to hold all possible values resulting from the operation. An OverflowException exception can occur if the value is too large for the result data type.

Unary + and – Operators

The following table shows the result data types for the two unary operators, `+` and `-`.

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`
Unary `+`	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong
Unary `-`	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal

<< and >> Operators

The following table shows the result data types for the two bit-shift operators, `<<` and `>>`. Visual Basic treats each bit-shift operator as a unary operator on its left operand (the bit pattern to be shifted).

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`

`<<`, `>>`	Short	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong

If the left operand is `Decimal`, `Single`, `Double`, or `String`, Visual Basic attempts to convert it to `Long` before the operation, and the result data type is `Long`. The right operand (the number of bit positions to shift) must be `Integer` or a type that widens to `Integer`.

Binary +, –, *, and Mod Operators

The following table shows the result data types for the binary `+` and `-` operators and the `*` and `Mod` operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`
`Boolean`	Short	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
`SByte`	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Decimal
`Byte`	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
`Short`	Short	Short	Short	Short	Integer	Integer	Long	Long	Decimal
`UShort`	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
`Integer`	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Decimal
`UInteger`	Long	Long	UIntege r	Long	UIntege r	Long	UInteger	Long	ULong
`Long`	Long	Long	Long	Long	Long	Long	Long	Long	Decimal
`ULong`	Decimal	Decimal	ULong	Decimal	ULong	Decimal	ULong	Decimal	ULong

\ Operator

The following table shows the result data types for the `\` operator. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`
`Boolean`	Short	SByte	Short	Short	Integer	Integer	Long	Long	Long
`SByte`	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
`Byte`	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
`Short`	Short	Short	Short	Short	Integer	Integer	Long	Long	Long

`UShort`	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
`Integer`	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
`UInteger`	Long	Long	UIntege r	Long	UIntege r	Long	UInteger	Long	ULong
`Long`	Long	Long	Long	Long	Long	Long	Long	Long	Long
`ULong`	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If either operand of the `\` operator is Decimal, Single, or Double, Visual Basic attempts to convert it to Long before the operation, and the result data type is `Long`.

Relational and Bitwise Comparisons

The result data type of a relational operation (`=` , `<>` , `<` , `>` , `<=` , `>=`) is always `Boolean` Boolean Data Type. The same is true for logical operations (`And` , `AndAlso` , `Not` , `Or` , `OrElse` , `Xor`) on `Boolean` operands.

The result data type of a bitwise logical operation depends on the data types of the operands. Note that `AndAlso` and `OrElse` are defined only for `Boolean` , and Visual Basic converts each operand as necessary to `Boolean` before performing the operation.

=, <>, <, >, <=, and >= Operators

If both operands are `Boolean` , Visual Basic considers `True` to be less than `False` . If a numeric type is compared with a `String` , Visual Basic attempts to convert the `String` to `Double` before the operation. A `Char` or `Date` operand can be compared only with another operand of the same data type. The result data type is always `Boolean` .

Bitwise Not Operator

The following table shows the result data types for the bitwise `Not` operator.

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`
`Not`	Boolean	SByte	Byte	Short	UShort	Integer	UInteger	Long	ULong

If the operand is `Decimal` , `Single` , `Double` , or `String` , Visual Basic attempts to convert it to `Long` before the operation, and the result data type is `Long` .

Bitwise And, Or, and Xor Operators

The following table shows the result data types for the bitwise `And` , `Or` , and `Xor` operators. Note that this table is symmetric; for a given combination of operand data types, the result data type is the same regardless of the order of the operands.

	`Boolean`	`SByte`	`Byte`	`Short`	`UShort`	`Integer`	`UInteger`	`Long`	`ULong`
`Boolean`	Boolean	SByte	Short	Short	Integer	Integer	Long	Long	Long

`SByte`	SByte	SByte	Short	Short	Integer	Integer	Long	Long	Long
`Byte`	Short	Short	Byte	Short	UShort	Integer	UInteger	Long	ULong
`Short`	Short	Short	Short	Short	Integer	Integer	Long	Long	Long
`UShort`	Integer	Integer	UShort	Integer	UShort	Integer	UInteger	Long	ULong
`Integer`	Integer	Integer	Integer	Integer	Integer	Integer	Long	Long	Long
`UInteger`	Long	Long	UIntege r	Long	UIntege r	Long	UInteger	Long	ULong
`Long`	Long	Long	Long	Long	Long	Long	Long	Long	Long
`ULong`	Long	Long	ULong	Long	ULong	Long	ULong	Long	ULong

If an operand is `Decimal`, `Single`, `Double`, or `String`, Visual Basic attempts to convert it to `Long` before the operation, and the result data type is the same as if that operand had already been `Long`.

Miscellaneous Operators

The `&` operator is defined only for concatenation of `String` operands. Visual Basic converts each operand as necessary to `String` before the operation, and the result data type is always `String`. For the purposes of the `&` operator, all conversions to `String` are considered to be widening, even if `Option Strict` is `On`.

The `Is` and `IsNot` operators require both operands to be of a reference type. The `TypeOf` ... `Is` expression requires the first operand to be of a reference type and the second operand to be the name of a data type. In all these cases the result data type is `Boolean`.

The `Like` operator is defined only for pattern matching of `String` operands. Visual Basic attempts to convert each operand as necessary to `String` before the operation. The result data type is always `Boolean`.

See also

- Data Types
- Operators and Expressions
- Arithmetic Operators in Visual Basic
- Comparison Operators in Visual Basic
- Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators
- Comparison Operators
- Option Strict Statement

Operators Listed by Functionality (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

See one of the categories listed below, or open this portion of the Help table of contents to see an alphabetical list of Visual Basic operators.

Categories of Operators

OPERATORS	DESCRIPTION
Arithmetic Operators	These operators perform mathematical calculations.
Assignment Operators	These operators perform assignment operations.
Comparison Operators	These operators perform comparisons.
Concatenation Operators	These operators combine strings.
Logical/Bitwise Operators	These operators perform logical operations.
Bit Shift Operators	These operators perform arithmetic shifts on bit patterns.
Miscellaneous Operators	These operators perform miscellaneous operations.

See also

- Operators and Expressions
- Operator Precedence in Visual Basic

Arithmetic Operators (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following are the arithmetic operators defined in Visual Basic.

^ Operator

* Operator

/ Operator

\ Operator

Mod Operator

+ Operator (unary and binary)

- Operator (unary and binary)

See also

- Operator Precedence in Visual Basic
- Arithmetic Operators in Visual Basic

Assignment Operators (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following are the assignment operators defined in Visual Basic.

= Operator

^= Operator

*= Operator

/= Operator

\= Operator

+= Operator

-= Operator

<<= Operator

>>= Operator

&= Operator

See also

- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

Bit Shift Operators (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following are the bit shift operators defined in Visual Basic.

<< Operator

>> Operator

See also

- Operators Listed by Functionality

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

Is Operator

IsNot Operator

Like Operator

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. `Is`, `IsNot`, and `Like` are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

`result`
Required. A `Boolean` value representing the result of the comparison.

`expression1`, `expression2`
Required. Any expression.

`comparisonoperator`
Required. Any relational comparison operator.

`object1`, `object2`
Required. Any reference object names.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether `result` is `True` or `False`.

OPERATOR	`TRUE` IF	`FALSE` IF
`<` (Less than)	`expression1 < expression2`	`expression1 >= expression2`
`<=` (Less than or equal to)	`expression1 <= expression2`	`expression1 > expression2`
`>` (Greater than)	`expression1 > expression2`	`expression1 <= expression2`
`>=` (Greater than or equal to)	`expression1 >= expression2`	`expression1 < expression2`
`=` (Equal to)	`expression1 = expression2`	`expression1 <> expression2`
`<>` (Not equal to)	`expression1 <> expression2`	`expression1 = expression2`

> **NOTE**
> The = Operator is also used as an assignment operator.

The `Is` operator, the `IsNot` operator, and the `Like` operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type `Single` to one of type `Double`, the `Single` expression is converted to `Double`. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type `Decimal` to an expression of type `Single` or `Double`, the `Decimal` expression is converted to `Single` or `Double`. For `Decimal` expressions, any fractional value less than 1E-28 might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you should take care when using equality (`=`) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator. For more information, see Troubleshooting Data Types.

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the `Option Compare` setting.

`Option Compare Binary` bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

`Option Compare Text` bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set `Option Compare Text` and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set `Option Compare Text`, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting `Option Compare Binary` or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with `Object` expressions is not allowed under `Option Strict On`. When `Option Strict` is `Off`, and either `expression1` or `expression2` is an `Object` expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

IF OPERANDS ARE	COMPARISON IS
Both `String`	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to `Double`, numeric comparison.
One numeric and one `String`	The `String` is converted to a `Double` and numeric comparison is performed. If the `String` cannot be converted to `Double`, an InvalidCastException is thrown.
Either or both are reference types other than `String`	An InvalidCastException is thrown.

Numeric comparisons treat `Nothing` as 0. String comparisons treat `Nothing` as `""` (an empty string).

Overloading

The relational comparison operators (`<`, `<=`, `>`, `>=`, `=`, `<>`) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures.

Notice that the = Operator can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a `Boolean` result that represents whether or not the stated expression evaluates to `True`. When you apply the `>` and `<` operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- InvalidCastException
- = Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Comparison Operators in Visual Basic

Concatenation Operators (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following are the concatenation operators defined in Visual Basic.

& Operator

+ Operator

See also

- System.Text
- StringBuilder
- Operator Precedence in Visual Basic
- Concatenation Operators in Visual Basic

Logical/Bitwise Operators (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The following are the logical/bitwise operators defined in Visual Basic.

And Operator

Not Operator

Or Operator

Xor Operator

AndAlso Operator

OrElse Operator

IsFalse Operator

IsTrue Operator

See also

- Operator Precedence in Visual Basic
- Logical and Bitwise Operators in Visual Basic

Miscellaneous operators (Visual Basic)

1/23/2019 • 2 minutes to read • Edit Online

The following are miscellaneous operators defined in Visual Basic.

?. null-conditional operator

?() null-conditional operator

AddressOf operator

Await operator

GetType operator

Function expression

If operator

TypeOf operator

See also

- Operators listed by functionality

& Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Generates a string concatenation of two expressions.

Syntax

```
result = expression1 & expression2
```

Parts

`result`
Required. Any `String` or `Object` variable.

`expression1`
Required. Any expression with a data type that widens to `String`.

`expression2`
Required. Any expression with a data type that widens to `String`.

Remarks

If the data type of `expression1` or `expression2` is not `String` but widens to `String`, it is converted to `String`. If either of the data types does not widen to `String`, the compiler generates an error.

The data type of `result` is `String`. If one or both expressions evaluate to Nothing or have a value of DBNull.Value, they are treated as a string with a value of "".

> **NOTE**
> The `&` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

> **NOTE**
> The ampersand (&) character can also be used to identify variables as type `Long`. For more information, see Type Characters.

Example

This example uses the `&` operator to force string concatenation. The result is a string value representing the concatenation of the two string operands.

```
Dim sampleStr As String
sampleStr = "Hello" & " World"
' The preceding statement sets sampleStr to "Hello World".
```

See also

- &= Operator
- Concatenation Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Concatenation Operators in Visual Basic

&= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Concatenates a `String` expression to a `String` variable or property and assigns the result to the variable or property.

Syntax

```
variableorproperty &= expression
```

Parts

`variableorproperty`
Required. Any `String` variable or property.

`expression`
Required. Any `String` expression.

Remarks

The element on the left side of the `&=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly. The `&=` operator concatenates the `String` expression on its right to the `String` variable or property on its left, and assigns the result to the variable or property on its left.

Overloading

The & Operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `&` operator affects the behavior of the `&=` operator. If your code uses `&=` on a class or structure that overloads `&`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `&=` operator to concatenate two `String` variables and assign the result to the first variable.

```
Dim var1 As String = "Hello "
Dim var2 As String = "World!"
var1 &= var2
' The value of var1 is now "Hello World!".
```

See also

- & Operator
- += Operator
- Assignment Operators
- Concatenation Operators
- Operator Precedence in Visual Basic

- Operators Listed by Functionality
- Statements

* Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Multiplies two numbers.

Syntax

```
number1 * number2
```

Parts

TERM	DEFINITION
`number1`	Required. Any numeric expression.
`number2`	Required. Any numeric expression.

Result

The result is the product of `number1` and `number2`.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Remarks

The data type of the result depends on the types of the operands. The following table shows how the data type of the result is determined.

OPERAND DATA TYPES	RESULT DATA TYPE
Both expressions are integral data types (SByte, Byte, Short, UShort, Integer, UInteger, Long, ULong)	A numeric data type appropriate for the data types of `number1` and `number2`. See the "Integer Arithmetic" tables in Data Types of Operator Results.
Both expressions are Decimal	`Decimal`
Both expressions are Single	`Single`
Either expression is a floating-point data type (`Single` or Double) but not both `Single` (note `Decimal` is not a floating-point data type)	`Double`

If an expression evaluates to Nothing, it is treated as zero.

Overloading

The `*` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

This example uses the `*` operator to multiply two numbers. The result is the product of the two operands.

```
Dim testValue As Double
testValue = 2 * 2
' The preceding statement sets testValue to 4.
testValue = 459.35 * 334.9
' The preceding statement sets testValue to 153836.315.
```

See also

- *= Operator
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

*= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Multiplies the value of a variable or property by the value of an expression and assigns the result to the variable or property.

Syntax

```
variableorproperty *= expression
```

Parts

`variableorproperty`
Required. Any numeric variable or property.

`expression`
Required. Any numeric expression.

Remarks

The element on the left side of the `*=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `*=` operator first multiplies the value of the expression (on the right-hand side of the operator) by the value of the variable or property (on the left-hand side of the operator). The operator then assigns the result of that operation to the variable or property.

Overloading

The * Operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `*` operator affects the behavior of the `*=` operator. If your code uses `*=` on a class or structure that overloads `*`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `*=` operator to multiply one `Integer` variable by a second and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 *= var2
' The value of var1 is now 30.
```

See also

- * Operator
- Assignment Operators

- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

+ Operator (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Adds two numbers or returns the positive value of a numeric expression. Can also be used to concatenate two string expressions.

Syntax

```
expression1 + expression2
- or -
+ expression1
```

Parts

TERM	DEFINITION
`expression1`	Required. Any numeric or string expression.
`expression2`	Required unless the `+` operator is calculating a negative value. Any numeric or string expression.

Result

If `expression1` and `expression2` are both numeric, the result is their arithmetic sum.

If `expression2` is absent, the `+` operator is the *unary* identity operator for the unchanged value of an expression. In this sense, the operation consists of retaining the sign of `expression1`, so the result is negative if `expression1` is negative.

If `expression1` and `expression2` are both strings, the result is the concatenation of their values.

If `expression1` and `expression2` are of mixed types, the action taken depends on their types, their contents, and the setting of the Option Strict Statement. For more information, see the tables in "Remarks."

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`, and `String`.

Remarks

In general, `+` performs arithmetic addition when possible, and concatenates only when both expressions are strings.

If neither expression is an `Object`, Visual Basic takes the following actions.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both expressions are numeric data types (`SByte` , `Byte` , `Short` , `UShort` , `Integer` , `UInteger` , `Long` , `ULong` , `Decimal` , `Single` , or `Double`)	Add. The result data type is a numeric type appropriate for the data types of `expression1` and `expression2` . See the "Integer Arithmetic" tables in Data Types of Operator Results.
Both expressions are of type `String`	Concatenate.
One expression is a numeric data type and the other is a string	If `Option Strict` is `On` , then generate a compiler error. If `Option Strict` is `Off` , then implicitly convert the `String` to `Double` and add. If the `String` cannot be converted to `Double` , then throw an InvalidCastException exception.
One expression is a numeric data type, and the other is Nothing	Add, with `Nothing` valued as zero.
One expression is a string, and the other is `Nothing`	Concatenate, with `Nothing` valued as "".

If one expression is an `Object` expression, Visual Basic takes the following actions.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
`Object` expression holds a numeric value and the other is a numeric data type	If `Option Strict` is `On` , then generate a compiler error. If `Option Strict` is `Off` , then add.
`Object` expression holds a numeric value and the other is of type `String`	If `Option Strict` is `On` , then generate a compiler error. If `Option Strict` is `Off` , then implicitly convert the `String` to `Double` and add. If the `String` cannot be converted to `Double` , then throw an InvalidCastException exception.
`Object` expression holds a string and the other is a numeric data type	If `Option Strict` is `On` , then generate a compiler error. If `Option Strict` is `Off` , then implicitly convert the string `Object` to `Double` and add. If the string `Object` cannot be converted to `Double` , then throw an InvalidCastException exception.
`Object` expression holds a string and the other is of type `String`	If `Option Strict` is `On` , then generate a compiler error. If `Option Strict` is `Off` , then implicitly convert `Object` to `String` and concatenate.

If both expressions are `Object` expressions, Visual Basic takes the following actions (`Option Strict Off` only).

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both `Object` expressions hold numeric values	Add.

DATA TYPES OF EXPRESSIONS	ACTION BY COMPILER
Both `Object` expressions are of type `String`	Concatenate.
One `Object` expression holds a numeric value and the other holds a string	Implicitly convert the string `Object` to `Double` and add. If the string `Object` cannot be converted to a numeric value, then throw an InvalidCastException exception.

If either `Object` expression evaluates to Nothing or DBNull, the `+` operator treats it as a `String` with a value of "".

> **NOTE**
> When you use the `+` operator, you might not be able to determine whether addition or string concatenation will occur. Use the `&` operator for concatenation to eliminate ambiguity and to provide self-documenting code.

Overloading

The `+` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `+` operator to add numbers. If the operands are both numeric, Visual Basic computes the arithmetic result. The arithmetic result represents the sum of the two operands.

```
Dim sumNumber As Integer
sumNumber = 2 + 2
sumNumber = 4257.04 + 98112
' The preceding statements set sumNumber to 4 and 102369.
```

You can also use the `+` operator to concatenate strings. If the operands are both strings, Visual Basic concatenates them. The concatenation result represents a single string consisting of the contents of the two operands one after the other.

If the operands are of mixed types, the result depends on the setting of the Option Strict Statement. The following example illustrates the result when `Option Strict` is `On`.

```
Option Strict On
```

```
Dim var1 As String = "34"
Dim var2 As Integer = 6
Dim concatenatedNumber As Integer = var1 + var2
```

```
' The preceding statement generates a COMPILER ERROR.
```

The following example illustrates the result when `Option Strict` is `Off`.

```
Option Strict Off
```

```
Dim var1 As String = "34"
Dim var2 As Integer = 6
Dim concatenatedNumber As Integer = var1 + var2
```

```
' The preceding statement returns 40 after the string in var1 is
' converted to a numeric value. This might be an unexpected result.
' We do not recommend use of Option Strict Off for these operations.
```

To eliminate ambiguity, you should use the `&` operator instead of `+` for concatenation.

See also

- & Operator
- Concatenation Operators
- Arithmetic Operators
- Operators Listed by Functionality
- Operator Precedence in Visual Basic
- Arithmetic Operators in Visual Basic
- Option Strict Statement

+= Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Adds the value of a numeric expression to the value of a numeric variable or property and assigns the result to the variable or property. Can also be used to concatenate a `String` expression to a `String` variable or property and assign the result to the variable or property.

Syntax

```
variableorproperty += expression
```

Parts

`variableorproperty`
Required. Any numeric or `String` variable or property.

`expression`
Required. Any numeric or `String` expression.

Remarks

The element on the left side of the `+=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `+=` operator adds the value on its right to the variable or property on its left, and assigns the result to the variable or property on its left. The `+=` operator can also be used to concatenate the `String` expression on its right to the `String` variable or property on its left, and assign the result to the variable or property on its left.

> **NOTE**
> When you use the `+=` operator, you might not be able to determine whether addition or string concatenation will occur. Use the `&=` operator for concatenation to eliminate ambiguity and to provide self-documenting code.

This assignment operator implicitly performs widening but not narrowing conversions if the compilation environment enforces strict semantics. For more information on these conversions, see Widening and Narrowing Conversions. For more information on strict and permissive semantics, see Option Strict Statement.

If permissive semantics are allowed, the `+=` operator implicitly performs a variety of string and numeric conversions identical to those performed by the `+` operator. For details on these conversions, see + Operator.

Overloading

The `+` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `+` operator affects the behavior of the `+=` operator. If your code uses `+=` on a class or structure that overloads `+`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `+=` operator to combine the value of one variable with another. The first part uses `+=` with numeric variables to add one value to another. The second part uses `+=` with `String` variables to concatenate one value with another. In both cases, the result is assigned to the first variable.

```
' This part uses numeric variables.
Dim num1 As Integer = 10
Dim num2 As Integer = 3
num1 += num2
```

```
' This part uses string variables.
Dim str1 As String = "10"
Dim str2 As String = "3"
str1 += str2
```

The value of `num1` is now 13, and the value of `str1` is now "103".

See also

- + Operator
- Assignment Operators
- Arithmetic Operators
- Concatenation Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

= Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Assigns a value to a variable or property.

Syntax

```
variableorproperty = value
```

Parts

`variableorproperty`
Any writable variable or any property.

`value`
Any literal, constant, or expression.

Remarks

The element on the left side of the equal sign (`=`) can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly. The `=` operator assigns the value on its right to the variable or property on its left.

> **NOTE**
> The `=` operator is also used as a comparison operator. For details, see Comparison Operators.

Overloading

The `=` operator can be overloaded only as a relational comparison operator, not as an assignment operator. For more information, see Operator Procedures.

Example

The following example demonstrates the assignment operator. The value on the right is assigned to the variable on the left.

```
Dim testInt As Integer
Dim testString As String
Dim testButton As System.Windows.Forms.Button
Dim testObject As Object
testInt = 42
testString = "This is an example of a string literal."
testButton = New System.Windows.Forms.Button()
testObject = testInt
testObject = testString
testObject = testButton
```

See also

- &= Operator
- *= Operator
- += Operator
- -= Operator (Visual Basic)
- /= Operator (Visual Basic)
- \= Operator
- ^= Operator
- Statements
- Comparison Operators
- ReadOnly
- Local Type Inference

- Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Returns the difference between two numeric expressions or the negative value of a numeric expression.

Syntax

```
      expression1 - expression2
- or -
- expression1
```

Parts

`expression1`
Required. Any numeric expression.

`expression2`
Required unless the `-` operator is calculating a negative value. Any numeric expression.

Result

The result is the difference between `expression1` and `expression2`, or the negated value of `expression1`.

The result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Integer Arithmetic" tables in Data Types of Operator Results.

Supported Types

All numeric types. This includes the unsigned and floating-point types and `Decimal`.

Remarks

In the first usage shown in the syntax shown previously, the `-` operator is the *binary* arithmetic subtraction operator for the difference between two numeric expressions.

In the second usage shown in the syntax shown previously, the `-` operator is the *unary* negation operator for the negative value of an expression. In this sense, the negation consists of reversing the sign of `expression1` so that the result is positive if `expression1` is negative.

If either expression evaluates to Nothing, the `-` operator treats it as zero.

> **NOTE**
> The `-` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, make sure that you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `-` operator to calculate and return the difference between two numbers, and then

to negate a number.

```
Dim binaryResult As Double = 459.35 - 334.9
Dim unaryResult As Double = -334.9
```

Following the execution of these statements, `binaryResult` contains 124.45 and `unaryResult` contains –334.90.

See also

- -= Operator (Visual Basic)
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

-= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Subtracts the value of an expression from the value of a variable or property and assigns the result to the variable or property.

Syntax

```
variableorproperty -= expression
```

Parts

`variableorproperty`
Required. Any numeric variable or property.

`expression`
Required. Any numeric expression.

Remarks

The element on the left side of the `-=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `-=` operator first subtracts the value of the expression (on the right-hand side of the operator) from the value of the variable or property (on the left-hand side of the operator). The operator then assigns the result of that operation to the variable or property.

Overloading

The - Operator (Visual Basic) can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `-` operator affects the behavior of the `-=` operator. If your code uses `-=` on a class or structure that overloads `-`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `-=` operator to subtract one `Integer` variable from another and assign the result to the latter variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 -= var2
' The value of var1 is now 7.
```

See also

- - Operator (Visual Basic)
- Assignment Operators

- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

Is Operator

IsNot Operator

Like Operator

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. `Is`, `IsNot`, and `Like` are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

`result`
Required. A `Boolean` value representing the result of the comparison.

`expression1`, `expression2`
Required. Any expression.

`comparisonoperator`
Required. Any relational comparison operator.

`object1`, `object2`
Required. Any reference object names.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether `result` is `True` or `False`.

OPERATOR	`TRUE` IF	`FALSE` IF
`<` (Less than)	`expression1 < expression2`	`expression1 >= expression2`
`<=` (Less than or equal to)	`expression1 <= expression2`	`expression1 > expression2`
`>` (Greater than)	`expression1 > expression2`	`expression1 <= expression2`
`>=` (Greater than or equal to)	`expression1 >= expression2`	`expression1 < expression2`
`=` (Equal to)	`expression1 = expression2`	`expression1 <> expression2`
`<>` (Not equal to)	`expression1 <> expression2`	`expression1 = expression2`

> **NOTE**
> The = Operator is also used as an assignment operator.

The `Is` operator, the `IsNot` operator, and the `Like` operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type `Single` to one of type `Double`, the `Single` expression is converted to `Double`. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type `Decimal` to an expression of type `Single` or `Double`, the `Decimal` expression is converted to `Single` or `Double`. For `Decimal` expressions, any fractional value less than 1E-28 might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you should take care when using equality (`=`) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator. For more information, see Troubleshooting Data Types.

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the `Option Compare` setting.

`Option Compare Binary` bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

`Option Compare Text` bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set `Option Compare Text` and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set `Option Compare Text`, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting `Option Compare Binary` or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with `Object` expressions is not allowed under `Option Strict On`. When `Option Strict` is `Off`, and either `expression1` or `expression2` is an `Object` expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

IF OPERANDS ARE	COMPARISON IS
Both `String`	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to `Double`, numeric comparison.
One numeric and one `String`	The `String` is converted to a `Double` and numeric comparison is performed. If the `String` cannot be converted to `Double`, an InvalidCastException is thrown.
Either or both are reference types other than `String`	An InvalidCastException is thrown.

Numeric comparisons treat `Nothing` as 0. String comparisons treat `Nothing` as `""` (an empty string).

Overloading

The relational comparison operators (`<`, `<=`, `>`, `>=`, `=`, `<>`) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures.

Notice that the = Operator can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a `Boolean` result that represents whether or not the stated expression evaluates to `True`. When you apply the `>` and `<` operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- InvalidCastException
- = Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Comparison Operators in Visual Basic

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

Is Operator

IsNot Operator

Like Operator

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. `Is`, `IsNot`, and `Like` are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

`result`
Required. A `Boolean` value representing the result of the comparison.

`expression1`, `expression2`
Required. Any expression.

`comparisonoperator`
Required. Any relational comparison operator.

`object1`, `object2`
Required. Any reference object names.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether `result` is `True` or `False`.

OPERATOR	`TRUE` IF	`FALSE` IF
`<` (Less than)	`expression1 < expression2`	`expression1 >= expression2`
`<=` (Less than or equal to)	`expression1 <= expression2`	`expression1 > expression2`
`>` (Greater than)	`expression1 > expression2`	`expression1 <= expression2`
`>=` (Greater than or equal to)	`expression1 >= expression2`	`expression1 < expression2`
`=` (Equal to)	`expression1 = expression2`	`expression1 <> expression2`
`<>` (Not equal to)	`expression1 <> expression2`	`expression1 = expression2`

> **NOTE**
> The = Operator is also used as an assignment operator.

The `Is` operator, the `IsNot` operator, and the `Like` operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type `Single` to one of type `Double`, the `Single` expression is converted to `Double`. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type `Decimal` to an expression of type `Single` or `Double`, the `Decimal` expression is converted to `Single` or `Double`. For `Decimal` expressions, any fractional value less than 1E-28 might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you should take care when using equality (`=`) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator. For more information, see Troubleshooting Data Types.

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the `Option Compare` setting.

`Option Compare Binary` bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

`Option Compare Text` bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set `Option Compare Text` and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set `Option Compare Text`, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting `Option Compare Binary` or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with `Object` expressions is not allowed under `Option Strict On`. When `Option Strict` is `Off`, and either `expression1` or `expression2` is an `Object` expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

IF OPERANDS ARE	COMPARISON IS
Both `String`	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to `Double`, numeric comparison.
One numeric and one `String`	The `String` is converted to a `Double` and numeric comparison is performed. If the `String` cannot be converted to `Double`, an InvalidCastException is thrown.
Either or both are reference types other than `String`	An InvalidCastException is thrown.

Numeric comparisons treat `Nothing` as 0. String comparisons treat `Nothing` as `""` (an empty string).

Overloading

The relational comparison operators (`<`, `<=`, `>`, `>=`, `=`, `<>`) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures.

Notice that the = Operator can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a `Boolean` result that represents whether or not the stated expression evaluates to `True`. When you apply the `>` and `<` operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- InvalidCastException
- = Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Comparison Operators in Visual Basic

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

Is Operator

IsNot Operator

Like Operator

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. `Is`, `IsNot`, and `Like` are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

`result`
Required. A `Boolean` value representing the result of the comparison.

`expression1`, `expression2`
Required. Any expression.

`comparisonoperator`
Required. Any relational comparison operator.

`object1`, `object2`
Required. Any reference object names.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether `result` is `True` or `False`.

OPERATOR	`TRUE` IF	`FALSE` IF
`<` (Less than)	`expression1 < expression2`	`expression1 >= expression2`
`<=` (Less than or equal to)	`expression1 <= expression2`	`expression1 > expression2`
`>` (Greater than)	`expression1 > expression2`	`expression1 <= expression2`
`>=` (Greater than or equal to)	`expression1 >= expression2`	`expression1 < expression2`
`=` (Equal to)	`expression1 = expression2`	`expression1 <> expression2`
`<>` (Not equal to)	`expression1 <> expression2`	`expression1 = expression2`

> **NOTE**
> The = Operator is also used as an assignment operator.

The `Is` operator, the `IsNot` operator, and the `Like` operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type `Single` to one of type `Double`, the `Single` expression is converted to `Double`. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type `Decimal` to an expression of type `Single` or `Double`, the `Decimal` expression is converted to `Single` or `Double`. For `Decimal` expressions, any fractional value less than 1E-28 might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you should take care when using equality (`=`) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator. For more information, see Troubleshooting Data Types.

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the `Option Compare` setting.

`Option Compare Binary` bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

`Option Compare Text` bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set `Option Compare Text` and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set `Option Compare Text`, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting `Option Compare Binary` or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with `Object` expressions is not allowed under `Option Strict On`. When `Option Strict` is `Off`, and either `expression1` or `expression2` is an `Object` expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

IF OPERANDS ARE	COMPARISON IS
Both `String`	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to `Double`, numeric comparison.
One numeric and one `String`	The `String` is converted to a `Double` and numeric comparison is performed. If the `String` cannot be converted to `Double`, an InvalidCastException is thrown.
Either or both are reference types other than `String`	An InvalidCastException is thrown.

Numeric comparisons treat `Nothing` as 0. String comparisons treat `Nothing` as `""` (an empty string).

Overloading

The relational comparison operators (`<`, `<=`, `>`, `>=`, `=`, `<>`) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures.

Notice that the = Operator can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a `Boolean` result that represents whether or not the stated expression evaluates to `True`. When you apply the `>` and `<` operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- InvalidCastException
- = Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Comparison Operators in Visual Basic

Comparison Operators (Visual Basic)

8/22/2019 • 5 minutes to read • Edit Online

The following are the comparison operators defined in Visual Basic.

`<` operator

`<=` operator

`>` operator

`>=` operator

`=` operator

`<>` operator

Is Operator

IsNot Operator

Like Operator

These operators compare two expressions to determine whether or not they are equal, and if not, how they differ. `Is`, `IsNot`, and `Like` are discussed in detail on separate Help pages. The relational comparison operators are discussed in detail on this page.

Syntax

```
result = expression1 comparisonoperator expression2
result = object1 [Is | IsNot] object2
result = string Like pattern
```

Parts

`result`
Required. A `Boolean` value representing the result of the comparison.

`expression1`, `expression2`
Required. Any expression.

`comparisonoperator`
Required. Any relational comparison operator.

`object1`, `object2`
Required. Any reference object names.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression or range of characters.

Remarks

The following table contains a list of the relational comparison operators and the conditions that determine whether `result` is `True` or `False`.

OPERATOR	`TRUE` IF	`FALSE` IF
`<` (Less than)	`expression1 < expression2`	`expression1 >= expression2`
`<=` (Less than or equal to)	`expression1 <= expression2`	`expression1 > expression2`
`>` (Greater than)	`expression1 > expression2`	`expression1 <= expression2`
`>=` (Greater than or equal to)	`expression1 >= expression2`	`expression1 < expression2`
`=` (Equal to)	`expression1 = expression2`	`expression1 <> expression2`
`<>` (Not equal to)	`expression1 <> expression2`	`expression1 = expression2`

> **NOTE**
> The = Operator is also used as an assignment operator.

The `Is` operator, the `IsNot` operator, and the `Like` operator have specific comparison functionalities that differ from the operators in the preceding table.

Comparing Numbers

When you compare an expression of type `Single` to one of type `Double`, the `Single` expression is converted to `Double`. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when you compare an expression of type `Decimal` to an expression of type `Single` or `Double`, the `Decimal` expression is converted to `Single` or `Double`. For `Decimal` expressions, any fractional value less than 1E-28 might be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, you should take care when using equality (`=`) to compare two floating-point variables. It is safer to test whether the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Floating-point Imprecision

When you work with floating-point numbers, keep in mind that they do not always have a precise representation in memory. This could lead to unexpected results from certain operations, such as value comparison and the Mod Operator. For more information, see Troubleshooting Data Types.

Comparing Strings

When you compare strings, the string expressions are evaluated based on their alphabetical sort order, which depends on the `Option Compare` setting.

`Option Compare Binary` bases string comparisons on a sort order derived from the internal binary representations of the characters. The sort order is determined by the code page. The following example shows a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

`Option Compare Text` bases string comparisons on a case-insensitive, textual sort order determined by your application's locale. When you set `Option Compare Text` and sort the characters in the preceding example, the following text sort order applies:

```
(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)
```

Locale Dependence

When you set `Option Compare Text`, the result of a string comparison can depend on the locale in which the application is running. Two characters might compare as equal in one locale but not in another. If you are using a string comparison to make important decisions, such as whether to accept an attempt to log on, you should be alert to locale sensitivity. Consider either setting `Option Compare Binary` or calling the StrComp, which takes the locale into account.

Typeless Programming with Relational Comparison Operators

The use of relational comparison operators with `Object` expressions is not allowed under `Option Strict On`. When `Option Strict` is `Off`, and either `expression1` or `expression2` is an `Object` expression, the run-time types determine how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands.

IF OPERANDS ARE	COMPARISON IS
Both `String`	Sort comparison based on string sorting characteristics.
Both numeric	Objects converted to `Double`, numeric comparison.
One numeric and one `String`	The `String` is converted to a `Double` and numeric comparison is performed. If the `String` cannot be converted to `Double`, an InvalidCastException is thrown.
Either or both are reference types other than `String`	An InvalidCastException is thrown.

Numeric comparisons treat `Nothing` as 0. String comparisons treat `Nothing` as `""` (an empty string).

Overloading

The relational comparison operators (`<`, `<=`, `>`, `>=`, `=`, `<>`) can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. If your code uses any of these operators on such a class or structure, be sure you understand the redefined behavior. For more information, see Operator Procedures.

Notice that the = Operator can be overloaded only as a relational comparison operator, not as an assignment operator.

Example

The following example shows various uses of relational comparison operators, which you use to compare expressions. Relational comparison operators return a `Boolean` result that represents whether or not the stated expression evaluates to `True`. When you apply the `>` and `<` operators to strings, the comparison is made using the normal alphabetical sorting order of the strings. This order can be dependent on your locale setting. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim x As testClass
Dim y As New testClass()
x = y
If x Is y Then
    ' Insert code to run if x and y point to the same instance.
End If
```

In the preceding example, the first comparison returns `False` and the remaining comparisons return `True`.

See also

- InvalidCastException
- = Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Comparison Operators in Visual Basic

<< Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Performs an arithmetic left shift on a bit pattern.

Syntax

```
result = pattern << amount
```

Parts

`result`
Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of `pattern`.

`pattern`
Required. Integral numeric expression. The bit pattern to be shifted. The data type must be an integral type (`SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`
Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be `Integer` or widen to `Integer`.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

To prevent a shift by more bits than the result can hold, Visual Basic masks the value of `amount` with a size mask that corresponds to the data type of `pattern`. The binary AND of these values is used for the shift amount. The size masks are as follows:

DATA TYPE OF `PATTERN`	SIZE MASK (DECIMAL)	SIZE MASK (HEXADECIMAL)
`SByte`, `Byte`	7	&H00000007
`Short`, `UShort`	15	&H0000000F
`Integer`, `UInteger`	31	&H0000001F
`Long`, `ULong`	63	&H0000003F

If `amount` is zero, the value of `result` is identical to the value of `pattern`. If `amount` is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

NOTE

The `<<` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure that you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `<<` operator to perform arithmetic left shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim pattern As Short = 192
' The bit pattern is 0000 0000 1100 0000.
Dim result1, result2, result3, result4, result5 As Short
result1 = pattern << 0
result2 = pattern << 4
result3 = pattern << 9
result4 = pattern << 17
result5 = pattern << -1
```

The results of the previous example are as follows:

- `result1` is 192 (0000 0000 1100 0000).
- `result2` is 3072 (0000 1100 0000 0000).
- `result3` is -32768 (1000 0000 0000 0000).
- `result4` is 384 (0000 0001 1000 0000).
- `result5` is 0 (shifted 15 places to the left).

The shift amount for `result4` is calculated as 17 AND 15, which equals 1.

See also

- Bit Shift Operators
- Assignment Operators
- <<= Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

<<= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Performs an arithmetic left shift on the value of a variable or property and assigns the result back to the variable or property.

Syntax

```
variableorproperty <<= amount
```

Parts

`variableorproperty`
Required. Variable or property of an integral type (`SByte` , `Byte` , `Short` , `UShort` , `Integer` , `UInteger` , `Long` , or `ULong`).

`amount`
Required. Numeric expression of a data type that widens to `Integer` .

Remarks

The element on the left side of the `<<=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `<<=` operator first performs an arithmetic left shift on the value of the variable or property. The operator then assigns the result of that operation back to that variable or property.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

Overloading

The << Operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `<<` operator affects the behavior of the `<<=` operator. If your code uses `<<=` on a class or structure that overloads `<<` , be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `<<=` operator to shift the bit pattern of an `Integer` variable left by the specified amount and assign the result to the variable.

```
Dim var As Integer = 10
Dim shift As Integer = 3
var <<= shift
' The value of var is now 80.
```

See also

- << Operator
- Assignment Operators
- Bit Shift Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

>> Operator (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Performs an arithmetic right shift on a bit pattern.

Syntax

```
result = pattern >> amount
```

Parts

`result`
Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of `pattern`.

`pattern`
Required. Integral numeric expression. The bit pattern to be shifted. The data type must be an integral type (`SByte`, `Byte`, `Short`, `UShort`, `Integer`, `UInteger`, `Long`, or `ULong`).

`amount`
Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be `Integer` or widen to `Integer`.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost (sign) bit is propagated into the bit positions vacated at the left. This means that if `pattern` has a negative value, the vacated positions are set to one; otherwise they are set to zero.

Note that the data types `Byte`, `UShort`, `UInteger`, and `ULong` are unsigned, so there is no sign bit to propagate. If `pattern` is of any unsigned type, the vacated positions are always set to zero.

To prevent shifting by more bits than the result can hold, Visual Basic masks the value of `amount` with a size mask corresponding to the data type of `pattern`. The binary AND of these values is used for the shift amount. The size masks are as follows:

DATA TYPE OF `PATTERN`	SIZE MASK (DECIMAL)	SIZE MASK (HEXADECIMAL)
`SByte`, `Byte`	7	&H00000007
`Short`, `UShort`	15	&H0000000F
`Integer`, `UInteger`	31	&H0000001F
`Long`, `ULong`	63	&H0000003F

If `amount` is zero, the value of `result` is identical to the value of `pattern`. If `amount` is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

Overloading

The `>>` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `>>` operator to perform arithmetic right shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim pattern As Short = 2560
' The bit pattern is 0000 1010 0000 0000.
Dim result1, result2, result3, result4, result5 As Short
result1 = pattern >> 0
result2 = pattern >> 4
result3 = pattern >> 10
result4 = pattern >> 18
result5 = pattern >> -1
```

The results of the preceding example are as follows:

- `result1` is 2560 (0000 1010 0000 0000).
- `result2` is 160 (0000 0000 1010 0000).
- `result3` is 2 (0000 0000 0000 0010).
- `result4` is 640 (0000 0010 1000 0000).
- `result5` is 0 (shifted 15 places to the right).

The shift amount for `result4` is calculated as 18 AND 15, which equals 2.

The following example shows arithmetic shifts on a negative value.

```
Dim negPattern As Short = -8192
' The bit pattern is 1110 0000 0000 0000.
Dim negResult1, negResult2 As Short
negResult1 = negPattern >> 4
negResult2 = negPattern >> 13
```

The results of the preceding example are as follows:

- `negresult1` is -512 (1111 1110 0000 0000).
- `negresult2` is -1 (the sign bit is propagated).

See also

- Bit Shift Operators
- Assignment Operators
- >>= Operator
- Operator Precedence in Visual Basic
- Operators Listed by Functionality

- Arithmetic Operators in Visual Basic

>>= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Performs an arithmetic right shift on the value of a variable or property and assigns the result back to the variable or property.

Syntax

```
variableorproperty >>= amount
```

Parts

`variableorproperty`
Required. Variable or property of an integral type (`SByte` , `Byte` , `Short` , `UShort` , `Integer` , `UInteger` , `Long` , or `ULong`).

`amount`
Required. Numeric expression of a data type that widens to `Integer` .

Remarks

The element on the left side of the `>>=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `>>=` operator first performs an arithmetic right shift on the value of the variable or property. The operator then assigns the result of that operation back to the variable or property.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost bit is propagated into the bit positions vacated at the left. This means that if `variableorproperty` has a negative value, the vacated positions are set to one. If `variableorproperty` is positive, or if its data type is an unsigned type, the vacated positions are set to zero.

Overloading

The >> Operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `>>` operator affects the behavior of the `>>=` operator. If your code uses `>>=` on a class or structure that overloads `>>` , be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `>>=` operator to shift the bit pattern of an `Integer` variable right by the specified amount and assign the result to the variable.

```
Dim var As Integer = 10
Dim shift As Integer = 2
var >>= shift
' The value of var is now 2 (two bits were lost off the right end).
```

See also

- >> Operator
- Assignment Operators
- Bit Shift Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

/ Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Divides two numbers and returns a floating-point result.

Syntax

```
expression1 / expression2
```

Parts

`expression1`
Required. Any numeric expression.

`expression2`
Required. Any numeric expression.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Result

The result is the full quotient of `expression1` divided by `expression2`, including any remainder.

The \ Operator (Visual Basic) returns the integer quotient, which drops the remainder.

Remarks

The data type of the result depends on the types of the operands. The following table shows how the data type of the result is determined.

OPERAND DATA TYPES	RESULT DATA TYPE
Both expressions are integral data types (SByte, Byte, Short, UShort, Integer, UInteger, Long, ULong)	`Double`
One expression is a Single data type and the other is not a Double	`Single`
One expression is a Decimal data type and the other is not a Single or a Double	`Decimal`
Either expression is a Double data type	`Double`

Before division is performed, any integral numeric expressions are widened to `Double`. If you assign the result to an integral data type, Visual Basic attempts to convert the result from `Double` to that type. This can throw an exception if the result does not fit in that type. In particular, see "Attempted Division by Zero" on this Help page.

If `expression1` or `expression2` evaluates to Nothing, it is treated as zero.

Attempted Division by Zero

If `expression2` evaluates to zero, the `/` operator behaves differently for different operand data types. The following table shows the possible behaviors.

OPERAND DATA TYPES	BEHAVIOR IF `EXPRESSION2` IS ZERO
Floating-point (`Single` or `Double`)	Returns infinity (PositiveInfinity or NegativeInfinity), or NaN (not a number) if `expression1` is also zero
`Decimal`	Throws DivideByZeroException
Integral (signed or unsigned)	Attempted conversion back to integral type throws OverflowException because integral types cannot accept PositiveInfinity, NegativeInfinity, or NaN

> **NOTE**
> The `/` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

This example uses the `/` operator to perform floating-point division. The result is the quotient of the two operands.

```
Dim resultValue As Double
resultValue = 10 / 4
resultValue = 10 / 3
```

The expressions in the preceding example return values of 2.5 and 3.333333. Note that the result is always floating-point (`Double`), even though both operands are integer constants.

See also

- /= Operator (Visual Basic)
- \ Operator (Visual Basic)
- Data Types of Operator Results
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

/= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Divides the value of a variable or property by the value of an expression and assigns the floating-point result to the variable or property.

Syntax

```
variableorproperty /= expression
```

Parts

`variableorproperty`
Required. Any numeric variable or property.

`expression`
Required. Any numeric expression.

Remarks

The element on the left side of the `/=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `/=` operator first divides the value of the variable or property (on the left-hand side of the operator) by the value of the expression (on the right-hand side of the operator). The operator then assigns the floating-point result of that operation to the variable or property.

This statement assigns a `Double` value to the variable or property on the left. If `Option Strict` is `On`, `variableorproperty` must be a `Double`. If `Option Strict` is `Off`, Visual Basic performs an implicit conversion and assigns the resulting value to `variableorproperty`, with a possible error at run time. For more information, see Widening and Narrowing Conversions and Option Strict Statement.

Overloading

The / Operator (Visual Basic) can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `/` operator affects the behavior of the `/=` operator. If your code uses `/=` on a class or structure that overloads `/`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `/=` operator to divide one `Integer` variable by a second and assign the quotient to the first variable.

```
Dim var1 As Integer = 12
Dim var2 As Integer = 3
var1 /= var2
' The value of var1 is now 4.
```

See also

- / Operator (Visual Basic)
- \= Operator
- Assignment Operators
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

\ Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Divides two numbers and returns an integer result.

Syntax

```
expression1 \ expression2
```

Parts

`expression1`
Required. Any numeric expression.

`expression2`
Required. Any numeric expression.

Supported Types

All numeric types, including the unsigned and floating-point types and `Decimal`.

Result

The result is the integer quotient of `expression1` divided by `expression2`, which discards any remainder and retains only the integer portion. This is known as *truncation*.

The result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Integer Arithmetic" tables in Data Types of Operator Results.

The / Operator (Visual Basic) returns the full quotient, which retains the remainder in the fractional portion.

Remarks

Before performing the division, Visual Basic attempts to convert any floating-point numeric expression to `Long`. If `Option Strict` is `On`, a compiler error occurs. If `Option Strict` is `Off`, an OverflowException is possible if the value is outside the range of the Long Data Type. The conversion to `Long` is also subject to *banker's rounding*. For more information, see "Fractional Parts" in Type Conversion Functions.

If `expression1` or `expression2` evaluates to Nothing, it is treated as zero.

Attempted Division by Zero

If `expression2` evaluates to zero, the `\` operator throws a DivideByZeroException exception. This is true for all numeric data types of the operands.

NOTE

The `\` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `\` operator to perform integer division. The result is an integer that represents the integer quotient of the two operands, with the remainder discarded.

```
Dim resultValue As Integer
resultValue = 11 \ 4
resultValue = 9 \ 3
resultValue = 100 \ 3
resultValue = 67 \ -3
```

The expressions in the preceding example return values of 2, 3, 33, and -22, respectively.

See also

- \= Operator
- / Operator (Visual Basic)
- Option Strict Statement
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

\= Operator

4/2/2019 • 2 minutes to read • Edit Online

Divides the value of a variable or property by the value of an expression and assigns the integer result to the variable or property.

Syntax

```
variableorproperty \= expression
```

Parts

`variableorproperty`
Required. Any numeric variable or property.

`expression`
Required. Any numeric expression.

Remarks

The element on the left side of the `\=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `\=` operator divides the value of a variable or property on its left by the value on its right, and assigns the integer result to the variable or property on its left

For further information on integer division, see \ Operator (Visual Basic).

Overloading

The `\` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `\` operator affects the behavior of the `\=` operator. If your code uses `\=` on a class or structure that overloads `\`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `\=` operator to divide one `Integer` variable by a second and assign the integer result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 \= var2
' The value of var1 is now 3.
```

See also

- \ Operator (Visual Basic)
- /= Operator (Visual Basic)

- Assignment Operators
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

^ Operator (Visual Basic)

3/8/2019 • 2 minutes to read • Edit Online

Raises a number to the power of another number.

Syntax

```
number ^ exponent
```

Parts

`number`
Required. Any numeric expression.

`exponent`
Required. Any numeric expression.

Result

The result is `number` raised to the power of `exponent`, always as a `Double` value.

Supported Types

`Double`. Operands of any different type are converted to `Double`.

Remarks

Visual Basic always performs exponentiation in the Double Data Type.

The value of `exponent` can be fractional, negative, or both.

When more than one exponentiation is performed in a single expression, the `^` operator is evaluated as it is encountered from left to right.

> **NOTE**
> The `^` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `^` operator to raise a number to the power of an exponent. The result is the first operand raised to the power of the second.

```
Dim exp1, exp2, exp3, exp4, exp5, exp6 As Double
exp1 = 2 ^ 2
exp2 = 3 ^ 3 ^ 3
exp3 = (-5) ^ 3
exp4 = (-5) ^ 4
exp5 = 8 ^ (1.0 / 3.0)
exp6 = 8 ^ (-1.0 / 3.0)
```

The preceding example produces the following results:

`exp1` is set to 4 (2 squared).

`exp2` is set to 19683 (3 cubed, then that value cubed).

`exp3` is set to -125 (-5 cubed).

`exp4` is set to 625 (-5 to the fourth power).

`exp5` is set to 2 (cube root of 8).

`exp6` is set to 0.5 (1.0 divided by the cube root of 8).

Note the importance of the parentheses in the expressions in the preceding example. Because of *operator precedence*, Visual Basic normally performs the `^` operator before any others, even the unary `-` operator. If `exp4` and `exp6` had been calculated without parentheses, they would have produced the following results:

`exp4 = -5 ^ 4` would be calculated as –(5 to the fourth power), which would result in -625.

`exp6 = 8 ^ -1.0 / 3.0` would be calculated as (8 to the –1 power, or 0.125) divided by 3.0, which would result in 0.041666666666666666666666666666667.

See also

- ^= Operator
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Arithmetic Operators in Visual Basic

^= Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Raises the value of a variable or property to the power of an expression and assigns the result back to the variable or property.

Syntax

```
variableorproperty ^= expression
```

Parts

`variableorproperty`
Required. Any numeric variable or property.

`expression`
Required. Any numeric expression.

Remarks

The element on the left side of the `^=` operator can be a simple scalar variable, a property, or an element of an array. The variable or property cannot be ReadOnly.

The `^=` operator first raises the value of the variable or property (on the left-hand side of the operator) to the power of the value of the expression (on the right-hand side of the operator). The operator then assigns the result of that operation back to the variable or property.

Visual Basic always performs exponentiation in the Double Data Type. Operands of any different type are converted to `Double`, and the result is always `Double`.

The value of `expression` can be fractional, negative, or both.

Overloading

The ^ Operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. Overloading the `^` operator affects the behavior of the `^=` operator. If your code uses `^=` on a class or structure that overloads `^`, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `^=` operator to raise the value of one `Integer` variable to the power of a second variable and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 ^= var2
' The value of var1 is now 1000.
```

See also

- ^ Operator
- Assignment Operators
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Statements

?. and ?() null-conditional operators (Visual Basic)

5/6/2019 • 2 minutes to read • Edit Online

Tests the value of the left-hand operand for null (`Nothing`) before performing a member access (`?.`) or index (`?()`) operation; returns `Nothing` if the left-hand operand evaluates to `Nothing`. Note that in expressions that ordinarily return value types, the null-conditional operator returns a Nullable<T>.

These operators help you write less code to handle null checks, especially when descending into data structures. For example:

```
' Nothing if customers is Nothing
Dim length As Integer? = customers?.Length

' Nothing if customers is Nothing
Dim first As Customer = customers?(0)

' Nothing if customers, the first customer, or Orders is Nothing
Dim count As Integer? = customers?(0)?.Orders?.Count()
```

For comparison, the alternative code for the first of these expressions without a null-conditional operator is:

```
Dim length As Integer
If customers IsNot Nothing Then
   length = customers.Length
End If
```

Sometimes you need to take an action on an object that may be null, based on the value of a Boolean member on that object (like the Boolean property `IsAllowedFreeShipping` in the following example):

```
  Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

  If customer IsNot Nothing AndAlso customer.IsAllowedFreeShipping Then
   ApplyFreeShippingToOrders(customer)
  End If
```

You can shorten your code and avoid manually checking for null by using the null-conditional operator as follows:

```
 Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

 If customer?.IsAllowedFreeShipping Then ApplyFreeShippingToOrders(customer)
```

The null-conditional operators are short-circuiting. If one operation in a chain of conditional member access and index operations returns `Nothing`, the rest of the chain's execution stops. In the following example, `C(E)` isn't evaluated if `A`, `B`, or `C` evaluates to `Nothing`.

```
A?.B?.C?(E);
```

Another use for null-conditional member access is to invoke delegates in a thread-safe way with much less code. The following example defines two types, a `NewsBroadcaster` and a `NewsReceiver`. News items are sent to the receiver by the `NewsBroadcaster.SendNews` delegate.

```
Public Module NewsBroadcaster
   Dim SendNews As Action(Of String)

   Public Sub Main()
      Dim rec As New NewsReceiver()
      Dim rec2 As New NewsReceiver()
      SendNews?.Invoke("Just in: A newsworthy item...")
   End Sub

   Public Sub Register(client As Action(Of String))
      SendNews = SendNews.Combine({SendNews, client})
   End Sub
End Module

Public Class NewsReceiver
   Public Sub New()
      NewsBroadcaster.Register(AddressOf Me.DisplayNews)
   End Sub

   Public Sub DisplayNews(newsItem As String)
      Console.WriteLine(newsItem)
   End Sub
End Class
```

If there are no elements in the `SendNews` invocation list, the `SendNews` delegate throws a NullReferenceException. Before null conditional operators, code like the following ensured that the delegate invocation list was not `Nothing`:

```
SendNews = SendNews.Combine({SendNews, client})
If SendNews IsNot Nothing Then
   SendNews("Just in...")
End If
```

The new way is much simpler:

```
SendNews = SendNews.Combine({SendNews, client})
SendNews?.Invoke("Just in...")
```

The new way is thread-safe because the compiler generates code to evaluate `SendNews` one time only, keeping the result in a temporary variable. You need to explicitly call the `Invoke` method because there is no null-conditional delegate invocation syntax `SendNews?(String)`.

See also

- Operators (Visual Basic)
- Visual Basic Programming Guide
- Visual Basic Language Reference

?. and ?() null-conditional operators (Visual Basic)

5/6/2019 • 2 minutes to read • Edit Online

Tests the value of the left-hand operand for null (`Nothing`) before performing a member access (`?.`) or index (`?()`) operation; returns `Nothing` if the left-hand operand evaluates to `Nothing`. Note that in expressions that ordinarily return value types, the null-conditional operator returns a Nullable<T>.

These operators help you write less code to handle null checks, especially when descending into data structures. For example:

```
' Nothing if customers is Nothing
Dim length As Integer? = customers?.Length

' Nothing if customers is Nothing
Dim first As Customer = customers?(0)

' Nothing if customers, the first customer, or Orders is Nothing
Dim count As Integer? = customers?(0)?.Orders?.Count()
```

For comparison, the alternative code for the first of these expressions without a null-conditional operator is:

```
Dim length As Integer
If customers IsNot Nothing Then
   length = customers.Length
End If
```

Sometimes you need to take an action on an object that may be null, based on the value of a Boolean member on that object (like the Boolean property `IsAllowedFreeShipping` in the following example):

```
  Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

  If customer IsNot Nothing AndAlso customer.IsAllowedFreeShipping Then
   ApplyFreeShippingToOrders(customer)
  End If
```

You can shorten your code and avoid manually checking for null by using the null-conditional operator as follows:

```
 Dim customer = FindCustomerByID(123) 'customer will be Nothing if not found.

 If customer?.IsAllowedFreeShipping Then ApplyFreeShippingToOrders(customer)
```

The null-conditional operators are short-circuiting. If one operation in a chain of conditional member access and index operations returns `Nothing`, the rest of the chain's execution stops. In the following example, `C(E)` isn't evaluated if `A`, `B`, or `C` evaluates to `Nothing`.

```
A?.B?.C?(E);
```

Another use for null-conditional member access is to invoke delegates in a thread-safe way with much less code. The following example defines two types, a `NewsBroadcaster` and a `NewsReceiver`. News items are sent to the receiver by the `NewsBroadcaster.SendNews` delegate.

```
Public Module NewsBroadcaster
   Dim SendNews As Action(Of String)

   Public Sub Main()
      Dim rec As New NewsReceiver()
      Dim rec2 As New NewsReceiver()
      SendNews?.Invoke("Just in: A newsworthy item...")
   End Sub

   Public Sub Register(client As Action(Of String))
      SendNews = SendNews.Combine({SendNews, client})
   End Sub
End Module

Public Class NewsReceiver
   Public Sub New()
      NewsBroadcaster.Register(AddressOf Me.DisplayNews)
   End Sub

   Public Sub DisplayNews(newsItem As String)
      Console.WriteLine(newsItem)
   End Sub
End Class
```

If there are no elements in the `SendNews` invocation list, the `SendNews` delegate throws a NullReferenceException. Before null conditional operators, code like the following ensured that the delegate invocation list was not `Nothing`:

```
SendNews = SendNews.Combine({SendNews, client})
If SendNews IsNot Nothing Then
   SendNews("Just in...")
End If
```

The new way is much simpler:

```
SendNews = SendNews.Combine({SendNews, client})
SendNews?.Invoke("Just in...")
```

The new way is thread-safe because the compiler generates code to evaluate `SendNews` one time only, keeping the result in a temporary variable. You need to explicitly call the `Invoke` method because there is no null-conditional delegate invocation syntax `SendNews?(String)`.

See also

- Operators (Visual Basic)
- Visual Basic Programming Guide
- Visual Basic Language Reference

AddressOf Operator (Visual Basic)

7/10/2019 • 2 minutes to read • Edit Online

Creates a delegate instance that references the specific procedure.

Syntax

```
AddressOf procedurename
```

Parts

`procedurename`
Required. Specifies the procedure to be referenced by the newly created delegate.

Remarks

The `AddressOf` operator creates a delegate that points to the sub or function specified by `procedurename`. When the specified procedure is an instance method then the delegate refers to both the instance and the method. Then, when the delegate is invoked the specified method of the specified instance is called.

The `AddressOf` operator can be used as the operand of a delegate constructor or it can be used in a context in which the type of the delegate can be determined by the compiler.

Example

This example uses the `AddressOf` operator to designate a delegate to handle the `Click` event of a button.

```
' Add the following line to Sub Form1_Load().
AddHandler Button1.Click, AddressOf Button1_Click
```

Example

The following example uses the `AddressOf` operator to designate the startup function for a thread.

```
Public Sub CountSheep()
    Dim i As Integer = 1 ' Sheep do not count from 0.
    Do While (True) ' Endless loop.
        Console.WriteLine("Sheep " & i & " Baah")
        i = i + 1
        System.Threading.Thread.Sleep(1000) 'Wait 1 second.
    Loop
End Sub

Sub UseThread()
    Dim t As New System.Threading.Thread(AddressOf CountSheep)
    t.Start()
End Sub
```

See also

- Declare Statement
- Function Statement
- Sub Statement
- Delegates

And Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Performs a logical conjunction on two `Boolean` expressions, or a bitwise conjunction on two numeric expressions.

Syntax

```
result = expression1 And expression2
```

Parts

`result`
Required. Any `Boolean` or numeric expression. For Boolean comparison, `result` is the logical conjunction of two `Boolean` values. For bitwise operations, `result` is a numeric value representing the bitwise conjunction of two numeric bit patterns.

`expression1`
Required. Any `Boolean` or numeric expression.

`expression2`
Required. Any `Boolean` or numeric expression.

Remarks

For Boolean comparison, `result` is `True` if and only if both `expression1` and `expression2` evaluate to `True`. The following table illustrates how `result` is determined.

IF `EXPRESSION1` IS	AND `EXPRESSION2` IS	THE VALUE OF `RESULT` IS
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

NOTE
In a Boolean comparison, the `And` operator always evaluates both expressions, which could include making procedure calls. The AndAlso Operator performs *short-circuiting*, which means that if `expression1` is `False`, then `expression2` is not evaluated.

When applied to numeric values, the `And` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN `EXPRESSION1` IS	AND BIT IN `EXPRESSION2` IS	THE BIT IN `RESULT` IS
1	1	1
1	0	0
0	1	0
0	0	0

> **NOTE**
> Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate results.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (–1 for `True` and 0 for `False`) and performs a bitwise operation.

For a Boolean comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in Data Types of Operator Results.

> **NOTE**
> The `And` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `And` operator to perform a logical conjunction on two expressions. The result is a `Boolean` value that represents whether both of the expressions are `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck As Boolean
firstCheck = a > b And b > c
secondCheck = b > a And b > c
```

The preceding example produces results of `True` and `False`, respectively.

Example

The following example uses the `And` operator to perform logical conjunction on the individual bits of two numeric expressions. The bit in the result pattern is set if the corresponding bits in the operands are both set to 1.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a And b)
secondPattern = (a And c)
thirdPattern = (b And c)
```

The preceding example produces results of 8, 2, and 0, respectively.

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- AndAlso Operator
- Logical and Bitwise Operators in Visual Basic

AndAlso Operator (Visual Basic)

7/12/2019 • 2 minutes to read • Edit Online

Performs short-circuiting logical conjunction on two expressions.

Syntax

```
result = expression1 AndAlso expression2
```

Parts

TERM	DEFINITION
`result`	Required. Any `Boolean` expression. The result is the `Boolean` result of comparison of the two expressions.
`expression1`	Required. Any `Boolean` expression.
`expression2`	Required. Any `Boolean` expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the second expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If both expressions evaluate to `True`, `result` is `True`. The following table illustrates how `result` is determined.

IF `EXPRESSION1` IS	AND `EXPRESSION2` IS	THE VALUE OF `RESULT` IS
`True`	`True`	`True`
`True`	`False`	`False`
`False`	(not evaluated)	`False`

Data Types

The `AndAlso` operator is defined only for the Boolean Data Type. Visual Basic converts each operand as necessary to `Boolean` before evaluating the expression. If you assign the result to a numeric type, Visual Basic converts it from `Boolean` to that type such that `False` becomes `0` and `True` becomes `-1`. For more information, see Boolean Type Conversions

Overloading

The And Operator and the IsFalse Operator can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. Overloading the `And` and `IsFalse` operators affects the behavior of the `AndAlso` operator. If your code uses `AndAlso` on a class or structure that overloads `And` and `IsFalse`, be sure you understand their redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `AndAlso` operator to perform a logical conjunction on two expressions. The result is a `Boolean` value that represents whether the entire conjoined expression is true. If the first expression is `False`, the second is not evaluated.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b AndAlso b > c
secondCheck = b > a AndAlso b > c
thirdCheck = a > b AndAlso c > b
```

The preceding example produces results of `True`, `False`, and `False`, respectively. In the calculation of `secondCheck`, the second expression is not evaluated because the first is already `False`. However, the second expression is evaluated in the calculation of `thirdCheck`.

Example

The following example shows a `Function` procedure that searches for a given value among the elements of an array. If the array is empty, or if the array length has been exceeded, the `While` statement does not test the array element against the search value.

```
Public Function findValue(ByVal arr() As Double,
    ByVal searchValue As Double) As Double
    Dim i As Integer = 0
    While i <= UBound(arr) AndAlso arr(i) <> searchValue
        ' If i is greater than UBound(arr), searchValue is not checked.
        i += 1
    End While
    If i > UBound(arr) Then i = -1
    Return i
End Function
```

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- And Operator
- IsFalse Operator
- Logical and Bitwise Operators in Visual Basic

Await Operator (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

You apply the `Await` operator to an operand in an asynchronous method or lambda expression to suspend execution of the method until the awaited task completes. The task represents ongoing work.

The method in which `Await` is used must have an Async modifier. Such a method, defined by using the `Async` modifier, and usually containing one or more `Await` expressions, is referred to as an *async method*.

> **NOTE**
>
> The `Async` and `Await` keywords were introduced in Visual Studio 2012. For an introduction to async programming, see Asynchronous Programming with Async and Await.

Typically, the task to which you apply the `Await` operator is the return value from a call to a method that implements the Task-Based Asynchronous Pattern, that is, a Task or a Task<TResult>.

In the following code, the HttpClient method GetByteArrayAsync returns `getContentsTask`, a `Task(Of Byte())`. The task is a promise to produce the actual byte array when the operation is complete. The `Await` operator is applied to `getContentsTask` to suspend execution in `SumPageSizesAsync` until `getContentsTask` is complete. In the meantime, control is returned to the caller of `SumPageSizesAsync`. When `getContentsTask` is finished, the `Await` expression evaluates to a byte array.

```
Private Async Function SumPageSizesAsync() As Task

    ' To use the HttpClient type in desktop apps, you must include a using directive and add a
    ' reference for the System.Net.Http namespace.
    Dim client As HttpClient = New HttpClient()
    ' . . .
    Dim getContentsTask As Task(Of Byte()) = client.GetByteArrayAsync(url)
    Dim urlContents As Byte() = Await getContentsTask

    ' Equivalently, now that you see how it works, you can write the same thing in a single line.
    'Dim urlContents As Byte() = Await client.GetByteArrayAsync(url)
    ' . . .
End Function
```

> **IMPORTANT**
>
> For the complete example, see Walkthrough: Accessing the Web by Using Async and Await. You can download the sample from Developer Code Samples on the Microsoft website. The example is in the AsyncWalkthrough_HttpClient project.

If `Await` is applied to the result of a method call that returns a `Task(Of TResult)`, the type of the `Await` expression is TResult. If `Await` is applied to the result of a method call that returns a `Task`, the `Await` expression doesn't return a value. The following example illustrates the difference.

```
' Await used with a method that returns a Task(Of TResult).
Dim result As TResult = Await AsyncMethodThatReturnsTaskTResult()

' Await used with a method that returns a Task.
Await AsyncMethodThatReturnsTask()
```

An `Await` expression or statement does not block the thread on which it is executing. Instead, it causes the compiler to sign up the rest of the async method, after the `Await` expression, as a continuation on the awaited task. Control then returns to the caller of the async method. When the task completes, it invokes its continuation, and execution of the async method resumes where it left off.

An `Await` expression can occur only in the body of an immediately enclosing method or lambda expression that is marked by an `Async` modifier. The term *Await* serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier. Within the async method or lambda expression, an `Await` expression cannot occur in a query expression, in the `catch` or `finally` block of a Try...Catch...Finally statement, in the loop control variable expression of a `For` or `For Each` loop, or in the body of a SyncLock statement.

Exceptions

Most async methods return a Task or Task<TResult>. The properties of the returned task carry information about its status and history, such as whether the task is complete, whether the async method caused an exception or was canceled, and what the final result is. The `Await` operator accesses those properties.

If you await a task-returning async method that causes an exception, the `Await` operator rethrows the exception.

If you await a task-returning async method that is canceled, the `Await` operator rethrows an OperationCanceledException.

A single task that is in a faulted state can reflect multiple exceptions. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, the await operation rethrows only one of the exceptions. However, you can't predict which of the exceptions is rethrown.

For examples of error handling in async methods, see Try...Catch...Finally Statement.

Example

The following Windows Forms example illustrates the use of `Await` in an async method, `WaitAsynchronouslyAsync`. Contrast the behavior of that method with the behavior of `WaitSynchronously`. Without an `Await` operator, `WaitSynchronously` runs synchronously despite the use of the `Async` modifier in its definition and a call to Thread.Sleep in its body.

```
Private Async Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    ' Call the method that runs asynchronously.
    Dim result As String = Await WaitAsynchronouslyAsync()

    ' Call the method that runs synchronously.
    'Dim result As String = Await WaitSynchronously()

    ' Display the result.
    TextBox1.Text &= result
End Sub

' The following method runs asynchronously. The UI thread is not
' blocked during the delay. You can move or resize the Form1 window
' while Task.Delay is running.
Public Async Function WaitAsynchronouslyAsync() As Task(Of String)
    Await Task.Delay(10000)
    Return "Finished"
End Function

' The following method runs synchronously, despite the use of Async.
' You cannot move or resize the Form1 window while Thread.Sleep
' is running because the UI thread is blocked.
Public Async Function WaitSynchronously() As Task(Of String)
    ' Import System.Threading for the Sleep method.
    Thread.Sleep(10000)
    Return "Finished"
End Function
```

See also

- Asynchronous Programming with Async and Await
- Walkthrough: Accessing the Web by Using Async and Await
- Async

DirectCast Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Introduces a type conversion operation based on inheritance or implementation.

Remarks

`DirectCast` does not use the Visual Basic run-time helper routines for conversion, so it can provide somewhat better performance than `CType` when converting to and from data type `Object`.

You use the `DirectCast` keyword similar to the way you use the CType Function and the TryCast Operator keyword. You supply an expression as the first argument and a type to convert it to as the second argument. `DirectCast` requires an inheritance or implementation relationship between the data types of the two arguments. This means that one type must inherit from or implement the other.

Errors and Failures

`DirectCast` generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, the runtime throws an InvalidCastException error.

Conversion Keywords

A comparison of the type conversion keywords is as follows.

KEYWORD	DATA TYPES	ARGUMENT RELATIONSHIP	RUN-TIME FAILURE
CType Function	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws InvalidCastException
`DirectCast`	Any data types	One type must inherit from or implement the other type	Throws InvalidCastException
TryCast Operator	Reference types only	One type must inherit from or implement the other type	Returns Nothing

Example

The following example demonstrates two uses of `DirectCast`, one that fails at run time and one that succeeds.

```
Dim q As Object = 2.37
Dim i As Integer = CType(q, Integer)
' The following conversion fails at run time
Dim j As Integer = DirectCast(q, Integer)
Dim f As New System.Windows.Forms.Form
Dim c As System.Windows.Forms.Control
' The following conversion succeeds.
c = DirectCast(f, System.Windows.Forms.Control)
```

In the preceding example, the run-time type of `q` is `Double`. `CType` succeeds because `Double` can be converted

to `Integer`. However, the first `DirectCast` fails at run time because the run-time type of `Double` has no inheritance relationship with `Integer`, even though a conversion exists. The second `DirectCast` succeeds because it converts from type Form to type Control, from which Form inherits.

See also

- Convert.ChangeType
- Widening and Narrowing Conversions
- Implicit and Explicit Conversions

Function Expression (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Declares the parameters and code that define a function lambda expression.

Syntax

```
Function ( [ parameterlist ] ) expression
- or -
Function ( [ parameterlist ] )
  [ statements ]
End Function
```

Parts

TERM	DEFINITION
`parameterlist`	Optional. A list of local variable names that represent the parameters of this procedure. The parentheses must be present even when the list is empty. See Parameter List.
`expression`	Required. A single expression. The type of the expression is the return type of the function.
`statements`	Required. A list of statements that returns a value by using the `Return` statement. (See Return Statement.) The type of the value returned is the return type of the function.

Remarks

A *lambda expression* is a function without a name that calculates and returns a value. You can use a lambda expression anywhere you can use a delegate type, except as an argument to `RemoveHandler`. For more information about delegates, and the use of lambda expressions with delegates, see Delegate Statement and Relaxed Delegate Conversion.

Lambda Expression Syntax

The syntax of a lambda expression resembles that of a standard function. The differences are as follows:

- A lambda expression does not have a name.
- Lambda expressions cannot have modifiers, such as `Overloads` or `Overrides`.
- Lambda expressions do not use an `As` clause to designate the return type of the function. Instead, the type is inferred from the value that the body of a single-line lambda expression evaluates to, or the return value of a multiline lambda expression. For example, if the body of a single-line lambda expression is `Where cust.City = "London"`, its return type is `Boolean`.
- The body of a single-line lambda expression must be an expression, not a statement. The body can consist of a call to a function procedure, but not a call to a sub procedure.

- Either all parameters must have specified data types or all must be inferred.
- Optional and Paramarray parameters are not permitted.
- Generic parameters are not permitted.

Example

The following examples show two ways to create simple lambda expressions. The first uses a `Dim` to provide a name for the function. To call the function, you send in a value for the parameter.

```
Dim add1 = Function(num As Integer) num + 1
```

```
' The following line prints 6.
Console.WriteLine(add1(5))
```

Example

Alternatively, you can declare and run the function at the same time.

```
Console.WriteLine((Function(num As Integer) num + 1)(5))
```

Example

Following is an example of a lambda expression that increments its argument and returns the value. The example shows both the single-line and multiline lambda expression syntax for a function. For more examples, see Lambda Expressions.

```
Dim increment1 = Function(x) x + 1
Dim increment2 = Function(x)
                     Return x + 2
                 End Function

' Write the value 2.
Console.WriteLine(increment1(1))

' Write the value 4.
Console.WriteLine(increment2(2))
```

Example

Lambda expressions underlie many of the query operators in Language-Integrated Query (LINQ), and can be used explicitly in method-based queries. The following example shows a typical LINQ query, followed by the translation of the query into method format.

```
Dim londonCusts = From cust In db.Customers
                      Where cust.City = "London"
                      Select cust

' This query is compiled to the following code:
Dim londonCusts = db.Customers.
                  Where(Function(cust) cust.City = "London").
                  Select(Function(cust) cust)
```

For more information about query methods, see Queries. For more information about standard query operators, see Standard Query Operators Overview.

See also

- Function Statement
- Lambda Expressions
- Operators and Expressions
- Statements
- Value Comparisons
- Boolean Expressions
- If Operator
- Relaxed Delegate Conversion

GetType Operator (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Returns a Type object for the specified type. The Type object provides information about the type such as its properties, methods, and events.

Syntax

```
GetType(typename)
```

Parameters

PARAMETER	DESCRIPTION
`typename`	The name of the type for which you desire information.

Remarks

The `GetType` operator returns the Type object for the specified `typename`. You can pass the name of any defined type in `typename`. This includes the following:

- Any Visual Basic data type, such as `Boolean` or `Date`.
- Any .NET Framework class, structure, module, or interface, such as System.ArgumentException or System.Double.
- Any class, structure, module, or interface defined by your application.
- Any array defined by your application.
- Any delegate defined by your application.
- Any enumeration defined by Visual Basic, the .NET Framework, or your application.

If you want to get the type object of an object variable, use the Type.GetType method.

The `GetType` operator can be useful in the following circumstances:

- You must access the metadata for a type at run time. The Type object supplies metadata such as type members and deployment information. You need this, for example, to reflect over an assembly. For more information, see System.Reflection.
- You want to compare two object references to see if they refer to instances of the same type. If they do, `GetType` returns references to the same Type object.

Example

The following examples show the `GetType` operator in use.

```
' The following statement returns the Type object for Integer.
MsgBox(GetType(Integer).ToString())
' The following statement returns the Type object for one-dimensional string arrays.
MsgBox(GetType(String()).ToString())
```

See also

- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Operators and Expressions

GetXmlNamespace Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Gets the XNamespace object that corresponds to the specified XML namespace prefix.

Syntax

```
GetXmlNamespace(xmlNamespacePrefix)
```

Parts

`xmlNamespacePrefix`
Optional. The string that identifies the XML namespace prefix. If supplied, this string must be a valid XML identifier. For more information, see Names of Declared XML Elements and Attributes. If no prefix is specified, the default namespace is returned. If no default namespace is specified, the empty namespace is returned.

Return Value

The XNamespace object that corresponds to the XML namespace prefix.

Remarks

The `GetXmlNamespace` operator gets the XNamespace object that corresponds to the XML namespace prefix `xmlNamespacePrefix`.

You can use XML namespace prefixes directly in XML literals and XML axis properties. However, you must use the `GetXmlNamespace` operator to convert a namespace prefix to an XNamespace object before you can use it in your code. You can append an unqualified element name to an XNamespace object to get a fully qualified XName object, which many LINQ to XML methods require.

Example

The following example imports `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node that has the qualified name `ns:phone`. It then passes that child node to the `ShowName` subroutine, which constructs a qualified name by using the `GetXmlNamespace` operator. The `ShowName` subroutine then passes the qualified name to the Ancestors method to get the parent `ns:contact` node.

```
' Place Imports statements at the top of your program.
Imports <xmlns:ns="http://SomeNamespace">

Module GetXmlNamespaceSample

    Sub RunSample()

        ' Create test by using a global XML namespace prefix.

        Dim contact =
            <ns:contact>
                <ns:name>Patrick Hines</ns:name>
                <ns:phone ns:type="home">206-555-0144</ns:phone>
                <ns:phone ns:type="work">425-555-0145</ns:phone>
            </ns:contact>

        ShowName(contact.<ns:phone>(0))
    End Sub

    Sub ShowName(ByVal phone As XElement)
        Dim qualifiedName = GetXmlNamespace(ns) + "contact"
        Dim contact = phone.Ancestors(qualifiedName)(0)
        Console.WriteLine("Name: " & contact.<ns:name>.Value)
    End Sub

End Module
```

When you call `TestGetXmlNamespace.RunSample()`, it displays a message box that contains the following text:

```
Name: Patrick Hines
```

See also

- Imports Statement (XML Namespace)
- Accessing XML in Visual Basic

If Operator (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

Uses short-circuit evaluation to conditionally return one of two values. The `If` operator can be called with three arguments or with two arguments.

Syntax

```
If( [argument1,] argument2, argument3 )
```

If Operator Called with Three Arguments

When `If` is called by using three arguments, the first argument must evaluate to a value that can be cast as a `Boolean`. That `Boolean` value will determine which of the other two arguments is evaluated and returned. The following list applies only when the `If` operator is called by using three arguments.

Parts

TERM	DEFINITION
`argument1`	Required. `Boolean`. Determines which of the other arguments to evaluate and return.
`argument2`	Required. `Object`. Evaluated and returned if `argument1` evaluates to `True`.
`argument3`	Required. `Object`. Evaluated and returned if `argument1` evaluates to `False` or if `argument1` is a Nullable `Boolean` variable that evaluates to Nothing.

An `If` operator that is called with three arguments works like an `IIf` function except that it uses short-circuit evaluation. An `IIf` function always evaluates all three of its arguments, whereas an `If` operator that has three arguments evaluates only two of them. The first `If` argument is evaluated and the result is cast as a `Boolean` value, `True` or `False`. If the value is `True`, `argument2` is evaluated and its value is returned, but `argument3` is not evaluated. If the value of the `Boolean` expression is `False`, `argument3` is evaluated and its value is returned, but `argument2` is not evaluated. The following examples illustrate the use of `If` when three arguments are used:

```
' This statement prints TruePart, because the first argument is true.
Console.WriteLine(If(True, "TruePart", "FalsePart"))

' This statement prints FalsePart, because the first argument is false.
Console.WriteLine(If(False, "TruePart", "FalsePart"))

Dim number = 3
' With number set to 3, this statement prints Positive.
Console.WriteLine(If(number >= 0, "Positive", "Negative"))

number = -1
' With number set to -1, this statement prints Negative.
Console.WriteLine(If(number >= 0, "Positive", "Negative"))
```

The following example illustrates the value of short-circuit evaluation. The example shows two attempts to divide variable `number` by variable `divisor` except when `divisor` is zero. In that case, a 0 should be returned, and no attempt should be made to perform the division because a run-time error would result. Because the `If` expression uses short-circuit evaluation, it evaluates either the second or the third argument, depending on the value of the first argument. If the first argument is true, the divisor is not zero and it is safe to evaluate the second argument and perform the division. If the first argument is false, only the third argument is evaluated and a 0 is returned. Therefore, when the divisor is 0, no attempt is made to perform the division and no error results. However, because `IIf` does not use short-circuit evaluation, the second argument is evaluated even when the first argument is false. This causes a run-time divide-by-zero error.

```
number = 12

' When the divisor is not 0, both If and IIf return 4.
Dim divisor = 3
Console.WriteLine(If(divisor <> 0, number \ divisor, 0))
Console.WriteLine(IIf(divisor <> 0, number \ divisor, 0))

' When the divisor is 0, IIf causes a run-time error, but If does not.
divisor = 0
Console.WriteLine(If(divisor <> 0, number \ divisor, 0))
' Console.WriteLine(IIf(divisor <> 0, number \ divisor, 0))
```

If Operator Called with Two Arguments

The first argument to `If` can be omitted. This enables the operator to be called by using only two arguments. The following list applies only when the `If` operator is called with two arguments.

Parts

TERM	DEFINITION
`argument2`	Required. `Object`. Must be a reference or nullable type. Evaluated and returned when it evaluates to anything other than `Nothing`.
`argument3`	Required. `Object`. Evaluated and returned if `argument2` evaluates to `Nothing`.

When the `Boolean` argument is omitted, the first argument must be a reference or nullable type. If the first argument evaluates to `Nothing`, the value of the second argument is returned. In all other cases, the value of the first argument is returned. The following example illustrates how this evaluation works.

```
' Variable first is a nullable type.
Dim first? As Integer = 3
Dim second As Integer = 6

' Variable first <> Nothing, so its value, 3, is returned.
Console.WriteLine(If(first, second))

second = Nothing
' Variable first <> Nothing, so the value of first is returned again.
Console.WriteLine(If(first, second))

first = Nothing
second = 6
' Variable first = Nothing, so 6 is returned.
Console.WriteLine(If(first, second))
```

See also

- IIf
- Nullable Value Types
- Nothing

Is Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Compares two object reference variables.

Syntax

```
result = object1 Is object2
```

Parts

`result`
Required. Any `Boolean` value.

`object1`
Required. Any `Object` name.

`object2`
Required. Any `Object` name.

Remarks

The `Is` operator determines if two object references refer to the same object. However, it does not perform value comparisons. If `object1` and `object2` both refer to the exact same object instance, `result` is `True`; if they do not, `result` is `False`.

`Is` can also be used with the `TypeOf` keyword to make a `TypeOf` ... `Is` expression, which tests whether an object variable is compatible with a data type.

> **NOTE**
> The `Is` keyword is also used in the Select...Case Statement.

Example

The following example uses the `Is` operator to compare pairs of object references. The results are assigned to a `Boolean` value representing whether the two objects are identical.

```
Dim myObject As New Object
Dim otherObject As New Object
Dim yourObject, thisObject, thatObject As Object
Dim myCheck As Boolean
yourObject = myObject
thisObject = myObject
thatObject = otherObject
' The following statement sets myCheck to True.
myCheck = yourObject Is thisObject
' The following statement sets myCheck to False.
myCheck = thatObject Is thisObject
' The following statement sets myCheck to False.
myCheck = myObject Is thatObject
thatObject = myObject
' The following statement sets myCheck to True.
myCheck = thisObject Is thatObject
```

As the preceding example demonstrates, you can use the `Is` operator to test both early bound and late bound objects.

See also

- TypeOf Operator
- IsNot Operator
- Comparison Operators in Visual Basic
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Operators and Expressions

IsFalse Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Determines whether an expression is `False`.

You cannot call `IsFalse` explicitly in your code, but the Visual Basic compiler can use it to generate code from `AndAlso` clauses. If you define a class or structure and then use a variable of that type in an `AndAlso` clause, you must define `IsFalse` on that class or structure.

The compiler considers the `IsFalse` and `IsTrue` operators as a *matched pair*. This means that if you define one of them, you must also define the other one.

> **NOTE**
>
> The `IsFalse` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following code example defines the outline of a structure that includes definitions for the `IsFalse` and `IsTrue` operators.

```
Public Structure p
    Dim a As Double
    Public Shared Operator IsFalse(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of w.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of w.
        Return b
    End Operator
End Structure
```

See also

- IsTrue Operator
- How to: Define an Operator
- AndAlso Operator

IsNot Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Compares two object reference variables.

Syntax

```
result = object1 IsNot object2
```

Parts

`result`
Required. A `Boolean` value.

`object1`
Required. Any `Object` variable or expression.

`object2`
Required. Any `Object` variable or expression.

Remarks

The `IsNot` operator determines if two object references refer to different objects. However, it does not perform value comparisons. If `object1` and `object2` both refer to the exact same object instance, `result` is `False`; if they do not, `result` is `True`.

`IsNot` is the opposite of the `Is` operator. The advantage of `IsNot` is that you can avoid awkward syntax with `Not` and `Is`, which can be difficult to read.

You can use the `Is` and `IsNot` operators to test both early-bound and late-bound objects.

> **NOTE**
> The `IsNot` operator cannot be used to compare expressions returned from the `TypeOf` operator. Instead, you must use the `Not` and `Is` operators.

Example

The following code example uses both the `Is` operator and the `IsNot` operator to accomplish the same comparison.

```
Dim o1, o2 As New Object
If Not o1 Is o2 Then MsgBox("o1 and o2 do not refer to the same instance.")
If o1 IsNot o2 Then MsgBox("o1 and o2 do not refer to the same instance.")
```

See also

- Is Operator

- TypeOf Operator
- Operator Precedence in Visual Basic
- How to: Test Whether Two Objects Are the Same

IsTrue Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Determines whether an expression is `True`.

You cannot call `IsTrue` explicitly in your code, but the Visual Basic compiler can use it to generate code from `OrElse` clauses. If you define a class or structure and then use a variable of that type in an `OrElse` clause, you must define `IsTrue` on that class or structure.

The compiler considers the `IsTrue` and `IsFalse` operators as a *matched pair*. This means that if you define one of them, you must also define the other one.

Compiler Use of IsTrue

When you have defined a class or structure, you can use a variable of that type in a `For`, `If`, `Else If`, or `While` statement, or in a `When` clause. If you do this, the compiler requires an operator that converts your type into a `Boolean` value so it can test a condition. It searches for a suitable operator in the following order:

1. A widening conversion operator from your class or structure to `Boolean`.
2. A widening conversion operator from your class or structure to `Boolean?`.
3. The `IsTrue` operator on your class or structure.
4. A narrowing conversion to `Boolean?` that does not involve a conversion from `Boolean` to `Boolean?`.
5. A narrowing conversion operator from your class or structure to `Boolean`.

If you have not defined any conversion to `Boolean` or an `IsTrue` operator, the compiler signals an error.

> **NOTE**
>
> The `IsTrue` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following code example defines the outline of a structure that includes definitions for the `IsFalse` and `IsTrue` operators.

```
Public Structure p
    Dim a As Double
    Public Shared Operator IsFalse(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of w.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal w As p) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of w.
        Return b
    End Operator
End Structure
```

See also

- IsFalse Operator
- How to: Define an Operator
- OrElse Operator

Like Operator (Visual Basic)

4/2/2019 • 5 minutes to read • Edit Online

Compares a string against a pattern.

> **IMPORTANT**
> The `Like` operator is currently not supported in .NET Core and .NET Standard projects.

Syntax

```
result = string Like pattern
```

Parts

`result`
Required. Any `Boolean` variable. The result is a `Boolean` value indicating whether or not the `string` satisfies the `pattern`.

`string`
Required. Any `String` expression.

`pattern`
Required. Any `String` expression conforming to the pattern-matching conventions described in "Remarks."

Remarks

If the value in `string` satisfies the pattern contained in `pattern`, `result` is `True`. If the string does not satisfy the pattern, `result` is `False`. If both `string` and `pattern` are empty strings, the result is `True`.

Comparison Method

The behavior of the `Like` operator depends on the Option Compare Statement. The default string comparison method for each source file is `Option Compare Binary`.

Pattern Options

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to match each character in `string` against a specific character, a wildcard character, a character list, or a character range. The following table shows the characters allowed in `pattern` and what they match.

CHARACTERS IN `PATTERN`	MATCHES IN `STRING`
`?`	Any single character
`*`	Zero or more characters

CHARACTERS IN `pattern`	MATCHES IN `string`
`#`	Any single digit (0–9)
`[charlist]`	Any single character in `charlist`
`[!charlist]`	Any single character not in `charlist`

Character Lists

A group of one or more characters (`charlist`) enclosed in brackets (`[ ]`) can be used to match any single character in `string` and can include almost any character code, including digits.

An exclamation point (`!`) at the beginning of `charlist` means that a match is made if any character except the characters in `charlist` is found in `string` . When used outside brackets, the exclamation point matches itself.

Special Characters

To match the special characters left bracket (`[`), question mark (`?`), number sign (`#`), and asterisk (`*`), enclose them in brackets. The right bracket (`]`) cannot be used within a group to match itself, but it can be used outside a group as an individual character.

The character sequence `[]` is considered a zero-length string (`""`). However, it cannot be part of a character list enclosed in brackets. If you want to check whether a position in `string` contains one of a group of characters or no character at all, you can use `Like` twice. For an example, see How to: Match a String against a Pattern.

Character Ranges

By using a hyphen (`-`) to separate the lower and upper bounds of the range, `charlist` can specify a range of characters. For example, `[A-Z]` results in a match if the corresponding character position in `string` contains any character within the range `A` – `Z` , and `[!H-L]` results in a match if the corresponding character position contains any character outside the range `H` – `L` .

When you specify a range of characters, they must appear in ascending sort order, that is, from lowest to highest. Thus, `[A-Z]` is a valid pattern, but `[Z-A]` is not.

Multiple Character Ranges

To specify multiple ranges for the same character position, put them within the same brackets without delimiters. For example, `[A-CX-Z]` results in a match if the corresponding character position in `string` contains any character within either the range `A` – `C` or the range `X` – `Z` .

Usage of the Hyphen

A hyphen (`-`) can appear either at the beginning (after an exclamation point, if any) or at the end of `charlist` to match itself. In any other location, the hyphen identifies a range of characters delimited by the characters on either side of the hyphen.

Collating Sequence

The meaning of a specified range depends on the character ordering at run time, as determined by `Option Compare` and the locale setting of the system the code is running on. With `Option Compare Binary` , the range `[A-E]` matches `A` , `B` , `C` , `D` , and `E` . With `Option Compare Text` , `[A-E]` matches `A` , `a` , `À` , `à` , `B` , `b` , `C` , `c` , `D` , `d` , `E` , and `e` . The range does not match `Ê` or `ê` because accented characters collate after unaccented characters in the sort order.

Digraph Characters

In some languages, there are alphabetic characters that represent two separate characters. For example, several languages use the character `æ` to represent the characters `a` and `e` when they appear together. The `Like` operator recognizes that the single digraph character and the two individual characters are equivalent.

When a language that uses a digraph character is specified in the system locale settings, an occurrence of the single digraph character in either `pattern` or `string` matches the equivalent two-character sequence in the other string. Similarly, a digraph character in `pattern` enclosed in brackets (by itself, in a list, or in a range) matches the equivalent two-character sequence in `string`.

Overloading

The `Like` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

This example uses the `Like` operator to compare strings to various patterns. The results go into a `Boolean` variable indicating whether each string satisfies the pattern.

```
Dim testCheck As Boolean
' The following statement returns True (does "F" satisfy "F"?)
testCheck = "F" Like "F"
' The following statement returns False for Option Compare Binary
'    and True for Option Compare Text (does "F" satisfy "f"?)
testCheck = "F" Like "f"
' The following statement returns False (does "F" satisfy "FFF"?)
testCheck = "F" Like "FFF"
' The following statement returns True (does "aBBBa" have an "a" at the
'    beginning, an "a" at the end, and any number of characters in
'    between?)
testCheck = "aBBBa" Like "a*a"
' The following statement returns True (does "F" occur in the set of
'    characters from "A" through "Z"?)
testCheck = "F" Like "[A-Z]"
' The following statement returns False (does "F" NOT occur in the
'    set of characters from "A" through "Z"?)
testCheck = "F" Like "[!A-Z]"
' The following statement returns True (does "a2a" begin and end with
'    an "a" and have any single-digit number in between?)
testCheck = "a2a" Like "a#a"
' The following statement returns True (does "aM5b" begin with an "a",
'    followed by any character from the set "L" through "P", followed
'    by any single-digit number, and end with any character NOT in
'    the character set "c" through "e"?)
testCheck = "aM5b" Like "a[L-P]#[!c-e]"
' The following statement returns True (does "BAT123khg" begin with a
'    "B", followed by any single character, followed by a "T", and end
'    with zero or more characters of any type?)
testCheck = "BAT123khg" Like "B?T*"
' The following statement returns False (does "CAT123khg"?) begin with
'    a "B", followed by any single character, followed by a "T", and
'    end with zero or more characters of any type?)
testCheck = "CAT123khg" Like "B?T*"
```

See also

- InStr

- StrComp
- Comparison Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Option Compare Statement
- Operators and Expressions
- How to: Match a String against a Pattern

Mod operator (Visual Basic)

8/19/2019 • 3 minutes to read • Edit Online

Divides two numbers and returns only the remainder.

Syntax

```
result = number1 Mod number2
```

Parts

`result` Required. Any numeric variable or property.

`number1` Required. Any numeric expression.

`number2` Required. Any numeric expression.

Supported types

All numeric types. This includes the unsigned and floating-point types and `Decimal`.

Result

The result is the remainder after `number1` is divided by `number2`. For example, the expression `14 Mod 4` evaluates to 2.

> **NOTE**
>
> There is a difference between *remainder* and *modulus* in mathematics, with different results for negative numbers. The `Mod` operator in Visual Basic, the .NET Framework `op_Modulus` operator, and the underlying rem IL instruction all perform a remainder operation.

The result of a `Mod` operation retains the sign of the dividend, `number1`, and so it may be positive or negative. The result is always in the range (-`number2`, `number2`), exclusive. For example:

```
Public Module Example
   Public Sub Main()
      Console.WriteLine($" 8 Mod  3 = {8 Mod 3}")
      Console.WriteLine($"-8 Mod  3 = {-8 Mod 3}")
      Console.WriteLine($" 8 Mod -3 = {8 Mod -3}")
      Console.WriteLine($"-8 Mod -3 = {-8 Mod -3}")
   End Sub
End Module
' The example displays the following output:
'       8 Mod  3 = 2
'      -8 Mod  3 = -2
'       8 Mod -3 = 2
'      -8 Mod -3 = -2
```

Remarks

If either `number1` or `number2` is a floating-point value, the floating-point remainder of the division is returned. The data type of the result is the smallest data type that can hold all possible values that result from division with the data types of `number1` and `number2`.

If `number1` or `number2` evaluates to Nothing, it is treated as zero.

Related operators include the following:

- The \ Operator (Visual Basic) returns the integer quotient of a division. For example, the expression `14 \ 4` evaluates to 3.
- The / Operator (Visual Basic) returns the full quotient, including the remainder, as a floating-point number. For example, the expression `14 / 4` evaluates to 3.5.

Attempted division by zero

If `number2` evaluates to zero, the behavior of the `Mod` operator depends on the data type of the operands:

- An integral division throws a DivideByZeroException exception if `number2` cannot be determined in compile-time and generates a compile-time error `BC30542 Division by zero occurred while evaluating this expression` if `number2` is evaluated to zero at compile-time.
- A floating-point division returns Double.NaN.

Equivalent formula

The expression `a Mod b` is equivalent to either of the following formulas:

```
a - (b * (a \ b))
```

```
a - (b * Fix(a / b))
```

Floating-point imprecision

When you work with floating-point numbers, remember that they do not always have a precise decimal representation in memory. This can lead to unexpected results from certain operations, such as value comparison and the `Mod` operator. For more information, see Troubleshooting Data Types.

Overloading

The `Mod` operator can be *overloaded*, which means that a class or structure can redefine its behavior. If your code applies `Mod` to an instance of a class or structure that includes such an overload, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `Mod` operator to divide two numbers and return only the remainder. If either number is a floating-point number, the result is a floating-point number that represents the remainder.

```
Debug.WriteLine(10 Mod 5)
' Output: 0
Debug.WriteLine(10 Mod 3)
' Output: 1
Debug.WriteLine(-10 Mod 3)
' Output: -1
Debug.WriteLine(12 Mod 4.3)
' Output: 3.4
Debug.WriteLine(12.6 Mod 5)
' Output: 2.6
Debug.WriteLine(47.9 Mod 9.35)
' Output: 1.15
```

Example

The following example demonstrates the potential imprecision of floating-point operands. In the first statement, the operands are `Double`, and 0.2 is an infinitely repeating binary fraction with a stored value of 0.20000000000000001. In the second statement, the literal type character `D` forces both operands to `Decimal`, and 0.2 has a precise representation.

```
firstResult = 2.0 Mod 0.2
' Double operation returns 0.2, not 0.
secondResult = 2D Mod 0.2D
' Decimal operation returns 0.
```

See also

- Int
- Fix
- Arithmetic Operators
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Troubleshooting Data Types
- Arithmetic Operators in Visual Basic
- \ Operator (Visual Basic)

New Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Introduces a `New` clause to create a new object instance, specifies a constructor constraint on a type parameter, or identifies a `Sub` procedure as a class constructor.

Remarks

In a declaration or assignment statement, a `New` clause must specify a defined class from which the instance can be created. This means that the class must expose one or more constructors that the calling code can access.

You can use a `New` clause in a declaration statement or an assignment statement. When the statement runs, it calls the appropriate constructor of the specified class, passing any arguments you have supplied. The following example demonstrates this by creating instances of a `Customer` class that has two constructors, one that takes no parameters and one that takes a string parameter.

```
' For customer1, call the constructor that takes no arguments.
Dim customer1 As New Customer()

' For customer2, call the constructor that takes the name of the
' customer as an argument.
Dim customer2 As New Customer("Blue Yonder Airlines")

' For customer3, declare an instance of Customer in the first line
' and instantiate it in the second.
Dim customer3 As Customer
customer3 = New Customer()

' With Option Infer set to On, the following declaration declares
' and instantiates a new instance of Customer.
Dim customer4 = New Customer("Coho Winery")
```

Since arrays are classes, `New` can create a new array instance, as shown in the following examples.

```
Dim intArray1() As Integer
intArray1 = New Integer() {1, 2, 3, 4}

Dim intArray2() As Integer = {5, 6}

' The following example requires that Option Infer be set to On.
Dim intArray3() = New Integer() {6, 7, 8}
```

The common language runtime (CLR) throws an OutOfMemoryException error if there is insufficient memory to create the new instance.

> **NOTE**
>
> The `New` keyword is also used in type parameter lists to specify that the supplied type must expose an accessible parameterless constructor. For more information about type parameters and constraints, see Type List.

To create a constructor procedure for a class, set the name of a `Sub` procedure to the `New` keyword. For more information, see Object Lifetime: How Objects Are Created and Destroyed.

The `New` keyword can be used in these contexts:

Dim Statement

Of

Sub Statement

See also

- OutOfMemoryException
- Keywords
- Type List
- Generic Types in Visual Basic
- Object Lifetime: How Objects Are Created and Destroyed

Not Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Performs logical negation on a `Boolean` expression, or bitwise negation on a numeric expression.

Syntax

```
result = Not expression
```

Parts

`result`
Required. Any `Boolean` or numeric expression.

`expression`
Required. Any `Boolean` or numeric expression.

Remarks

For `Boolean` expressions, the following table illustrates how `result` is determined.

IF `EXPRESSION` IS	THE VALUE OF `RESULT` IS
`True`	`False`
`False`	`True`

For numeric expressions, the `Not` operator inverts the bit values of any numeric expression and sets the corresponding bit in `result` according to the following table.

IF BIT IN `EXPRESSION` IS	THE BIT IN `RESULT` IS
1	0
0	1

> **NOTE**
> Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

Data Types

For a Boolean negation, the data type of the result is `Boolean`. For a bitwise negation, the result data type is the same as that of `expression`. However, if expression is `Decimal`, the result is `Long`.

Overloading

The `Not` operator can be *overloaded*, which means that a class or structure can redefine its behavior when its operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `Not` operator to perform logical negation on a `Boolean` expression. The result is a `Boolean` value that represents the reverse of the value of the expression.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck As Boolean
firstCheck = Not (a > b)
secondCheck = Not (b > a)
```

The preceding example produces results of `False` and `True`, respectively.

Example

The following example uses the `Not` operator to perform logical negation of the individual bits of a numeric expression. The bit in the result pattern is set to the reverse of the corresponding bit in the operand pattern, including the sign bit.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (Not a)
secondPattern = (Not b)
thirdPattern = (Not c)
```

The preceding example produces results of −11, −9, and −7, respectively.

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Logical and Bitwise Operators in Visual Basic

Or Operator (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Performs a logical disjunction on two `Boolean` expressions, or a bitwise disjunction on two numeric expressions.

Syntax

```
result = expression1 Or expression2
```

Parts

`result`
Required. Any `Boolean` or numeric expression. For `Boolean` comparison, `result` is the inclusive logical disjunction of two `Boolean` values. For bitwise operations, `result` is a numeric value representing the inclusive bitwise disjunction of two numeric bit patterns.

`expression1`
Required. Any `Boolean` or numeric expression.

`expression2`
Required. Any `Boolean` or numeric expression.

Remarks

For `Boolean` comparison, `result` is `False` if and only if both `expression1` and `expression2` evaluate to `False`. The following table illustrates how `result` is determined.

IF `EXPRESSION1` IS	AND `EXPRESSION2` IS	THE VALUE OF `RESULT` IS
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

NOTE
In a `Boolean` comparison, the `Or` operator always evaluates both expressions, which could include making procedure calls. The OrElse Operator performs *short-circuiting*, which means that if `expression1` is `True`, then `expression2` is not evaluated.

For bitwise operations, the `Or` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN `EXPRESSION1` IS	AND BIT IN `EXPRESSION2` IS	THE BIT IN `RESULT` IS
1	1	1
1	0	1
0	1	1
0	0	0

> **NOTE**
> Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (−1 for `True` and 0 for `False`) and performs a bitwise operation.

For a `Boolean` comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in Data Types of Operator Results.

Overloading

The `Or` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, be sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `Or` operator to perform an inclusive logical disjunction on two expressions. The result is a `Boolean` value that represents whether either of the two expressions is `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b Or b > c
secondCheck = b > a Or b > c
thirdCheck = b > a Or c > b
```

The preceding example produces results of `True`, `True`, and `False`, respectively.

Example

The following example uses the `Or` operator to perform inclusive logical disjunction on the individual bits of two numeric expressions. The bit in the result pattern is set if either of the corresponding bits in the operands is set to 1.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a Or b)
secondPattern = (a Or c)
thirdPattern = (b Or c)
```

The preceding example produces results of 10, 14, and 14, respectively.

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- OrElse Operator
- Logical and Bitwise Operators in Visual Basic

OrElse Operator (Visual Basic)

7/12/2019 • 2 minutes to read • Edit Online

Performs short-circuiting inclusive logical disjunction on two expressions.

Syntax

```
result = expression1 OrElse expression2
```

Parts

`result`
Required. Any `Boolean` expression.

`expression1`
Required. Any `Boolean` expression.

`expression2`
Required. Any `Boolean` expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the second expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If either or both expressions evaluate to `True`, `result` is `True`. The following table illustrates how `result` is determined.

IF `EXPRESSION1` IS	AND `EXPRESSION2` IS	THE VALUE OF `RESULT` IS
`True`	(not evaluated)	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Data Types

The `OrElse` operator is defined only for the Boolean Data Type. Visual Basic converts each operand as necessary to `Boolean` before evaluating the expression. If you assign the result to a numeric type, Visual Basic converts it from `Boolean` to that type such that `False` becomes `0` and `True` becomes `-1`. For more information, see Boolean Type Conversions

Overloading

The Or Operator and the IsTrue Operator can be *overloaded*, which means that a class or structure can redefine their behavior when an operand has the type of that class or structure. Overloading the `Or` and `IsTrue` operators affects the behavior of the `OrElse` operator. If your code uses `OrElse` on a class or structure that overloads `Or` and `IsTrue`, be sure you understand their redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `OrElse` operator to perform logical disjunction on two expressions. The result is a `Boolean` value that represents whether either of the two expressions is true. If the first expression is `True`, the second is not evaluated.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b OrElse b > c
secondCheck = b > a OrElse b > c
thirdCheck = b > a OrElse c > b
```

The preceding example produces results of `True`, `True`, and `False` respectively. In the calculation of `firstCheck`, the second expression is not evaluated because the first is already `True`. However, the second expression is evaluated in the calculation of `secondCheck`.

Example

The following example shows an `If`...`Then` statement containing two procedure calls. If the first call returns `True`, the second procedure is not called. This could produce unexpected results if the second procedure performs important tasks that should always be performed when this section of the code runs.

```
If testFunction(5) = True OrElse otherFunction(4) = True Then
    ' If testFunction(5) is True, otherFunction(4) is not called.
    ' Insert code to be executed.
End If
```

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Or Operator
- IsTrue Operator
- Logical and Bitwise Operators in Visual Basic

Sub Expression (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Declares the parameters and code that define a subroutine lambda expression.

Syntax

```
Sub ( [ parameterlist ] ) statement
- or -
Sub ( [ parameterlist ] )
  [ statements ]
End Sub
```

Parts

TERM	DEFINITION
`parameterlist`	Optional. A list of local variable names that represent the parameters of the procedure. The parentheses must be present even when the list is empty. For more information, see Parameter List.
`statement`	Required. A single statement.
`statements`	Required. A list of statements.

Remarks

A *lambda expression* is a subroutine that does not have a name and that executes one or more statements. You can use a lambda expression anywhere that you can use a delegate type, except as an argument to `RemoveHandler`. For more information about delegates, and the use of lambda expressions with delegates, see Delegate Statement and Relaxed Delegate Conversion.

Lambda Expression Syntax

The syntax of a lambda expression resembles that of a standard subroutine. The differences are as follows:

- A lambda expression does not have a name.
- A lambda expression cannot have a modifier, such as `Overloads` or `Overrides`.
- The body of a single-line lambda expression must be a statement, not an expression. The body can consist of a call to a sub procedure, but not a call to a function procedure.
- In a lambda expression, either all parameters must have specified data types or all parameters must be inferred.
- Optional and `ParamArray` parameters are not permitted in lambda expressions.
- Generic parameters are not permitted in lambda expressions.

Example

Following is an example of a lambda expression that writes a value to the console. The example shows both the single-line and multiline lambda expression syntax for a subroutine. For more examples, see Lambda Expressions.

```
Dim writeline1 = Sub(x) Console.WriteLine(x)
Dim writeline2 = Sub(x)
                     Console.WriteLine(x)
                 End Sub

' Write "Hello".
writeline1("Hello")

' Write "World"
writeline2("World")
```

See also

- Sub Statement
- Lambda Expressions
- Operators and Expressions
- Statements
- Relaxed Delegate Conversion

TryCast Operator (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Introduces a type conversion operation that does not throw an exception.

Remarks

If an attempted conversion fails, `CType` and `DirectCast` both throw an InvalidCastException error. This can adversely affect the performance of your application. `TryCast` returns Nothing, so that instead of having to handle a possible exception, you need only test the returned result against `Nothing`.

You use the `TryCast` keyword the same way you use the CType Function and the DirectCast Operator keyword. You supply an expression as the first argument and a type to convert it to as the second argument. `TryCast` operates only on reference types, such as classes and interfaces. It requires an inheritance or implementation relationship between the two types. This means that one type must inherit from or implement the other.

Errors and Failures

`TryCast` generates a compiler error if it detects that no inheritance or implementation relationship exists. But the lack of a compiler error does not guarantee a successful conversion. If the desired conversion is narrowing, it could fail at run time. If this happens, `TryCast` returns Nothing.

Conversion Keywords

A comparison of the type conversion keywords is as follows.

KEYWORD	DATA TYPES	ARGUMENT RELATIONSHIP	RUN-TIME FAILURE
CType Function	Any data types	Widening or narrowing conversion must be defined between the two data types	Throws InvalidCastException
DirectCast Operator	Any data types	One type must inherit from or implement the other type	Throws InvalidCastException
`TryCast`	Reference types only	One type must inherit from or implement the other type	Returns Nothing

Example

The following example shows how to use `TryCast`.

```
Function PrintTypeCode(ByVal obj As Object) As String
    Dim objAsConvertible As IConvertible = TryCast(obj, IConvertible)
    If objAsConvertible Is Nothing Then
        Return obj.ToString() & " does not implement IConvertible"
    Else
        Return "Type code is " & objAsConvertible.GetTypeCode()
    End If
End Function
```

See also

- Widening and Narrowing Conversions
- Implicit and Explicit Conversions

TypeOf Operator (Visual Basic)

5/6/2019 • 2 minutes to read • Edit Online

Checks whether the runtime type of an expression's result is type-compatible with the specified type.

Syntax

```
result = TypeOf objectexpression Is typename
```

```
result = TypeOf objectexpression IsNot typename
```

Parts

`result`
Returned. A `Boolean` value.

`objectexpression`
Required. Any expression that evaluates to a reference type.

`typename`
Required. Any data type name.

Remarks

The `TypeOf` operator determines whether the run-time type of `objectexpression` is compatible with `typename`. The compatibility depends on the type category of `typename`. The following table shows how compatibility is determined.

TYPE CATEGORY OF `TYPENAME`	COMPATIBILITY CRITERION
Class	`objectexpression` is of type `typename` or inherits from `typename`
Structure	`objectexpression` is of type `typename`
Interface	`objectexpression` implements `typename` or inherits from a class that implements `typename`

If the run-time type of `objectexpression` satisfies the compatibility criterion, `result` is `True`. Otherwise, `result` is `False`. If `objectexpression` is null, then `TypeOf` ... `Is` returns `False`, and ... `IsNot` returns `True`.

`TypeOf` is always used with the `Is` keyword to construct a `TypeOf` ... `Is` expression, or with the `IsNot` keyword to construct a `TypeOf` ... `IsNot` expression.

Example

The following example uses `TypeOf` ... `Is` expressions to test the type compatibility of two object reference variables with various data types.

```
Dim refInteger As Object = 2
MsgBox("TypeOf Object[Integer] Is Integer? " & TypeOf refInteger Is Integer)
MsgBox("TypeOf Object[Integer] Is Double? " & TypeOf refInteger Is Double)
Dim refForm As Object = New System.Windows.Forms.Form
MsgBox("TypeOf Object[Form] Is Form? " & TypeOf refForm Is System.Windows.Forms.Form)
MsgBox("TypeOf Object[Form] Is Label? " & TypeOf refForm Is System.Windows.Forms.Label)
MsgBox("TypeOf Object[Form] Is Control? " & TypeOf refForm Is System.Windows.Forms.Control)
MsgBox("TypeOf Object[Form] Is IComponent? " & TypeOf refForm Is System.ComponentModel.IComponent)
```

The variable `refInteger` has a run-time type of `Integer`. It is compatible with `Integer` but not with `Double`. The variable `refForm` has a run-time type of Form. It is compatible with Form because that is its type, with Control because Form inherits from Control, and with IComponent because Form inherits from Component, which implements IComponent. However, `refForm` is not compatible with Label.

See also

- Is Operator
- IsNot Operator
- Comparison Operators in Visual Basic
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Operators and Expressions

Xor Operator (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Performs a logical exclusion on two `Boolean` expressions, or a bitwise exclusion on two numeric expressions.

Syntax

```
result = expression1 Xor expression2
```

Parts

`result`
Required. Any `Boolean` or numeric variable. For Boolean comparison, `result` is the logical exclusion (exclusive logical disjunction) of two `Boolean` values. For bitwise operations, `result` is a numeric value that represents the bitwise exclusion (exclusive bitwise disjunction) of two numeric bit patterns.

`expression1`
Required. Any `Boolean` or numeric expression.

`expression2`
Required. Any `Boolean` or numeric expression.

Remarks

For Boolean comparison, `result` is `True` if and only if exactly one of `expression1` and `expression2` evaluates to `True`. That is, if and only if `expression1` and `expression2` evaluate to opposite `Boolean` values. The following table illustrates how `result` is determined.

IF `EXPRESSION1` IS	AND `EXPRESSION2` IS	THE VALUE OF `RESULT` IS
`True`	`True`	`False`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

> **NOTE**
> In a Boolean comparison, the `Xor` operator always evaluates both expressions, which could include making procedure calls. There is no short-circuiting counterpart to `Xor`, because the result always depends on both operands. For *short-circuiting* logical operators, see AndAlso Operator and OrElse Operator.

For bitwise operations, the `Xor` operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in `result` according to the following table.

IF BIT IN `EXPRESSION1` IS	AND BIT IN `EXPRESSION2` IS	THE BIT IN `RESULT` IS
1	1	0
1	0	1
0	1	1
0	0	0

> **NOTE**
> Since the logical and bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

For example, 5 `Xor` 3 is 6. To see why this is so, convert 5 and 3 to their binary representations, 101 and 011. Then use the previous table to determine that 101 Xor 011 is 110, which is the binary representation of the decimal number 6.

Data Types

If the operands consist of one `Boolean` expression and one numeric expression, Visual Basic converts the `Boolean` expression to a numeric value (–1 for `True` and 0 for `False`) and performs a bitwise operation.

For a `Boolean` comparison, the data type of the result is `Boolean`. For a bitwise comparison, the result data type is a numeric type appropriate for the data types of `expression1` and `expression2`. See the "Relational and Bitwise Comparisons" table in Data Types of Operator Results.

Overloading

The `Xor` operator can be *overloaded*, which means that a class or structure can redefine its behavior when an operand has the type of that class or structure. If your code uses this operator on such a class or structure, make sure you understand its redefined behavior. For more information, see Operator Procedures.

Example

The following example uses the `Xor` operator to perform logical exclusion (exclusive logical disjunction) on two expressions. The result is a `Boolean` value that represents whether exactly one of the expressions is `True`.

```
Dim a As Integer = 10
Dim b As Integer = 8
Dim c As Integer = 6
Dim firstCheck, secondCheck, thirdCheck As Boolean
firstCheck = a > b Xor b > c
secondCheck = b > a Xor b > c
thirdCheck = b > a Xor c > b
```

The previous example produces results of `False`, `True`, and `False`, respectively.

Example

The following example uses the `Xor` operator to perform logical exclusion (exclusive logical disjunction) on the individual bits of two numeric expressions. The bit in the result pattern is set if exactly one of the corresponding

bits in the operands is set to 1.

```
Dim a As Integer = 10 ' 1010 in binary
Dim b As Integer = 8  ' 1000 in binary
Dim c As Integer = 6  ' 0110 in binary
Dim firstPattern, secondPattern, thirdPattern As Integer
firstPattern = (a Xor b)  '  2, 0010 in binary
secondPattern = (a Xor c) ' 12, 1100 in binary
thirdPattern = (b Xor c)  ' 14, 1110 in binary
```

The previous example produces results of 2, 12, and 14, respectively.

See also

- Logical/Bitwise Operators (Visual Basic)
- Operator Precedence in Visual Basic
- Operators Listed by Functionality
- Logical and Bitwise Operators in Visual Basic

Properties (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

This page lists the properties that are members of Visual Basic modules. Other properties that are members of specific Visual Basic objects are listed in Objects.

Visual Basic Properties

DateString	Returns or sets a `String` value representing the current date according to your system.
Now	Returns a `Date` value containing the current date and time according to your system.
ScriptEngine	Returns a `String` representing the runtime currently in use.
ScriptEngineBuildVersion	Returns an `Integer` containing the build version number of the runtime currently in use.
ScriptEngineMajorVersion	Returns an `Integer` containing the major version number of the runtime currently in use.
ScriptEngineMinorVersion	Returns an `Integer` containing the minor version number of the runtime currently in use.
TimeOfDay	Returns or sets a `Date` value containing the current time of day according to your system.
Timer	Returns a `Double` value representing the number of seconds elapsed since midnight.
TimeString	Returns or sets a `String` value representing the current time of day according to your system.
Today	Returns or sets a `Date` value containing the current date according to your system.

See also

- Visual Basic Language Reference
- Visual Basic

Queries (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Visual Basic enables you to create Language-Integrated Query (LINQ) expressions in your code.

In This Section

Aggregate Clause
Describes the `Aggregate` clause, which applies one or more aggregate functions to a collection.

Distinct Clause
Describes the `Distinct` clause, which restricts the values of the current range variable to eliminate duplicate values in query results.

From Clause
Describes the `From` clause, which specifies a collection and a range variable for a query.

Group By Clause
Describes the `Group By` clause, which groups the elements of a query result and can be used to apply aggregate functions to each group.

Group Join Clause
Describes the `Group Join` clause, which combines two collections into a single hierarchical collection.

Join Clause
Describes the `Join` clause, which combines two collections into a single collection.

Let Clause
Describes the `Let` clause, which computes a value and assigns it to a new variable in the query.

Order By Clause
Describes the `Order By` clause, which specifies the sort order for columns in a query.

Select Clause
Describes the `Select` clause, which declares a set of range variables for a query.

Skip Clause
Describes the `Skip` clause, which bypasses a specified number of elements in a collection and then returns the remaining elements.

Skip While Clause
Describes the `Skip While` clause, which bypasses elements in a collection as long as a specified condition is `true` and then returns the remaining elements.

Take Clause
Describes the `Take` clause, which returns a specified number of contiguous elements from the start of a collection.

Take While Clause
Describes the `Take While` clause, which includes elements in a collection as long as a specified condition is `true` and bypasses the remaining elements.

Where Clause
Describes the `Where` clause, which specifies a filtering condition for a query.

See also

- LINQ
- Introduction to LINQ in Visual Basic

Aggregate Clause (Visual Basic)

2/28/2019 • 6 minutes to read • Edit Online

Applies one or more aggregate functions to a collection.

Syntax

```
Aggregate element [As type] In collection _
  [, element2 [As type2] In collection2, [...]]
  [ clause ]
  Into expressionList
```

Parts

TERM	DEFINITION
`element`	Required. Variable used to iterate through the elements of the collection.
`type`	Optional. The type of `element`. If no type is specified, the type of `element` is inferred from `collection`.
`collection`	Required. Refers to the collection to operate on.
`clause`	Optional. One or more query clauses, such as a `Where` clause, to refine the query result to apply the aggregate clause or clauses to.
`expressionList`	Required. One or more comma-delimited expressions that identify an aggregate function to apply to the collection. You can apply an alias to an aggregate function to specify a member name for the query result. If no alias is supplied, the name of the aggregate function is used. For examples, see the section about aggregate functions later in this topic.

Remarks

The `Aggregate` clause can be used to include aggregate functions in your queries. Aggregate functions perform checks and computations over a set of values and return a single value. You can access the computed value by using a member of the query result type. The standard aggregate functions that you can use are the `All`, `Any`, `Average`, `Count`, `LongCount`, `Max`, `Min`, and `Sum` functions. These functions are familiar to developers who are familiar with aggregates in SQL. They are described in the following section of this topic.

The result of an aggregate function is included in the query result as a field of the query result type. You can supply an alias for the aggregate function result to specify the name of the member of the query result type that will hold the aggregate value. If no alias is supplied, the name of the aggregate function is used.

The `Aggregate` clause can begin a query, or it can be included as an additional clause in a query. If the `Aggregate` clause begins a query, the result is a single value that is the result of the aggregate function specified in the `Into` clause. If more than one aggregate function is specified in the `Into` clause, the query returns a single type with a

separate property to reference the result of each aggregate function in the `Into` clause. If the `Aggregate` clause is included as an additional clause in a query, the type returned in the query collection will have a separate property to reference the result of each aggregate function in the `Into` clause.

Aggregate Functions

The following are the standard aggregate functions that can be used with the `Aggregate` clause.

All

Returns `true` if all elements in the collection satisfy a specified condition; otherwise returns `false`. The following is an example:

```
Dim customerList1 = Aggregate order In orders
                    Into AllOrdersOver100 = All(order.Total >= 100)
```

Any

Returns `true` if any element in the collection satisfies a specified condition; otherwise returns `false`. The following is an example:

```
Dim customerList2 = From cust In customers
                    Aggregate order In cust.Orders
                    Into AnyOrderOver500 = Any(order.Total >= 500)
```

Average

Computes the average of all elements in the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerOrderAverage = Aggregate order In orders
                           Into Average(order.Total)
```

Count

Counts the number of elements in the collection. You can supply an optional `Boolean` expression to count only the number of elements in the collection that satisfy a condition. The following is an example:

```
Dim customerOrderAfter1996 = From cust In customers
                             Aggregate order In cust.Orders
                             Into Count(order.OrderDate > #12/31/1996#)
```

Group

Refers to query results that are grouped as a result of a `Group By` or `Group Join` clause. The `Group` function is valid only in the `Into` clause of a `Group By` or `Group Join` clause. For more information and examples, see Group By Clause and Group Join Clause.

LongCount

Counts the number of elements in the collection. You can supply an optional `Boolean` expression to count only the number of elements in the collection that satisfy a condition. Returns the result as a `Long`. For an example, see the `Count` aggregate function.

Max

Computes the maximum value from the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerMaxOrder = Aggregate order In orders
                       Into MaxOrder = Max(order.Total)
```

Min

Computes the minimum value from the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerMinOrder = From cust In customers
                       Aggregate order In cust.Orders
                       Into MinOrder = Min(order.Total)
```

Sum

Computes the sum of all elements in the collection, or computes a supplied expression for all elements in the collection. The following is an example:

```
Dim customerTotals = From cust In customers
                     Aggregate order In cust.Orders
                     Into Sum(order.Total)
```

Example

The following example shows how to use the `Aggregate` clause to apply aggregate functions to a query result.

```
Public Sub AggregateSample()
  Dim customers = GetCustomerList()

  Dim customerOrderTotal =
      From cust In customers
      Aggregate order In cust.Orders
      Into Sum(order.Total), MaxOrder = Max(order.Total),
      MinOrder = Min(order.Total), Avg = Average(order.Total)

  For Each customer In customerOrderTotal
    Console.WriteLine(customer.cust.CompanyName & vbCrLf &
                      vbTab & "Sum = " & customer.Sum & vbCrLf &
                      vbTab & "Min = " & customer.MinOrder & vbCrLf &
                      vbTab & "Max = " & customer.MaxOrder & vbCrLf &
                      vbTab & "Avg = " & customer.Avg.ToString("#.##"))
  Next
End Sub
```

Creating User-Defined Aggregate Functions

You can include your own custom aggregate functions in a query expression by adding extension methods to the IEnumerable<T> type. Your custom method can then perform a calculation or operation on the enumerable collection that has referenced your aggregate function. For more information about extension methods, see Extension Methods.

For example, the following example shows a custom aggregate function that calculates the median value of a collection of numbers. There are two overloads of the `Median` extension method. The first overload accepts, as input, a collection of type `IEnumerable(Of Double)`. If the `Median` aggregate function is called for a query field of type `Double`, this method will be called. The second overload of the `Median` method can be passed any generic type. The generic overload of the `Median` method takes a second parameter that references the `Func(Of T, Double)` lambda expression to project a value for a type (from a collection) as the corresponding

value of type `Double`. It then delegates the calculation of the median value to the other overload of the `Median` method. For more information about lambda expressions, see Lambda Expressions.

```
Imports System.Runtime.CompilerServices

Module UserDefinedAggregates

    ' Calculate the median value for a collection of type Double.
    <Extension()>
    Function Median(ByVal values As IEnumerable(Of Double)) As Double
        If values.Count = 0 Then
            Throw New InvalidOperationException("Cannot compute median for an empty set.")
        End If

        Dim sortedList = From number In values
                         Order By number

        Dim medianValue As Double

        Dim itemIndex = CInt(Int(sortedList.Count / 2))

        If sortedList.Count Mod 2 = 0 Then
            ' Even number of items in list.
            medianValue = ((sortedList(itemIndex) + sortedList(itemIndex - 1)) / 2)
        Else
            ' Odd number of items in list.
            medianValue = sortedList(itemIndex)
        End If

        Return medianValue
    End Function

    ' "Cast" the collection of generic items as type Double and call the
    ' Median() method to calculate the median value.
    <Extension()>
    Function Median(Of T)(ByVal values As IEnumerable(Of T),
                          ByVal selector As Func(Of T, Double)) As Double
        Return (From element In values Select selector(element)).Median()
    End Function

End Module
```

The following example shows sample queries that call the `Median` aggregate function on a collection of type `Integer`, and a collection of type `Double`. The query that calls the `Median` aggregate function on the collection of type `Double` calls the overload of the `Median` method that accepts, as input, a collection of type `Double`. The query that calls the `Median` aggregate function on the collection of type `Integer` calls the generic overload of the `Median` method.

```
Module Module1

    Sub Main()
        Dim numbers1 = {1, 2, 3, 4, 5}

        Dim query1 = Aggregate num In numbers1 Into Median(num)

        Console.WriteLine("Median = " & query1)

        Dim numbers2 = {1.9, 2, 8, 4, 5.7, 6, 7.2, 0}

        Dim query2 = Aggregate num In numbers2 Into Median()

        Console.WriteLine("Median = " & query2)
    End Sub

End Module
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Where Clause
- Group By Clause

Distinct Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Restricts the values of the current range variable to eliminate duplicate values in subsequent query clauses.

Syntax

```
Distinct
```

Remarks

You can use the `Distinct` clause to return a list of unique items. The `Distinct` clause causes the query to ignore duplicate query results. The `Distinct` clause applies to duplicate values for all return fields specified by the `Select` clause. If no `Select` clause is specified, the `Distinct` clause is applied to the range variable for the query identified in the `From` clause. If the range variable is not an immutable type, the query will only ignore a query result if all members of the type match an existing query result.

Example

The following query expression joins a list of customers and a list of customer orders. The `Distinct` clause is included to return a list of unique customer names and order dates.

```
Dim customerOrders = From cust In customers, ord In orders
                     Where cust.CustomerID = ord.CustomerID
                     Select cust.CompanyName, ord.OrderDate
                     Distinct
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- From Clause
- Select Clause
- Where Clause

Equals Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Compares keys from collections being joined.

Remarks

The `Equals` keyword is used in the following contexts:

Group Join Clause

Join Clause

See also

- Keywords

From Clause (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Specifies one or more range variables and a collection to query.

Syntax

```
From element [ As type ] In collection [ _ ]
  [, element2 [ As type2 ] In collection2 [, ... ] ]
```

Parts

TERM	DEFINITION
`element`	Required. A *range variable* used to iterate through the elements of the collection. A range variable is used to refer to each member of the `collection` as the query iterates through the `collection`. Must be an enumerable type.
`type`	Optional. The type of `element`. If no `type` is specified, the type of `element` is inferred from `collection`.
`collection`	Required. Refers to the collection to be queried. Must be an enumerable type.

Remarks

The `From` clause is used to identify the source data for a query and the variables that are used to refer to an element from the source collection. These variables are called *range variables*. The `From` clause is required for a query, except when the `Aggregate` clause is used to identify a query that returns only aggregated results. For more information, see Aggregate Clause.

You can specify multiple `From` clauses in a query to identify multiple collections to be joined. When multiple collections are specified, they are iterated over independently, or you can join them if they are related. You can join collections implicitly by using the `Select` clause, or explicitly by using the `Join` or `Group Join` clauses. As an alternative, you can specify multiple range variables and collections in a single `From` clause, with each related range variable and collection separated from the others by a comma. The following code example shows both syntax options for the `From` clause.

```
' Multiple From clauses in a query.
Dim result = From var1 In collection1, var2 In collection2

' Equivalent syntax with a single From clause.
Dim result2 = From var1 In collection1
              From var2 In collection2
```

The `From` clause defines the scope of a query, which is similar to the scope of a `For` loop. Therefore, each `element` range variable in the scope of a query must have a unique name. Because you can specify multiple `From` clauses for a query, subsequent `From` clauses can refer to range variables in the `From` clause, or they can

refer to range variables in a previous `From` clause. For example, the following example shows a nested `From` clause where the collection in the second clause is based on a property of the range variable in the first clause.

```
Dim allOrders = From cust In GetCustomerList()
                From ord In cust.Orders
                Select ord
```

Each `From` clause can be followed by any combination of additional query clauses to refine the query. You can refine the query in the following ways:

- Combine multiple collections implicitly by using the `From` and `Select` clauses, or explicitly by using the `Join` or `Group Join` clauses.
- Use the `Where` clause to filter the query result.
- Sort the result by using the `Order By` clause.
- Group similar results together by using the `Group By` clause.
- Use the `Aggregate` clause to identify aggregate functions to evaluate for the whole query result.
- Use the `Let` clause to introduce an iteration variable whose value is determined by an expression instead of a collection.
- Use the `Distinct` clause to ignore duplicate query results.
- Identify parts of the result to return by using the `Skip`, `Take`, `Skip While`, and `Take While` clauses.

Example

The following query expression uses a `From` clause to declare a range variable `cust` for each `Customer` object in the `customers` collection. The `Where` clause uses the range variable to restrict the output to customers from the specified region. The `For Each` loop displays the company name for each customer in the query result.

```
Sub DisplayCustomersForRegion(ByVal customers As List(Of Customer),
                              ByVal region As String)

  Dim customersForRegion = From cust In customers
                           Where cust.Region = region

  For Each cust In customersForRegion
    Console.WriteLine(cust.CompanyName)
  Next
End Sub
```

See also

- Queries
- Introduction to LINQ in Visual Basic
- For Each...Next Statement
- For...Next Statement
- Select Clause
- Where Clause
- Aggregate Clause
- Distinct Clause
- Join Clause

- Group Join Clause
- Order By Clause
- Let Clause
- Skip Clause
- Take Clause
- Skip While Clause
- Take While Clause

Group By Clause (Visual Basic)

5/31/2019 • 2 minutes to read • Edit Online

Groups the elements of a query result. Can also be used to apply aggregate functions to each group. The grouping operation is based on one or more keys.

Syntax

```
Group [ listField1 [, listField2 [...] ] By keyExp1 [, keyExp2 [...] ]
  Into aggregateList
```

Parts

- `listField1`, `listField2`

  Optional. One or more fields of the query variable or variables that explicitly identify the fields to be included in the grouped result. If no fields are specified, all fields of the query variable or variables are included in the grouped result.

- `keyExp1`

  Required. An expression that identifies the key to use to determine the groups of elements. You can specify more than one key to specify a composite key.

- `keyExp2`

  Optional. One or more additional keys that are combined with `keyExp1` to create a composite key.

- `aggregateList`

  Required. One or more expressions that identify how the groups are aggregated. To identify a member name for the grouped results, use the `Group` keyword, which can be in either of the following forms:

  ```
  Into Group
  ```

  -or-

  ```
  Into <alias> = Group
  ```

  You can also include aggregate functions to apply to the group.

Remarks

You can use the `Group By` clause to break the results of a query into groups. The grouping is based on a key or a composite key consisting of multiple keys. Elements that are associated with matching key values are included in the same group.

You use the `aggregateList` parameter of the `Into` clause and the `Group` keyword to identify the member name that is used to reference the group. You can also include aggregate functions in the `Into` clause to compute values for the grouped elements. For a list of standard aggregate functions, see Aggregate Clause.

Example

The following code example groups a list of customers based on their location (country/region) and provides a count of the customers in each group. The results are ordered by country/region name. The grouped results are ordered by city name.

```
Public Sub GroupBySample()
  Dim customers = GetCustomerList()

  Dim customersByCountry = From cust In customers
                           Order By cust.City
                           Group By CountryName = cust.Country
                           Into RegionalCustomers = Group, Count()
                           Order By CountryName

  For Each country In customersByCountry
    Console.WriteLine(country.CountryName &
                      " (" & country.Count & ")" & vbCrLf)

    For Each customer In country.RegionalCustomers
      Console.WriteLine(vbTab & customer.CompanyName &
                        " (" & customer.City & ")")
    Next
  Next
End Sub
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Order By Clause
- Aggregate Clause
- Group Join Clause

Group Join Clause (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

Combines two collections into a single hierarchical collection. The join operation is based on matching keys.

Syntax

```
Group Join element [As type] In collection _
  On key1 Equals key2 [ And key3 Equals key4 [... ] ] _
  Into expressionList
```

Parts

TERM	DEFINITION
`element`	Required. The control variable for the collection being joined.
`type`	Optional. The type of `element`. If no `type` is specified, the type of `element` is inferred from `collection`.
`collection`	Required. The collection to combine with the collection that is on the left side of the `Group Join` operator. A `Group Join` clause can be nested in a `Join` clause or in another `Group Join` clause.
`key1` `Equals` `key2`	Required. Identifies keys for the collections being joined. You must use the `Equals` operator to compare keys from the collections being joined. You can combine join conditions by using the `And` operator to identify multiple keys. The `key1` parameter must be from the collection on the left side of the `Join` operator. The `key2` parameter must be from the collection on the right side of the `Join` operator. The keys used in the join condition can be expressions that include more than one item from the collection. However, each key expression can contain only items from its respective collection.
`expressionList`	Required. One or more expressions that identify how the groups of elements from the collection are aggregated. To identify a member name for the grouped results, use the `Group` keyword (`<alias> = Group`). You can also include aggregate functions to apply to the group.

Remarks

The `Group Join` clause combines two collections based on matching key values from the collections being joined. The resulting collection can contain a member that references a collection of elements from the second collection that match the key value from the first collection. You can also specify aggregate functions to apply to the grouped elements from the second collection. For information about aggregate functions, see Aggregate

Clause.

Consider, for example, a collection of managers and a collection of employees. Elements from both collections have a ManagerID property that identifies the employees that report to a particular manager. The results from a join operation would contain a result for each manager and employee with a matching ManagerID value. The results from a `Group Join` operation would contain the complete list of managers. Each manager result would have a member that referenced the list of employees that were a match for the specific manager.

The collection resulting from a `Group Join` operation can contain any combination of values from the collection identified in the `From` clause and the expressions identified in the `Into` clause of the `Group Join` clause. For more information about valid expressions for the `Into` clause, see Aggregate Clause.

A `Group Join` operation will return all results from the collection identified on the left side of the `Group Join` operator. This is true even if there are no matches in the collection being joined. This is like a `LEFT OUTER JOIN` in SQL.

You can use the `Join` clause to combine collections into a single collection. This is equivalent to an `INNER JOIN` in SQL.

Example

The following code example joins two collections by using the `Group Join` clause.

```
Dim customerList = From cust In customers
                   Group Join ord In orders On
                   cust.CustomerID Equals ord.CustomerID
                   Into CustomerOrders = Group,
                        OrderTotal = Sum(ord.Total)
                   Select cust.CompanyName, cust.CustomerID,
                          CustomerOrders, OrderTotal

For Each customer In customerList
  Console.WriteLine(customer.CompanyName &
                    " (" & customer.OrderTotal & ")")

  For Each order In customer.CustomerOrders
    Console.WriteLine(vbTab & order.OrderID & ": " & order.Total)
  Next
Next
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Join Clause
- Where Clause
- Group By Clause

Join Clause (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

Combines two collections into a single collection. The join operation is based on matching keys and uses the `Equals` operator.

Syntax

```
Join element In collection _
  [ joinClause _ ]
  [ groupJoinClause ... _ ]
On key1 Equals key2 [ And key3 Equals key4 [... ]
```

Parts

`element`
Required. The control variable for the collection being joined.

`collection`
Required. The collection to combine with the collection identified on the left side of the `Join` operator. A `Join` clause can be nested in another `Join` clause, or in a `Group Join` clause.

`joinClause`
Optional. One or more additional `Join` clauses to further refine the query.

`groupJoinClause`
Optional. One or more additional `Group Join` clauses to further refine the query.

`key1` `Equals` `key2`
Required. Identifies keys for the collections being joined. You must use the `Equals` operator to compare keys from the collections being joined. You can combine join conditions by using the `And` operator to identify multiple keys. `key1` must be from the collection on the left side of the `Join` operator. `key2` must be from the collection on the right side of the `Join` operator.

The keys used in the join condition can be expressions that include more than one item from the collection. However, each key expression can contain only items from its respective collection.

Remarks

The `Join` clause combines two collections based on matching key values from the collections being joined. The resulting collection can contain any combination of values from the collection identified on the left side of the `Join` operator and the collection identified in the `Join` clause. The query will return only results for which the condition specified by the `Equals` operator is met. This is equivalent to an `INNER JOIN` in SQL.

You can use multiple `Join` clauses in a query to join two or more collections into a single collection.

You can perform an implicit join to combine collections without the `Join` clause. To do this, include multiple `In` clauses in your `From` clause and specify a `Where` clause that identifies the keys that you want to use for the join.

You can use the `Group Join` clause to combine collections into a single hierarchical collection. This is like a `LEFT OUTER JOIN` in SQL.

Example

The following code example performs an implicit join to combine a list of customers with their orders.

```
Dim customerIDs() = {"ALFKI", "VICTE", "BLAUS", "TRAIH"}

Dim customerList = From cust In customers, custID In customerIDs
                   Where cust.CustomerID = custID
                   Select cust.CompanyName

For Each companyName In customerList
  Console.WriteLine(companyName)
Next
```

Example

The following code example joins two collections by using the `Join` clause.

```
Imports System.Diagnostics
Imports System.Security.Permissions

Public Class JoinSample

  <SecurityPermission(SecurityAction.Demand)>
  Public Sub ListProcesses()
    Dim processDescriptions As New List(Of ProcessDescription)
    processDescriptions.Add(New ProcessDescription With {
                                .ProcessName = "explorer",
                                .Description = "Windows Explorer"})
    processDescriptions.Add(New ProcessDescription With {
                                .ProcessName = "winlogon",
                                .Description = "Windows Logon"})
    processDescriptions.Add(New ProcessDescription With {
                                .ProcessName = "cmd",
                                .Description = "Command Window"})
    processDescriptions.Add(New ProcessDescription With {
                                .ProcessName = "iexplore",
                                .Description = "Internet Explorer"})

    Dim processes = From proc In Process.GetProcesses
                    Join desc In processDescriptions
                      On proc.ProcessName Equals desc.ProcessName
                    Select proc.ProcessName, proc.Id, desc.Description

    For Each proc In processes
      Console.WriteLine("{0} ({1}), {2}",
                        proc.ProcessName, proc.Id, proc.Description)
    Next
  End Sub

End Class

Public Class ProcessDescription
  Public ProcessName As String
  Public Description As String
End Class
```

This example will produce output similar to the following:

```
winlogon (968), Windows Logon
```

```
explorer (2424), File Explorer
```

```
cmd (5136), Command Window
```

Example

The following code example joins two collections by using the `Join` clause with two key columns.

```
Imports System.Diagnostics
Imports System.Security.Permissions

Public Class JoinSample2

  <SecurityPermission(SecurityAction.Demand)>
  Public Sub ListProcesses()
    Dim processDescriptions As New List(Of ProcessDescription2)

    ' 8 = Normal priority, 13 = High priority
    processDescriptions.Add(New ProcessDescription2 With {
                                .ProcessName = "explorer",
                                .Description = "Windows Explorer",
                                .Priority = 8})
    processDescriptions.Add(New ProcessDescription2 With {
                                .ProcessName = "winlogon",
                                .Description = "Windows Logon",
                                .Priority = 13})
    processDescriptions.Add(New ProcessDescription2 With {
                                .ProcessName = "cmd",
                                .Description = "Command Window",
                                .Priority = 8})
    processDescriptions.Add(New ProcessDescription2 With {
                                .ProcessName = "iexplore",
                                .Description = "Internet Explorer",
                                .Priority = 8})

    Dim processes = From proc In Process.GetProcesses
                    Join desc In processDescriptions
                      On proc.ProcessName Equals desc.ProcessName And
                         proc.BasePriority Equals desc.Priority
                    Select proc.ProcessName, proc.Id, desc.Description,
                           desc.Priority

    For Each proc In processes
      Console.WriteLine("{0} ({1}), {2}, Priority = {3}",
                        proc.ProcessName,
                        proc.Id,
                        proc.Description,
                        proc.Priority)
    Next
  End Sub

End Class

Public Class ProcessDescription2
  Public ProcessName As String
  Public Description As String
  Public Priority As Integer
End Class
```

The example will produce output similar to the following:

```
winlogon (968), Windows Logon, Priority = 13
```

```
cmd (700), Command Window, Priority = 8
```

```
explorer (2424), File Explorer, Priority = 8
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Group Join Clause
- Where Clause

Let Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Computes a value and assigns it to a new variable within the query.

Syntax

```
Let variable = expression [, ...]
```

Parts

TERM	DEFINITION
`variable`	Required. An alias that can be used to reference the results of the supplied expression.
`expression`	Required. An expression that will be evaluated and assigned to the specified variable.

Remarks

The `Let` clause enables you to compute values for each query result and reference them by using an alias. The alias can be used in other clauses, such as the `Where` clause. The `Let` clause enables you to create a query statement that is easier to read because you can specify an alias for an expression clause included in the query and substitute the alias each time the expression clause is used.

You can include any number of `variable` and `expression` assignments in the `Let` clause. Separate each assignment with a comma (,).

Example

The following code example uses the `Let` clause to compute a 10 percent discount on products.

```
Dim discountedProducts = From prod In products
                         Let Discount = prod.UnitPrice * 0.1
                         Where Discount >= 50
                         Select prod.ProductName, prod.UnitPrice, Discount

For Each prod In discountedProducts
  Console.WriteLine("Product: {0}, Price: {1}, Discounted Price: {2}",
                    prod.ProductName, prod.UnitPrice.ToString("$#.00"),
                    (prod.UnitPrice - prod.Discount).ToString("$#.00"))
Next
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause

- From Clause
- Where Clause

Order By Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the sort order for a query result.

Syntax

```
Order By orderExp1 [ Ascending | Descending ] [, orderExp2 [...] ]
```

Parts

`orderExp1`
Required. One or more fields from the current query result that identify how to order the returned values. The field names must be separated by commas (,). You can identify each field as sorted in ascending or descending order by using the `Ascending` or `Descending` keywords. If no `Ascending` or `Descending` keyword is specified, the default sort order is ascending. The sort order fields are given precedence from left to right.

Remarks

You can use the `Order By` clause to sort the results of a query. The `Order By` clause can only sort a result based on the range variable for the current scope. For example, the `Select` clause introduces a new scope in a query expression with new iteration variables for that scope. Range variables defined before a `Select` clause in a query are not available after the `Select` clause. Therefore, if you want to order your results by a field that is not available in the `Select` clause, you must put the `Order By` clause before the `Select` clause. One example of when you would have to do this is when you want to sort your query by fields that are not returned as part of the result.

Ascending and descending order for a field is determined by the implementation of the IComparable interface for the data type of the field. If the data type does not implement the IComparable interface, the sort order is ignored.

Example

The following query expression uses a `From` clause to declare a range variable `book` for the `books` collection. The `Order By` clause sorts the query result by price in ascending order (the default). Books with the same price are sorted by title in ascending order. The `Select` clause selects the `Title` and `Price` properties as the values returned by the query.

```
Dim titlesAscendingPrice = From book In books
                           Order By book.Price, book.Title
                           Select book.Title, book.Price
```

Example

The following query expression uses the `Order By` clause to sort the query result by price in descending order. Books with the same price are sorted by title in ascending order.

```
Dim titlesDescendingPrice = From book In books
                            Order By book.Price Descending, book.Title
                            Select book.Title, book.Price
```

Example

The following query expression uses a `Select` clause to select the book title, price, publish date, and author. It then populates the `Title`, `Price`, `PublishDate`, and `Author` fields of the range variable for the new scope. The `Order By` clause orders the new range variable by author name, book title, and then price. Each column is sorted in the default order (ascending).

```
Dim bookOrders =
  From book In books
  Select book.Title, book.Price, book.PublishDate, book.Author
  Order By Author, Title, Price
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause

Select Clause (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

Defines the result of a query.

Syntax

```
Select [ var1 = ] fieldName1 [, [ var2 = ] fieldName2 [...] ]
```

Parts

`var1`
Optional. An alias that can be used to reference the results of the column expression.

`fieldName1`
Required. The name of the field to return in the query result.

Remarks

You can use the `Select` clause to define the results to return from a query. This enables you to either define the members of a new anonymous type that is created by a query, or to target the members of a named type that is returned by a query. The `Select` clause is not required for a query. If no `Select` clause is specified, the query will return a type based on all members of the range variables identified for the current scope. For more information, see Anonymous Types. When a query creates a named type, it will return a result of type IEnumerable<T> where `T` is the created type.

The `Select` clause can reference any variables in the current scope. This includes range variables identified in the `From` clause (or `From` clauses). It also includes any new variables created with an alias by the `Aggregate`, `Let`, `Group By`, or `Group Join` clauses, or variables from a previous `Select` clause in the query expression. The `Select` clause can also include static values. For example, the following code example shows a query expression in which the `Select` clause defines the query result as a new anonymous type with four members: `ProductName`, `Price`, `Discount`, and `DiscountedPrice`. The `ProductName` and `Price` member values are taken from the product range variable that is defined in the `From` clause. The `DiscountedPrice` member value is calculated in the `Let` clause. The `Discount` member is a static value.

```
' 10% discount
Dim discount_10 = 0.1
Dim priceList =
  From product In products
  Let DiscountedPrice = product.UnitPrice * (1 - discount_10)
  Select product.ProductName, Price = product.UnitPrice,
  Discount = discount_10, DiscountedPrice
```

The `Select` clause introduces a new set of range variables for subsequent query clauses, and previous range variables are no longer in scope. The last `Select` clause in a query expression determines the return value of the query. For example, the following query returns the company name and order ID for every customer order for which the total exceeds 500. The first `Select` clause identifies the range variables for the `Where` clause and the second `Select` clause. The second `Select` clause identifies the values returned by the query as a new anonymous type.

```
Dim customerList = From cust In customers, ord In cust.Orders
                   Select Name = cust.CompanyName,
                          Total = ord.Total, ord.OrderID
                   Where Total > 500
                   Select Name, OrderID
```

If the `Select` clause identifies a single item to return, the query expression returns a collection of the type of that single item. If the `Select` clause identifies multiple items to return, the query expression returns a collection of a new anonymous type, based on the selected items. For example, the following two queries return collections of two different types based on the `Select` clause. The first query returns a collection of company names as strings. The second query returns a collection of `Customer` objects populated with the company names and address information.

```
Dim customerNames = From cust In customers
                    Select cust.CompanyName

Dim customerInfo As IEnumerable(Of Customer) =
  From cust In customers
  Select New Customer With {.CompanyName = cust.CompanyName,
                            .Address = cust.Address,
                            .City = cust.City,
                            .Region = cust.Region,
                            .Country = cust.Country}
```

Example

The following query expression uses a `From` clause to declare a range variable `cust` for the `customers` collection. The `Select` clause selects the customer name and ID value and populates the `CompanyName` and `CustomerID` columns of the new range variable. The `For Each` statement loops over each returned object and displays the `CompanyName` and `CustomerID` columns for each record.

```
Sub SelectCustomerNameAndId(ByVal customers() As Customer)
  Dim nameIds = From cust In customers
                Select cust.CompanyName, cust.CustomerID
  For Each nameId In nameIds
    Console.WriteLine(nameId.CompanyName & ": " & nameId.CustomerID)
  Next
End Sub
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- From Clause
- Where Clause
- Order By Clause
- Anonymous Types

Skip Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Bypasses a specified number of elements in a collection and then returns the remaining elements.

Syntax

```
Skip count
```

Parts

`count`
Required. A value or an expression that evaluates to the number of elements of the sequence to skip.

Remarks

The `Skip` clause causes a query to bypass elements at the beginning of a results list and return the remaining elements. The number of elements to skip is identified by the `count` parameter.

You can use the `Skip` clause with the `Take` clause to return a range of data from any segment of a query. To do this, pass the index of the first element of the range to the `Skip` clause and the size of the range to the `Take` clause.

When you use the `Skip` clause in a query, you may also need to ensure that the results are returned in an order that will enable the `Skip` clause to bypass the intended results. For more information about ordering query results, see Order By Clause.

You can use the `SkipWhile` clause to specify that only certain elements are ignored, depending on a supplied condition.

Example

The following code example uses the `Skip` clause together with the `Take` clause to return data from a query in pages. The `GetCustomers` function uses the `Skip` clause to bypass the customers in the list until the supplied starting index value, and uses the `Take` clause to return a page of customers starting from that index value.

```
Public Sub PagingSample()
  Dim pageNumber As Integer = 0
  Dim pageSize As Integer = 10

  Dim customersPage = GetCustomers(pageNumber * pageSize, pageSize)

  Do While customersPage IsNot Nothing
    Console.WriteLine(vbCrLf & "Page: " & pageNumber + 1 & vbCrLf)

    For Each cust In customersPage
      Console.WriteLine(cust.CustomerID & ", " & cust.CompanyName)
    Next

    Console.WriteLine(vbCrLf)

    pageNumber += 1
    customersPage = GetCustomers(pageNumber * pageSize, pageSize)
  Loop
End Sub

Public Function GetCustomers(ByVal startIndex As Integer,
                             ByVal pageSize As Integer) As List(Of Customer)

  Dim customers = GetCustomerList()

  Dim returnCustomers = From cust In customers
                        Skip startIndex Take pageSize

  If returnCustomers.Count = 0 Then Return Nothing

  Return returnCustomers
End Function
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Order By Clause
- Skip While Clause
- Take Clause

Skip While Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Bypasses elements in a collection as long as a specified condition is `true` and then returns the remaining elements.

Syntax

```
Skip While expression
```

Parts

TERM	DEFINITION
`expression`	Required. An expression that represents a condition to test elements for. The expression must return a `Boolean` value or a functional equivalent, such as an `Integer` to be evaluated as a `Boolean`.

Remarks

The `Skip While` clause bypasses elements from the beginning of a query result until the supplied `expression` returns `false`. After `expression` returns `false`, the query returns all the remaining elements. The `expression` is ignored for the remaining results.

The `Skip While` clause differs from the `Where` clause in that the `Where` clause can be used to exclude all elements from a query that do not meet a particular condition. The `Skip While` clause excludes elements only until the first time that the condition is not satisfied. The `Skip While` clause is most useful when you are working with an ordered query result.

You can bypass a specific number of results from the beginning of a query result by using the `Skip` clause.

Example

The following code example uses the `Skip While` clause to bypass results until the first customer from the United States is found.

```
Public Sub SkipWhileSample()
  Dim customers = GetCustomerList()

  ' Return customers starting from the first U.S. customer encountered.
  Dim customerList = From cust In customers
                     Order By cust.Country
                     Skip While IsInternationalCustomer(cust)

  For Each cust In customerList
    Console.WriteLine(cust.CompanyName & vbTab & cust.Country)
  Next
End Sub

Public Function IsInternationalCustomer(ByVal cust As Customer) As Boolean
  If cust.Country = "USA" Then Return False

  Return True
End Function
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Skip Clause
- Take While Clause
- Where Clause

Take Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Returns a specified number of contiguous elements from the start of a collection.

Syntax

```
Take count
```

Parts

`count`
Required. A value or an expression that evaluates to the number of elements of the sequence to return.

Remarks

The `Take` clause causes a query to include a specified number of contiguous elements from the start of a results list. The number of elements to include is specified by the `count` parameter.

You can use the `Take` clause with the `Skip` clause to return a range of data from any segment of a query. To do this, pass the index of the first element of the range to the `Skip` clause and the size of the range to the `Take` clause. In this case, the `Take` clause must be specified after the `Skip` clause.

When you use the `Take` clause in a query, you may also need to ensure that the results are returned in an order that will enable the `Take` clause to include the intended results. For more information about ordering query results, see Order By Clause.

You can use the `TakeWhile` clause to specify that only certain elements be returned, depending on a supplied condition.

Example

The following code example uses the `Take` clause together with the `Skip` clause to return data from a query in pages. The GetCustomers function uses the `Skip` clause to bypass the customers in the list until the supplied starting index value, and uses the `Take` clause to return a page of customers starting from that index value.

```
Public Sub PagingSample()
  Dim pageNumber As Integer = 0
  Dim pageSize As Integer = 10

  Dim customersPage = GetCustomers(pageNumber * pageSize, pageSize)

  Do While customersPage IsNot Nothing
    Console.WriteLine(vbCrLf & "Page: " & pageNumber + 1 & vbCrLf)

    For Each cust In customersPage
      Console.WriteLine(cust.CustomerID & ", " & cust.CompanyName)
    Next

    Console.WriteLine(vbCrLf)

    pageNumber += 1
    customersPage = GetCustomers(pageNumber * pageSize, pageSize)
  Loop
End Sub

Public Function GetCustomers(ByVal startIndex As Integer,
                             ByVal pageSize As Integer) As List(Of Customer)

  Dim customers = GetCustomerList()

  Dim returnCustomers = From cust In customers
                        Skip startIndex Take pageSize

  If returnCustomers.Count = 0 Then Return Nothing

  Return returnCustomers
End Function
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Order By Clause
- Take While Clause
- Skip Clause

Take While Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Includes elements in a collection as long as a specified condition is `true` and bypasses the remaining elements.

Syntax

```
Take While expression
```

Parts

TERM	DEFINITION
`expression`	Required. An expression that represents a condition to test elements for. The expression must return a `Boolean` value or a functional equivalent, such as an `Integer` to be evaluated as a `Boolean`.

Remarks

The `Take While` clause includes elements from the start of a query result until the supplied `expression` returns `false`. After the `expression` returns `false`, the query will bypass all remaining elements. The `expression` is ignored for the remaining results.

The `Take While` clause differs from the `Where` clause in that the `Where` clause can be used to include all elements from a query that meet a particular condition. The `Take While` clause includes elements only until the first time that the condition is not satisfied. The `Take While` clause is most useful when you are working with an ordered query result.

Example

The following code example uses the `Take While` clause to retrieve results until the first customer without any orders is found.

```
Public Sub TakeWhileSample()
  Dim customers = GetCustomerList()

  ' Return customers until the first customer with no orders is found.
  Dim customersWithOrders = From cust In customers
                            Order By cust.Orders.Count Descending
                            Take While HasOrders(cust)

  For Each cust In customersWithOrders
    Console.WriteLine(cust.CompanyName & " (" & cust.Orders.Length & ")")
  Next
End Sub

Public Function HasOrders(ByVal cust As Customer) As Boolean
  If cust.Orders.Length > 0 Then Return True

  Return False
End Function
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- Select Clause
- From Clause
- Take Clause
- Skip While Clause
- Where Clause

Where Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the filtering condition for a query.

Syntax

```
Where condition
```

Parts

`condition`
Required. An expression that determines whether the values for the current item in the collection are included in the output collection. The expression must evaluate to a `Boolean` value or the equivalent of a `Boolean` value. If the condition evaluates to `True`, the element is included in the query result; otherwise, the element is excluded from the query result.

Remarks

The `Where` clause enables you to filter query data by selecting only elements that meet certain criteria. Elements whose values cause the `Where` clause to evaluate to `True` are included in the query result; other elements are excluded. The expression that is used in a `Where` clause must evaluate to a `Boolean` or the equivalent of a `Boolean`, such as an Integer that evaluates to `False` when its value is zero. You can combine multiple expressions in a `Where` clause by using logical operators such as `And`, `Or`, `AndAlso`, `OrElse`, `Is`, and `IsNot`.

By default, query expressions are not evaluated until they are accessed—for example, when they are data-bound or iterated through in a `For` loop. As a result, the `Where` clause is not evaluated until the query is accessed. If you have values external to the query that are used in the `Where` clause, ensure that the appropriate value is used in the `Where` clause at the time the query is executed. For more information about query execution, see Writing Your First LINQ Query.

You can call functions within a `Where` clause to perform a calculation or operation on a value from the current element in the collection. Calling a function in a `Where` clause can cause the query to be executed immediately when it is defined instead of when it is accessed. For more information about query execution, see Writing Your First LINQ Query.

Example

The following query expression uses a `From` clause to declare a range variable `cust` for each `Customer` object in the `customers` collection. The `Where` clause uses the range variable to restrict the output to customers from the specified region. The `For Each` loop displays the company name for each customer in the query result.

```
Sub DisplayCustomersForRegion(ByVal customers As List(Of Customer),
                              ByVal region As String)

  Dim customersForRegion = From cust In customers
                           Where cust.Region = region

  For Each cust In customersForRegion
    Console.WriteLine(cust.CompanyName)
  Next
End Sub
```

Example

The following example uses `And` and `Or` logical operators in the `Where` clause.

```
Private Sub DisplayElements()
    Dim elements As List(Of Element) = BuildList()

    ' Get a list of elements that have an atomic number from 12 to 14,
    ' or that have a name that ends in "r".
    Dim subset = From theElement In elements
        Where (theElement.AtomicNumber >= 12 And theElement.AtomicNumber < 15) _
        Or theElement.Name.EndsWith("r")
        Order By theElement.Name

    For Each theElement In subset
        Console.WriteLine(theElement.Name & " " & theElement.AtomicNumber)
    Next

    ' Output:
    '  Aluminum 13
    '  Magnesium 12
    '  Silicon 14
    '  Sulfur 16
End Sub

Private Function BuildList() As List(Of Element)
    Return New List(Of Element) From
        {
            {New Element With {.Name = "Sodium", .AtomicNumber = 11}},
            {New Element With {.Name = "Magnesium", .AtomicNumber = 12}},
            {New Element With {.Name = "Aluminum", .AtomicNumber = 13}},
            {New Element With {.Name = "Silicon", .AtomicNumber = 14}},
            {New Element With {.Name = "Phosphorous", .AtomicNumber = 15}},
            {New Element With {.Name = "Sulfur", .AtomicNumber = 16}}
        }
End Function

Public Class Element
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class
```

See also

- Introduction to LINQ in Visual Basic
- Queries
- From Clause
- Select Clause
- For Each...Next Statement

Statements (Visual Basic)

5/4/2018 • 2 minutes to read • Edit Online

The topics in this section contain tables of the Visual Basic declaration and executable statements, and of important lists that apply to many statements.

In This Section

A-E Statements

F-P Statements

Q-Z Statements

Clauses

Declaration Contexts and Default Access Levels

Attribute List

Parameter List

Type List

Related Sections

Visual Basic Language Reference

Visual Basic

A-E Statements

4/2/2019 • 2 minutes to read • Edit Online

The following table contains a listing of Visual Basic language statements.

AddHandler	Call	Class	Const
Continue	Declare	Delegate	Dim
Do...Loop	Else	End	End <keyword>
Enum	Erase	Error	Event
Exit			

See also

- F-P Statements
- Q-Z Statements
- Visual Basic Language Reference

AddHandler Statement

8/22/2019 • 2 minutes to read • Edit Online

Associates an event with an event handler at run time.

Syntax

```
AddHandler event, AddressOf eventhandler
```

Parts

event	The name of the event to handle.
`eventhandler`	The name of a procedure that handles the event.

Remarks

The `AddHandler` and `RemoveHandler` statements allow you to start and stop event handling at any time during program execution.

The signature of the `eventhandler` procedure must match the signature of the event `event`.

The `Handles` keyword and the `AddHandler` statement both allow you to specify that particular procedures handle particular events, but there are differences. The `AddHandler` statement connects procedures to events at run time. Use the `Handles` keyword when defining a procedure to specify that it handles a particular event. For more information, see Handles.

> **NOTE**
> For custom events, the `AddHandler` statement invokes the event's `AddHandler` accessor. For more information on custom events, see Event Statement.

Example

```
Sub TestEvents()
    Dim Obj As New Class1
    ' Associate an event handler with an event.
    AddHandler Obj.Ev_Event, AddressOf EventHandler
    ' Call the method to raise the event.
    Obj.CauseSomeEvent()
    ' Stop handling events.
    RemoveHandler Obj.Ev_Event, AddressOf EventHandler
    ' This event will not be handled.
    Obj.CauseSomeEvent()
End Sub

Sub EventHandler()
    ' Handle the event.
    MsgBox("EventHandler caught event.")
End Sub

Public Class Class1
    ' Declare an event.
    Public Event Ev_Event()
    Sub CauseSomeEvent()
        ' Raise an event.
        RaiseEvent Ev_Event()
    End Sub
End Class
```

See also

- RemoveHandler Statement
- Handles
- Event Statement
- Events

Call Statement (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Transfers control to a `Function`, `Sub`, or dynamic-link library (DLL) procedure.

Syntax

```
[ Call ] procedureName [ (argumentList) ]
```

Parts

`procedureName`	Required. Name of the procedure to call.
`argumentList`	Optional. List of variables or expressions representing arguments that are passed to the procedure when it is called. Multiple arguments are separated by commas. If you include `argumentList`, you must enclose it in parentheses.

Remarks

You can use the `Call` keyword when you call a procedure. For most procedure calls, you aren't required to use this keyword.

You typically use the `Call` keyword when the called expression doesn't start with an identifier. Use of the `Call` keyword for other uses isn't recommended.

If the procedure returns a value, the `Call` statement discards it.

Example

The following code shows two examples where the `Call` keyword is necessary to call a procedure. In both examples, the called expression doesn't start with an identifier.

```
Sub TestCall()
    Call (Sub() Console.Write("Hello"))()

    Call New TheClass().ShowText()
End Sub

Class TheClass
    Public Sub ShowText()
        Console.Write(" World")
    End Sub
End Class
```

See also

- Function Statement
- Sub Statement
- Declare Statement
- Lambda Expressions

Class Statement (Visual Basic)

4/28/2019 • 4 minutes to read • Edit Online

Declares the name of a class and introduces the definition of the variables, properties, events, and procedures that the class comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ MustInherit | NotInheritable ] [ Partial ] _
Class name [ ( Of typelist ) ]
    [ Inherits classname ]
    [ Implements interfacenames ]
    [ statements ]
End Class
```

Parts

TERM	DEFINITION
`attributelist`	Optional. See Attribute List.
`accessmodifier`	Optional. Can be one of the following: - Public - Protected - Friend - Private - Protected Friend - Private Protected See Access levels in Visual Basic.
`Shadows`	Optional. See Shadows.
`MustInherit`	Optional. See MustInherit.
`NotInheritable`	Optional. See NotInheritable.
`Partial`	Optional. Indicates a partial definition of the class. See Partial.
`name`	Required. Name of this class. See Declared Element Names.
`Of`	Optional. Specifies that this is a generic class.
`typelist`	Required if you use the Of keyword. List of type parameters for this class. See Type List.
`Inherits`	Optional. Indicates that this class inherits the members of another class. See Inherits Statement.

TERM	DEFINITION
`classname`	Required if you use the `Inherits` statement. The name of the class from which this class derives.
`Implements`	Optional. Indicates that this class implements the members of one or more interfaces. See Implements Statement.
`interfacenames`	Required if you use the `Implements` statement. The names of the interfaces this class implements.
`statements`	Optional. Statements which define the members of this class.
`End Class`	Required. Terminates the `Class` definition.

Remarks

A `Class` statement defines a new data type. A *class* is a fundamental building block of object-oriented programming (OOP). For more information, see Objects and Classes.

You can use `Class` only at namespace or module level. This means the *declaration context* for a class must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see Declaration Contexts and Default Access Levels.

Each instance of a class has a lifetime independent of all other instances. This lifetime begins when it is created by a New Operator clause or by a function such as CreateObject. It ends when all variables pointing to the instance have been set to Nothing or to instances of other classes.

Classes default to Friend access. You can adjust their access levels with the access modifiers. For more information, see Access levels in Visual Basic.

Rules

- **Nesting.** You can define one class within another. The outer class is called the *containing class*, and the inner class is called a *nested class*.
- **Inheritance.** If the class uses the Inherits Statement, you can specify only one base class or interface. A class cannot inherit from more than one element.

  A class cannot inherit from another class with a more restrictive access level. For example, a `Public` class cannot inherit from a `Friend` class.

  A class cannot inherit from a class nested within it.
- **Implementation.** If the class uses the Implements Statement, you must implement every member defined by every interface you specify in `interfacenames`. An exception to this is reimplementation of a base class member. For more information, see "Reimplementation" in Implements.
- **Default Property.** A class can specify at most one property as its *default property*. For more information, see Default.

Behavior

- **Access Level.** Within a class, you can declare each member with its own access level. Class members

default to Public access, except variables and constants, which default to Private access. When a class has more restricted access than one of its members, the class access level takes precedence.

- **Scope.** A class is in scope throughout its containing namespace, class, structure, or module.

  The scope of every class member is the entire class.

  Lifetime. Visual Basic does not support static classes. The functional equivalent of a static class is provided by a module. For more information, see Module Statement.

  Class members have lifetimes depending on how and where they are declared. For more information, see Lifetime in Visual Basic.

- **Qualification.** Code outside a class must qualify a member's name with the name of that class.

  If code inside a nested class makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested class, then in its containing class, and so on out to the outermost containing element.

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.
- **Shared Members.** You can control whether a member of a class is a shared or instance member.
- **Object Orientation.** Classes are object-oriented, but modules are not. You can create one or more instances of a class. For more information, see Objects and Classes.

Example

The following example uses a `Class` statement to define a class and several members.

```
Class BankAccount
    Shared interestRate As Decimal
    Private accountNumber As String
    Private accountBalance As Decimal
    Public holdOnAccount As Boolean = False

    Public ReadOnly Property Balance() As Decimal
        Get
            Return accountBalance
        End Get
    End Property

    Public Sub PostInterest()
        accountBalance = accountBalance * (1 + interestRate)
    End Sub

    Public Sub PostDeposit(ByVal amountIn As Decimal)
        accountBalance = accountBalance + amountIn
    End Sub

    Public Sub PostWithdrawal(ByVal amountOut As Decimal)
        accountBalance = accountBalance - amountOut
    End Sub
End Class
```

See also

- Objects and Classes
- Structures and Classes
- Interface Statement
- Module Statement
- Property Statement
- Object Lifetime: How Objects Are Created and Destroyed
- Generic Types in Visual Basic
- How to: Use a Generic Class

Const Statement (Visual Basic)

4/28/2019 • 4 minutes to read • Edit Online

Declares and defines one or more constants.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ]
Const constantlist
```

Parts

`attributelist`
Optional. List of attributes that apply to all the constants declared in this statement. See Attribute List in angle brackets ("`<`" and "`>`").

`accessmodifier`
Optional. Use this to specify what code can access these constants. Can be Public, Protected, Friend, Protected Friend, Private, or Private Protected.

`Shadows`
Optional. Use this to redeclare and hide a programming element in a base class. See Shadows.

`constantlist`
Required. List of constants being declared in this statement.

`constant` `[ , ` `constant` `... ]`

Each `constant` has the following syntax and parts:

`constantname` `[ As` `datatype` `] =` `initializer`

PART	DESCRIPTION
`constantname`	Required. Name of the constant. See Declared Element Names.
`datatype`	Required if `Option Strict` is `On`. Data type of the constant.
`initializer`	Required. Expression that is evaluated at compile time and assigned to the constant.

Remarks

If you have a value that never changes in your application, you can define a named constant and use it in place of a literal value. A name is easier to remember than a value. You can define the constant just once and use it in many places in your code. If in a later version you need to redefine the value, the `Const` statement is the only place you need to make a change.

You can use `Const` only at module or procedure level. This means the *declaration context* for a variable must

be a class, structure, module, procedure, or block, and cannot be a source file, namespace, or interface. For more information, see Declaration Contexts and Default Access Levels.

Local constants (inside a procedure) default to public access, and you cannot use any access modifiers on them. Class and module member constants (outside any procedure) default to private access, and structure member constants default to public access. You can adjust their access levels with the access modifiers.

Rules

- **Declaration Context.** A constant declared at module level, outside any procedure, is a *member constant*; it is a member of the class, structure, or module that declares it.

  A constant declared at procedure level is a *local constant*; it is local to the procedure or block that declares it.

- **Attributes.** You can apply attributes only to member constants, not to local constants. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local constants.

- **Modifiers.** By default, all constants are `Shared`, `Static`, and `ReadOnly`. You cannot use any of these keywords when declaring a constant.

  At procedure level, you cannot use `Shadows` or any access modifiers to declare local constants.

- **Multiple Constants.** You can declare several constants in the same declaration statement, specifying the `constantname` part for each one. Multiple constants are separated by commas.

Data Type Rules

- **Data Types.** The `Const` statement can declare the data type of a variable. You can specify any data type or the name of an enumeration.

- **Default Type.** If you do not specify `datatype`, the constant takes the data type of `initializer`. If you specify both `datatype` and `initializer`, the data type of `initializer` must be convertible to `datatype`. If neither `datatype` nor `initializer` is present, the data type defaults to `Object`.

- **Different Types.** You can specify different data types for different constants by using a separate `As` clause for each variable you declare. However, you cannot declare several constants to be of the same type by using a common `As` clause.

- **Initialization.** You must initialize the value of every constant in `constantlist`. You use `initializer` to supply an expression to be assigned to the constant. The expression can be any combination of literals, other constants that are already defined, and enumeration members that are already defined. You can use arithmetic and logical operators to combine such elements.

  You cannot use variables or functions in `initializer`. However, you can use conversion keywords such as `CByte` and `CShort`. You can also use `AscW` if you call it with a constant `String` or `Char` argument, since that can be evaluated at compile time.

Behavior

- **Scope.** Local constants are accessible only from within their procedure or block. Member constants are accessible from anywhere within their class, structure, or module.

- **Qualification.** Code outside a class, structure, or module must qualify a member constant's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local constants within that procedure or block.

Example

The following example uses the `Const` statement to declare constants for use in place of literal values.

```
' The following statements declare constants.
Const maximum As Long = 459
Public Const helpString As String = "HELP"
Private Const startValue As Integer = 5
```

Example

If you define a constant with data type `Object`, the Visual Basic compiler gives it the type of `initializer`, instead of `Object`. In the following example, the constant `naturalLogBase` has the run-time type `Decimal`.

```
Const naturalLogBase As Object = CDec(2.7182818284)
MsgBox("Run-time type of constant naturalLogBase is " &
    naturalLogBase.GetType.ToString())
```

The preceding example uses the ToString method on the Type object returned by the GetType Operator, because Type cannot be converted to `String` using `CStr`.

See also

- Asc
- AscW
- Enum Statement
- #Const Directive
- Dim Statement
- ReDim Statement
- Implicit and Explicit Conversions
- Constants and Enumerations
- Constants and Enumerations
- Type Conversion Functions

Continue Statement (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Transfers control immediately to the next iteration of a loop.

Syntax

```
Continue { Do | For | While }
```

Remarks

You can transfer from inside a `Do`, `For`, or `While` loop to the next iteration of that loop. Control passes immediately to the loop condition test, which is equivalent to transferring to the `For` or `While` statement, or to the `Do` or `Loop` statement that contains the `Until` or `While` clause.

You can use `Continue` at any location in the loop that allows transfers. The rules allowing transfer of control are the same as with the GoTo Statement.

For example, if a loop is totally contained within a `Try` block, a `Catch` block, or a `Finally` block, you can use `Continue` to transfer out of the loop. If, on the other hand, the `Try` ... `End Try` structure is contained within the loop, you cannot use `Continue` to transfer control out of the `Finally` block, and you can use it to transfer out of a `Try` or `Catch` block only if you transfer completely out of the `Try` ... `End Try` structure.

If you have nested loops of the same type, for example a `Do` loop within another `Do` loop, a `Continue Do` statement skips to the next iteration of the innermost `Do` loop that contains it. You cannot use `Continue` to skip to the next iteration of a containing loop of the same type.

If you have nested loops of different types, for example a `Do` loop within a `For` loop, you can skip to the next iteration of either loop by using either `Continue Do` or `Continue For`.

Example

The following code example uses the `Continue While` statement to skip to the next column of an array if a divisor is zero. The `Continue While` is inside a `For` loop. It transfers to the `While col < lastcol` statement, which is the next iteration of the innermost `While` loop that contains the `For` loop.

```
Dim row, col As Integer
Dim lastrow As Integer = 6
Dim lastcol As Integer = 10
Dim a(,) As Double = New Double(lastrow, lastcol) {}
Dim b(7) As Double
row = -1
While row < lastrow
    row += 1
    col = -1
    While col < lastcol
        col += 1
        a(row, col) = 0
        For i As Integer = 0 To b.GetUpperBound(0)
            If b(i) = col Then
                Continue While
            Else
                a(row, col) += (row + b(i)) / (col - b(i))
            End If
        Next i
    End While
End While
```

See also

- Do...Loop Statement
- For...Next Statement
- While...End While Statement
- Try...Catch...Finally Statement

Declare Statement

7/9/2019 • 8 minutes to read • Edit Online

Declares a reference to a procedure implemented in an external file.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Sub ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ]
' -or-
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Overloads ] _
Declare [ charsetmodifier ] [ Function ] name Lib "libname" _
[ Alias "aliasname" ] [ ([ parameterlist ]) ] [ As returntype ]
```

Parts

TERM	DEFINITION
`attributelist`	Optional. See Attribute List.
`accessmodifier`	Optional. Can be one of the following: - Public - Protected - Friend - Private - Protected Friend - Private Protected See Access levels in Visual Basic.
`Shadows`	Optional. See Shadows.
`charsetmodifier`	Optional. Specifies character set and file search information. Can be one of the following: - Ansi (default) - Unicode - Auto
`Sub`	Optional, but either `Sub` or `Function` must appear. Indicates that the external procedure does not return a value.
`Function`	Optional, but either `Sub` or `Function` must appear. Indicates that the external procedure returns a value.
`name`	Required. Name of this external reference. For more information, see Declared Element Names.

TERM	DEFINITION
`Lib`	Required. Introduces a `Lib` clause, which identifies the external file (DLL or code resource) that contains an external procedure.
`libname`	Required. Name of the file that contains the declared procedure.
`Alias`	Optional. Indicates that the procedure being declared cannot be identified within its file by the name specified in `name`. You specify its identification in `aliasname`.
`aliasname`	Required if you use the `Alias` keyword. String that identifies the procedure in one of two ways: The entry point name of the procedure within its file, within quotes (`""`) -or- A number sign (`#`) followed by an integer specifying the ordinal number of the procedure's entry point within its file
`parameterlist`	Required if the procedure takes parameters. See Parameter List.
`returntype`	Required if `Function` is specified and `Option Strict` is `On`. Data type of the value returned by the procedure.

Remarks

Sometimes you need to call a procedure defined in a file (such as a DLL or code resource) outside your project. When you do this, the Visual Basic compiler does not have access to the information it needs to call the procedure correctly, such as where the procedure is located, how it is identified, its calling sequence and return type, and the string character set it uses. The `Declare` statement creates a reference to an external procedure and supplies this necessary information.

You can use `Declare` only at module level. This means the *declaration context* for an external reference must be a class, structure, or module, and cannot be a source file, namespace, interface, procedure, or block. For more information, see Declaration Contexts and Default Access Levels.

External references default to Public access. You can adjust their access levels with the access modifiers.

Rules

- **Attributes.** You can apply attributes to an external reference. Any attribute you apply has effect only in your project, not in the external file.
- **Modifiers.** External procedures are implicitly Shared. You cannot use the `Shared` keyword when declaring an external reference, and you cannot alter its shared status.

  An external procedure cannot participate in overriding, implement interface members, or handle events. Accordingly, you cannot use the `Overrides`, `Overridable`, `NotOverridable`, `MustOverride`, `Implements`, or `Handles` keyword in a `Declare` statement.

- **External Procedure Name.** You do not have to give this external reference the same name (in `name`) as the procedure's entry-point name within its external file (`aliasname`). You can use an `Alias` clause to specify the entry-point name. This can be useful if the external procedure has the same name as a Visual Basic reserved modifier or a variable, procedure, or any other programming element in the same scope.

  > **NOTE**
  > Entry-point names in most DLLs are case-sensitive.

- **External Procedure Number.** Alternatively, you can use an `Alias` clause to specify the ordinal number of the entry point within the export table of the external file. To do this, you begin `aliasname` with a number sign (`#`). This can be useful if any character in the external procedure name is not allowed in Visual Basic, or if the external file exports the procedure without a name.

Data Type Rules

- **Parameter Data Types.** If `Option Strict` is `On`, you must specify the data type of each parameter in `parameterlist`. This can be any data type or the name of an enumeration, structure, class, or interface. Within `parameterlist`, you use an `As` clause to specify the data type of the argument to be passed to each parameter.

  > **NOTE**
  > If the external procedure was not written for the .NET Framework, you must take care that the data types correspond. For example, if you declare an external reference to a Visual Basic 6.0 procedure with an `Integer` parameter (16 bits in Visual Basic 6.0), you must identify the corresponding argument as `Short` in the `Declare` statement, because that is the 16-bit integer type in Visual Basic. Similarly, `Long` has a different data width in Visual Basic 6.0, and `Date` is implemented differently.

- **Return Data Type.** If the external procedure is a `Function` and `Option Strict` is `On`, you must specify the data type of the value returned to the calling code. This can be any data type or the name of an enumeration, structure, class, or interface.

  > **NOTE**
  > The Visual Basic compiler does not verify that your data types are compatible with those of the external procedure. If there is a mismatch, the common language runtime generates a MarshalDirectiveException exception at run time.

- **Default Data Types.** If `Option Strict` is `Off` and you do not specify the data type of a parameter in `parameterlist`, the Visual Basic compiler converts the corresponding argument to the Object Data Type. Similarly, if you do not specify `returntype`, the compiler takes the return data type to be `Object`.

  > **NOTE**
  > Because you are dealing with an external procedure that might have been written on a different platform, it is dangerous to make any assumptions about data types or to allow them to default. It is much safer to specify the data type of every parameter and of the return value, if any. This also improves the readability of your code.

Behavior

- **Scope.** An external reference is in scope throughout its class, structure, or module.
- **Lifetime.** An external reference has the same lifetime as the class, structure, or module in which it is declared.
- **Calling an External Procedure.** You call an external procedure the same way you call a `Function` or `Sub` procedure—by using it in an expression if it returns a value, or by specifying it in a Call Statement if it does not return a value.

  You pass arguments to the external procedure exactly as specified by `parameterlist` in the `Declare` statement. Do not take into account how the parameters were originally declared in the external file. Similarly, if there is a return value, use it exactly as specified by `returntype` in the `Declare` statement.
- **Character Sets.** You can specify in `charsetmodifier` how Visual Basic should marshal strings when it calls the external procedure. The `Ansi` modifier directs Visual Basic to marshal all strings to ANSI values, and the `Unicode` modifier directs it to marshal all strings to Unicode values. The `Auto` modifier directs Visual Basic to marshal strings according to .NET Framework rules based on the external reference `name`, or `aliasname` if specified. The default value is `Ansi`.

  `charsetmodifier` also specifies how Visual Basic should look up the external procedure within its external file. `Ansi` and `Unicode` both direct Visual Basic to look it up without modifying its name during the search. `Auto` directs Visual Basic to determine the base character set of the run-time platform and possibly modify the external procedure name, as follows:

  - On an ANSI platform, such as Windows 95, Windows 98, or Windows Millennium Edition, first look up the external procedure with no name modification. If that fails, append "A" to the end of the external procedure name and look it up again.
  - On a Unicode platform, such as Windows NT, Windows 2000, or Windows XP, first look up the external procedure with no name modification. If that fails, append "W" to the end of the external procedure name and look it up again.
- **Mechanism.** Visual Basic uses the .NET Framework *platform invoke* (PInvoke) mechanism to resolve and access external procedures. The `Declare` statement and the DllImportAttribute class both use this mechanism automatically, and you do not need any knowledge of PInvoke. For more information, see Walkthrough: Calling Windows APIs.

> **IMPORTANT**
>
> If the external procedure runs outside the common language runtime (CLR), it is *unmanaged code*. When you call such a procedure, for example a Windows API function or a COM method, you might expose your application to security risks. For more information, see Secure Coding Guidelines for Unmanaged Code.

Example

The following example declares an external reference to a `Function` procedure that returns the current user name. It then calls the external procedure `GetUserNameA` as part of the `getUser` procedure.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Sub GetUser()
    Dim buffer As String = New String(CChar(" "), 25)
    Dim retVal As Integer = GetUserName(buffer, 25)
    Dim userName As String = Strings.Left(buffer, InStr(buffer, Chr(0)) - 1)
    MsgBox(userName)
End Sub
```

Example

The DllImportAttribute provides an alternative way of using functions in unmanaged code. The following example declares an imported function without using a `Declare` statement.

```
' Add an Imports statement at the top of the class, structure, or
' module that uses the DllImport attribute.
Imports System.Runtime.InteropServices
```

```
<DllImportAttribute("kernel32.dll", EntryPoint:="MoveFileW",
    SetLastError:=True, CharSet:=CharSet.Unicode,
    ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(ByVal src As String,
  ByVal dst As String) As Boolean
    ' This function copies a file from the path src to the path dst.
    ' Leave this function empty. The DLLImport attribute forces calls
    ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

See also

- LastDllError
- Imports Statement (.NET Namespace and Type)
- AddressOf Operator
- Function Statement
- Sub Statement
- Parameter List
- Call Statement
- Walkthrough: Calling Windows APIs

Delegate Statement

4/2/2019 • 4 minutes to read • Edit Online

Used to declare a delegate. A delegate is a reference type that refers to a `Shared` method of a type or to an instance method of an object. Any procedure with matching parameter and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _
[ Shadows ] Delegate [ Sub | Function ] name [( Of typeparamlist )] [([ parameterlist ])] [ As type ]
```

Parts

TERM	DEFINITION
`attrlist`	Optional. List of attributes that apply to this delegate. Multiple attributes are separated by commas. You must enclose the Attribute List in angle brackets ("`<`" and "`>`").
`accessmodifier`	Optional. Specifies what code can access the delegate. Can be one of the following: - Public. Any code that can access the element that declares the delegate can access it. - Protected. Only code within the delegate's class or a derived class can access it. - Friend. Only code within the same assembly can access the delegate. - Private. Only code within the element that declares the delegate can access it. - Protected Friend Only code within the delegate's class, a derived class, or the same assembly can access the delegate. - Private Protected Only code within the delegate's class or in a derived class in the same assembly can access the delegate.
`Shadows`	Optional. Indicates that this delegate redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a `Private` element shadows a base class element, code that does not have permission to access the `Private` element accesses the base class element instead.
`Sub`	Optional, but either `Sub` or `Function` must appear. Declares this procedure as a delegate `Sub` procedure that does not return a value.

TERM	DEFINITION
`Function`	Optional, but either `Sub` or `Function` must appear. Declares this procedure as a delegate `Function` procedure that returns a value.
`name`	Required. Name of the delegate type; follows standard variable naming conventions.
`typeparamlist`	Optional. List of type parameters for this delegate. Multiple type parameters are separated by commas. Optionally, each type parameter can be declared variant by using `In` and `Out` generic modifiers. You must enclose the Type List in parentheses and introduce it with the `Of` keyword.
`parameterlist`	Optional. List of parameters that are passed to the procedure when it is called. You must enclose the Parameter List in parentheses.
`type`	Required if you specify a `Function` procedure. Data type of the return value.

Remarks

The `Delegate` statement defines the parameter and return types of a delegate class. Any procedure with matching parameters and return types can be used to create an instance of this delegate class. The procedure can then later be invoked by means of the delegate instance, by calling the delegate's `Invoke` method.

Delegates can be declared at the namespace, module, class, or structure level, but not within a procedure.

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be a reference to a method, or a lambda expression.

To specify a reference to a method, use the following syntax:

`AddressOf` [`expression` .] `methodname`

The compile-time type of the `expression` must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The `methodname` can be either a shared method or an instance method. The `methodname` is not optional, even if you create a delegate for the default method of the class.

To specify a lambda expression, use the following syntax:

`Function` ([`parm` As `type` , `parm2` As `type2` , ...]) `expression`

The signature of the function must match that of the delegate type. For more information about lambda expressions, see Lambda Expressions.

For more information about delegates, see Delegates.

Example

The following example uses the `Delegate` statement to declare a delegate for operating on two numbers and returning a number. The `DelegateTest` method takes an instance of a delegate of this type and uses it to operate on pairs of numbers.

```
Delegate Function MathOperator(
    ByVal x As Double,
    ByVal y As Double
) As Double

Function AddNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x + y
End Function

Function SubtractNumbers(
    ByVal x As Double,
    ByVal y As Double
) As Double
    Return x - y
End Function

Sub DelegateTest(
    ByVal x As Double,
    ByVal op As MathOperator,
    ByVal y As Double
)
    Dim ret As Double
    ret = op.Invoke(x, y) ' Call the method.
    MsgBox(ret)
End Sub

Protected Sub Test()
    DelegateTest(5, AddressOf AddNumbers, 3)
    DelegateTest(9, AddressOf SubtractNumbers, 3)
End Sub
```

See also

- AddressOf Operator
- Of
- Delegates
- How to: Use a Generic Class
- Generic Types in Visual Basic
- Covariance and Contravariance
- In
- Out

Dim Statement (Visual Basic)

4/26/2019 • 12 minutes to read • Edit Online

Declares and allocates storage space for one or more variables.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [[ Shared ] [ Shadows ] | [ Static ]] [ ReadOnly ]
Dim [ WithEvents ] variablelist
```

Parts

- `attributelist`

  Optional. See Attribute List.

- `accessmodifier`

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - Protected Friend
  - Private Protected

  See Access levels in Visual Basic.

- `Shared`

  Optional. See Shared.

- `Shadows`

  Optional. See Shadows.

- `Static`

  Optional. See Static.

- `ReadOnly`

  Optional. See ReadOnly.

- `WithEvents`

Optional. Specifies that these are object variables that refer to instances of a class that can raise events. See WithEvents.

- `variablelist`

Required. List of variables being declared in this statement.

```
variable [ , variable ... ]
```

Each `variable` has the following syntax and parts:

```
variablename [ ( [ boundslist ] ) ] [ As [ New ] datatype [ With { [ .propertyname = propinitializer [ , ... ] ] } ] ] [ = initializer ]
```

PART	DESCRIPTION
`variablename`	Required. Name of the variable. See Declared Element Names.
`boundslist`	Optional. List of bounds of each dimension of an array variable.
`New`	Optional. Creates a new instance of the class when the `Dim` statement runs.
`datatype`	Optional. Data type of the variable.
`With`	Optional. Introduces the object initializer list.
`propertyname`	Optional. The name of a property in the class you are making an instance of.
`propinitializer`	Required after `propertyname` =. The expression that is evaluated and assigned to the property name.
`initializer`	Optional if `New` is not specified. Expression that is evaluated and assigned to the variable when it is created.

Remarks

The Visual Basic compiler uses the `Dim` statement to determine the variable's data type and other information, such as what code can access the variable. The following example declares a variable to hold an `Integer` value.

```
Dim numberOfStudents As Integer
```

You can specify any data type or the name of an enumeration, structure, class, or interface.

```
Dim finished As Boolean
Dim monitorBox As System.Windows.Forms.Form
```

For a reference type, you use the `New` keyword to create a new instance of the class or structure that is specified by the data type. If you use `New`, you do not use an initializer expression. Instead, you supply arguments, if they are required, to the constructor of the class from which you are creating the variable.

```
Dim bottomLabel As New System.Windows.Forms.Label
```

You can declare a variable in a procedure, block, class, structure, or module. You cannot declare a variable in a source file, namespace, or interface. For more information, see Declaration Contexts and Default Access Levels.

A variable that is declared at module level, outside any procedure, is a *member variable* or *field*. Member variables are in scope throughout their class, structure, or module. A variable that is declared at procedure level is a *local variable*. Local variables are in scope only within their procedure or block.

The following access modifiers are used to declare variables outside a procedure: `Public`, `Protected`, `Friend`, `Protected Friend`, and `Private`. For more information, see Access levels in Visual Basic.

The `Dim` keyword is optional and usually omitted if you specify any of the following modifiers: `Public`, `Protected`, `Friend`, `Protected Friend`, `Private`, `Shared`, `Shadows`, `Static`, `ReadOnly`, or `WithEvents`.

```
Public maximumAllowed As Double
Protected Friend currentUserName As String
Private salary As Decimal
Static runningTotal As Integer
```

If `Option Explicit` is on (the default), the compiler requires a declaration for every variable you use. For more information, see Option Explicit Statement.

Specifying an Initial Value

You can assign a value to a variable when it is created. For a value type, you use an *initializer* to supply an expression to be assigned to the variable. The expression must evaluate to a constant that can be calculated at compile time.

```
Dim quantity As Integer = 10
Dim message As String = "Just started"
```

If an initializer is specified and a data type is not specified in an `As` clause, *type inference* is used to infer the data type from the initializer. In the following example, both `num1` and `num2` are strongly typed as integers. In the second declaration, type inference infers the type from the value 3.

```
' Use explicit typing.
Dim num1 As Integer = 3

' Use local type inference.
Dim num2 = 3
```

Type inference applies at the procedure level. It does not apply outside a procedure in a class, structure, module, or interface. For more information about type inference, see Option Infer Statement and Local Type Inference.

For information about what happens when a data type or initializer is not specified, see Default Data Types and Values later in this topic.

You can use an *object initializer* to declare instances of named and anonymous types. The following code creates an instance of a `Student` class and uses an object initializer to initialize properties.

```
Dim student1 As New Student With {.First = "Michael",
                                  .Last = "Tucker"}
```

For more information about object initializers, see How to: Declare an Object by Using an Object

Initializer, Object Initializers: Named and Anonymous Types, and Anonymous Types.

Declaring Multiple Variables

You can declare several variables in one declaration statement, specifying the variable name for each one, and following each array name with parentheses. Multiple variables are separated by commas.

```
Dim lastTime, nextTime, allTimes() As Date
```

If you declare more than one variable with one `As` clause, you cannot supply an initializer for that group of variables.

You can specify different data types for different variables by using a separate `As` clause for each variable you declare. Each variable takes the data type specified in the first `As` clause encountered after its `variablename` part.

```
Dim a, b, c As Single, x, y As Double, i As Integer
' a, b, and c are all Single; x and y are both Double
```

Arrays

You can declare a variable to hold an *array*, which can hold multiple values. To specify that a variable holds an array, follow its `variablename` immediately with parentheses. For more information about arrays, see Arrays.

You can specify the lower and upper bound of each dimension of an array. To do this, include a `boundslist` inside the parentheses. For each dimension, the `boundslist` specifies the upper bound and optionally the lower bound. The lower bound is always zero, whether you specify it or not. Each index can vary from zero through its upper bound value.

The following two statements are equivalent. Each statement declares an array of 21 `Integer` elements. When you access the array, the index can vary from 0 through 20.

```
Dim totals(20) As Integer
Dim totals(0 To 20) As Integer
```

The following statement declares a two-dimensional array of type `Double`. The array has 4 rows (3 + 1) of 6 columns (5 + 1) each. Note that an upper bound represents the highest possible value for the index, not the length of the dimension. The length of the dimension is the upper bound plus one.

```
Dim matrix2(3, 5) As Double
```

An array can have from 1 to 32 dimensions.

You can leave all the bounds blank in an array declaration. If you do this, the array has the number of dimensions you specify, but it is uninitialized. It has a value of `Nothing` until you initialize at least some of its elements. The `Dim` statement must specify bounds either for all dimensions or for no dimensions.

```
' Declare an array with blank array bounds.
Dim messages() As String
' Initialize the array.
ReDim messages(4)
```

If the array has more than one dimension, you must include commas between the parentheses to indicate the number of dimensions.

```
Dim oneDimension(), twoDimensions(,), threeDimensions(,,) As Byte
```

You can declare a *zero-length array* by declaring one of the array's dimensions to be -1. A variable that holds a zero-length array does not have the value `Nothing`. Zero-length arrays are required by certain common language runtime functions. If you try to access such an array, a runtime exception occurs. For more information, see Arrays.

You can initialize the values of an array by using an array literal. To do this, surround the initialization values with braces (`{}`).

```
Dim longArray() As Long = {0, 1, 2, 3}
```

For multidimensional arrays, the initialization for each separate dimension is enclosed in braces in the outer dimension. The elements are specified in row-major order.

```
Dim twoDimensions(,) As Integer = {{0, 1, 2}, {10, 11, 12}}
```

For more information about array literals, see Arrays.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a `Dim` statement.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	`Dim qty`	If Option Strict is off (the default), the variable is set to `Nothing`. If `Option Strict` is on, a compile-time error occurs.
No	Yes	`Dim qty = 5`	If Option Infer is on (the default), the variable takes the data type of the initializer. See Local Type Inference. If `Option Infer` is off and `Option Strict` is off, the variable takes the data type of `Object`. If `Option Infer` is off and `Option Strict` is on, a compile-time error occurs.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
Yes	No	`Dim qty As Integer`	The variable is initialized to the default value for the data type. See the table later in this section.
Yes	Yes	`Dim qty As Integer = 5`	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

If you specify a data type but do not specify an initializer, Visual Basic initializes the variable to the default value for its data type. The following table shows the default initialization values.

DATA TYPE	DEFAULT VALUE
All numeric types (including `Byte` and `SByte`)	0
`Char`	Binary 0
All reference types (including `Object`, `String`, and all arrays)	`Nothing`
`Boolean`	`False`
`Date`	12:00 AM of January 1 of the year 1 (01/01/0001 12:00:00 AM)

Each element of a structure is initialized as if it were a separate variable. If you declare the length of an array but do not initialize its elements, each element is initialized as if it were a separate variable.

Static Local Variable Lifetime

A `Static` local variable has a longer lifetime than that of the procedure in which it is declared. The boundaries of the variable's lifetime depend on where the procedure is declared and whether it is `Shared`.

PROCEDURE DECLARATION	VARIABLE INITIALIZED	VARIABLE STOPS EXISTING
In a module	The first time the procedure is called	When your program stops execution
In a class or structure, procedure is `Shared`	The first time the procedure is called either on a specific instance or on the class or structure itself	When your program stops execution
In a class or structure, procedure isn't `Shared`	The first time the procedure is called on a specific instance	When the instance is released for garbage collection (GC)

Attributes and Modifiers

You can apply attributes only to member variables, not to local variables. An attribute contributes information to the assembly's metadata, which is not meaningful for temporary storage such as local

variables.

At module level, you cannot use the `Static` modifier to declare member variables. At procedure level, you cannot use `Shared`, `Shadows`, `ReadOnly`, `WithEvents`, or any access modifiers to declare local variables.

You can specify what code can access a variable by supplying an `accessmodifier`. Class and module member variables (outside any procedure) default to private access, and structure member variables default to public access. You can adjust their access levels with the access modifiers. You cannot use access modifiers on local variables (inside a procedure).

You can specify `WithEvents` only on member variables, not on local variables inside a procedure. If you specify `WithEvents`, the data type of the variable must be a specific class type, not `Object`. You cannot declare an array with `WithEvents`. For more information about events, see Events.

> **NOTE**
> Code outside a class, structure, or module must qualify a member variable's name with the name of that class, structure, or module. Code outside a procedure or block cannot refer to any local variables within that procedure or block.

Releasing Managed Resources

The .NET Framework garbage collector disposes of managed resources without any extra coding on your part. However, you can force the disposal of a managed resource instead of waiting for the garbage collector.

If a class holds onto a particularly valuable and scarce resource (such as a database connection or file handle), you might not want to wait until the next garbage collection to clean up a class instance that's no longer in use. A class may implement the IDisposable interface to provide a way to release resources before a garbage collection. A class that implements that interface exposes a `Dispose` method that can be called to force valuable resources to be released immediately.

The `Using` statement automates the process of acquiring a resource, executing a set of statements, and then disposing of the resource. However, the resource must implement the IDisposable interface. For more information, see Using Statement.

Example

The following example declares variables by using the `Dim` statement with various options.

```
' Declare and initialize a Long variable.
Dim startingAmount As Long = 500

' Declare a variable that refers to a Button object,
' create a Button object, and assign the Button object
' to the variable.
Dim switchButton As New System.Windows.Forms.Button

' Declare a local variable that always retains its value,
' even after its procedure returns to the calling code.
Static totalSales As Double

' Declare a variable that refers to an array.
Dim highTemperature(31) As Integer

' Declare and initialize an array variable that
' holds four Boolean check values.
Dim checkValues() As Boolean = {False, False, True, False}
```

Example

The following example lists the prime numbers between 1 and 30. The scope of local variables is described in code comments.

```
Public Sub ListPrimes()
    ' The sb variable can be accessed only
    ' within the ListPrimes procedure.
    Dim sb As New System.Text.StringBuilder()

    ' The number variable can be accessed only
    ' within the For...Next block.  A different
    ' variable with the same name could be declared
    ' outside of the For...Next block.
    For number As Integer = 1 To 30
        If CheckIfPrime(number) = True Then
            sb.Append(number.ToString & " ")
        End If
    Next

    Debug.WriteLine(sb.ToString)
    ' Output: 2 3 5 7 11 13 17 19 23 29
End Sub


Private Function CheckIfPrime(ByVal number As Integer) As Boolean
    If number < 2 Then
        Return False
    Else
        ' The root and highCheck variables can be accessed
        ' only within the Else block.  Different variables
        ' with the same names could be declared outside of
        ' the Else block.
        Dim root As Double = Math.Sqrt(number)
        Dim highCheck As Integer = Convert.ToInt32(Math.Truncate(root))

        ' The div variable can be accessed only within
        ' the For...Next block.
        For div As Integer = 2 To highCheck
            If number Mod div = 0 Then
                Return False
            End If
        Next

        Return True
    End If
End Function
```

Example

In the following example, the `speedValue` variable is declared at the class level. The `Private` keyword is used to declare the variable. The variable can be accessed by any procedure in the `Car` class.

```
' Create a new instance of a Car.
Dim theCar As New Car()
theCar.Accelerate(30)
theCar.Accelerate(20)
theCar.Accelerate(-5)

Debug.WriteLine(theCar.Speed.ToString)
' Output: 45
```

```
Public Class Car
    ' The speedValue variable can be accessed by
    ' any procedure in the Car class.
    Private speedValue As Integer = 0

    Public ReadOnly Property Speed() As Integer
        Get
            Return speedValue
        End Get
    End Property

    Public Sub Accelerate(ByVal speedIncrease As Integer)
        speedValue += speedIncrease
    End Sub
End Class
```

See also

- Const Statement
- ReDim Statement
- Option Explicit Statement
- Option Infer Statement
- Option Strict Statement
- Compile Page, Project Designer (Visual Basic)
- Variable Declaration
- Arrays
- Object Initializers: Named and Anonymous Types
- Anonymous Types
- Object Initializers: Named and Anonymous Types
- How to: Declare an Object by Using an Object Initializer
- Local Type Inference

Do...Loop Statement (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Repeats a block of statements while a `Boolean` condition is `True` or until the condition becomes `True`.

Syntax

```
Do { While | Until } condition
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop
-or-
Do
    [ statements ]
    [ Continue Do ]
    [ statements ]
    [ Exit Do ]
    [ statements ]
Loop { While | Until } condition
```

Parts

TERM	DEFINITION
`Do`	Required. Starts the definition of the `Do` loop.
`While`	Required unless `Until` is used. Repeat the loop until `condition` is `False`.
`Until`	Required unless `While` is used. Repeat the loop until `condition` is `True`.
`condition`	Optional. `Boolean` expression. If `condition` is `Nothing`, Visual Basic treats it as `False`.
`statements`	Optional. One or more statements that are repeated while, or until, `condition` is `True`.
`Continue Do`	Optional. Transfers control to the next iteration of the `Do` loop.
`Exit Do`	Optional. Transfers control out of the `Do` loop.
`Loop`	Required. Terminates the definition of the `Do` loop.

Remarks

Use a `Do...Loop` structure when you want to repeat a set of statements an indefinite number of times, until a

condition is satisfied. If you want to repeat the statements a set number of times, the For...Next Statement is usually a better choice.

You can use either `While` or `Until` to specify `condition`, but not both.

You can test `condition` only one time, at either the start or the end of the loop. If you test `condition` at the start of the loop (in the `Do` statement), the loop might not run even one time. If you test at the end of the loop (in the `Loop` statement), the loop always runs at least one time.

The condition usually results from a comparison of two values, but it can be any expression that evaluates to a Boolean Data Type value (`True` or `False`). This includes values of other data types, such as numeric types, that have been converted to `Boolean`.

You can nest `Do` loops by putting one loop within another. You can also nest different kinds of control structures within each other. For more information, see Nested Control Structures.

> **NOTE**
>
> The `Do...Loop` structure gives you more flexibility than the While...End While Statement because it enables you to decide whether to end the loop when `condition` stops being `True` or when it first becomes `True`. It also enables you to test `condition` at either the start or the end of the loop.

Exit Do

The Exit Do statement can provide an alternative way to exit a `Do…Loop`. `Exit Do` transfers control immediately to the statement that follows the `Loop` statement.

`Exit Do` is often used after some condition is evaluated, for example in an `If...Then...Else` structure. You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. One use of `Exit Do` is to test for a condition that could cause an *endless loop*, which is a loop that could run a large or even infinite number of times. You can use `Exit Do` to escape the loop.

You can include any number of `Exit Do` statements anywhere in a `Do…Loop`.

When used within nested `Do` loops, `Exit Do` transfers control out of the innermost loop and into the next higher level of nesting.

Example

In the following example, the statements in the loop continue to run until the `index` variable is greater than 10. The `Until` clause is at the end of the loop.

```
Dim index As Integer = 0
Do
    Debug.Write(index.ToString & " ")
    index += 1
Loop Until index > 10

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example uses a `While` clause instead of an `Until` clause, and `condition` is tested at the start of

the loop instead of at the end.

```
Dim index As Integer = 0
Do While index <= 10
    Debug.Write(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

In the following example, `condition` stops the loop when the `index` variable is greater than 100. The `If` statement in the loop, however, causes the `Exit Do` statement to stop the loop when the index variable is greater than 10.

```
Dim index As Integer = 0
Do While index <= 100
    If index > 10 Then
        Exit Do
    End If

    Debug.Write(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example reads all lines in a text file. The OpenText method opens the file and returns a StreamReader that reads the characters. In the `Do...Loop` condition, the Peek method of the `StreamReader` determines whether there are any additional characters.

```
Private Sub ShowText(ByVal textFilePath As String)
    If System.IO.File.Exists(textFilePath) = False Then
        Debug.WriteLine("File Not Found: " & textFilePath)
    Else
        Dim sr As System.IO.StreamReader = System.IO.File.OpenText(textFilePath)

        Do While sr.Peek() >= 0
            Debug.WriteLine(sr.ReadLine())
        Loop

        sr.Close()
    End If
End Sub
```

See also

- Loop Structures
- For...Next Statement
- Boolean Data Type
- Nested Control Structures

- Exit Statement
- While...End While Statement

Else Statement (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Introduces a group of statements to be run or compiled if no other conditional group of statements has been run or compiled.

Remarks

The `Else` keyword can be used in these contexts:

If...Then...Else Statement

Select...Case Statement

#If...Then...#Else Directive

See also

- Keywords

End Statement

8/22/2019 • 2 minutes to read • Edit Online

Terminates execution immediately.

Syntax

```
End
```

Remarks

You can place the `End` statement anywhere in a procedure to force the entire application to stop running. `End` closes any files opened with an `Open` statement and clears all the application's variables. The application closes as soon as there are no other programs holding references to its objects and none of its code is running.

> **NOTE**
> The `End` statement stops code execution abruptly, and does not invoke the `Dispose` or `Finalize` method, or any other Visual Basic code. Object references held by other programs are invalidated. If an `End` statement is encountered within a `Try` or `Catch` block, control does not pass to the corresponding `Finally` block.

The `Stop` statement suspends execution, but unlike `End`, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.

Because `End` terminates your application without attending to any resources that might be open, you should try to close down cleanly before using it. For example, if your application has any forms open, you should close them before control reaches the `End` statement.

You should use `End` sparingly, and only when you need to stop immediately. The normal ways to terminate a procedure (Return Statement and Exit Statement) not only close down the procedure cleanly but also give the calling code the opportunity to close down cleanly. A console application, for example, can simply `Return` from the `Main` procedure.

> **IMPORTANT**
> The `End` statement calls the Exit method of the Environment class in the System namespace. Exit requires that you have `UnmanagedCode` permission. If you do not, a SecurityException error occurs.

When followed by an additional keyword, End <keyword> Statement delineates the end of the definition of the appropriate procedure or block. For example, `End Function` terminates the definition of a `Function` procedure.

Example

The following example uses the `End` statement to terminate code execution if the user requests it.

```
Sub Form_Load()
  Dim answer As MsgBoxResult
  answer = MsgBox("Do you want to quit now?", MsgBoxStyle.YesNo)
  If answer = MsgBoxResult.Yes Then
      MsgBox("Terminating program")
      End
  End If
End Sub
```

Smart Device Developer Notes

This statement is not supported.

See also

- SecurityPermissionFlag
- Stop Statement
- End <keyword> Statement

End <keyword> Statement (Visual Basic)

1/30/2019 • 2 minutes to read • Edit Online

When followed by an additional keyword, terminates the definition of the statement block introduced by that keyword.

Syntax

```
End AddHandler
End Class
End Enum
End Event
End Function
End Get
End If
End Interface
End Module
End Namespace
End Operator
End Property
End RaiseEvent
End RemoveHandler
End Select
End Set
End Structure
End Sub
End SyncLock
End Try
End While
End With
```

Parts

PART	DESCRIPTION
`End`	Required. Terminates the definition of the programming element.
`AddHandler`	Required to terminate an `AddHandler` accessor begun by a matching `AddHandler` statement in a custom Event Statement.
`Class`	Required to terminate a class definition begun by a matching Class Statement.
`Enum`	Required to terminate an enumeration definition begun by a matching Enum Statement.
`Event`	Required to terminate a `Custom` event definition begun by a matching Event Statement.

PART	DESCRIPTION
`Function`	Required to terminate a `Function` procedure definition begun by a matching Function Statement. If execution encounters an `End Function` statement, control returns to the calling code.
`Get`	Required to terminate a `Property` procedure definition begun by a matching Get Statement. If execution encounters an `End Get` statement, control returns to the statement requesting the property's value.
`If`	Required to terminate an `If` ... `Then` ... `Else` block definition begun by a matching `If` statement. See If...Then...Else Statement.
`Interface`	Required to terminate an interface definition begun by a matching Interface Statement.
`Module`	Required to terminate a module definition begun by a matching Module Statement.
`Namespace`	Required to terminate a namespace definition begun by a matching Namespace Statement.
`Operator`	Required to terminate an operator definition begun by a matching Operator Statement.
`Property`	Required to terminate a property definition begun by a matching Property Statement.
`RaiseEvent`	Required to terminate a `RaiseEvent` accessor begun by a matching `RaiseEvent` statement in a custom Event Statement.
`RemoveHandler`	Required to terminate a `RemoveHandler` accessor begun by a matching `RemoveHandler` statement in a custom Event Statement.
`Select`	Required to terminate a `Select` ... `Case` block definition begun by a matching `Select` statement. See Select...Case Statement.
`Set`	Required to terminate a `Property` procedure definition begun by a matching Set Statement. If execution encounters an `End Set` statement, control returns to the statement setting the property's value.
`Structure`	Required to terminate a structure definition begun by a matching Structure Statement.
`Sub`	Required to terminate a `Sub` procedure definition begun by a matching Sub Statement. If execution encounters an `End Sub` statement, control returns to the calling code.

PART	DESCRIPTION
`SyncLock`	Required to terminate a `SyncLock` block definition begun by a matching `SyncLock` statement. See SyncLock Statement.
`Try`	Required to terminate a `Try` ... `Catch` ... `Finally` block definition begun by a matching `Try` statement. See Try...Catch...Finally Statement.
`While`	Required to terminate a `While` loop definition begun by a matching `While` statement. See While...End While Statement.
`With`	Required to terminate a `With` block definition begun by a matching `With` statement. See With...End With Statement.

Directives

When preceded by a number sign (`#`), the `End` keyword terminates a preprocessing block introduced by the corresponding directive.

```
#End ExternalSource
#End If
#End Region
```

PART	DESCRIPTION
`#End`	Required. Terminates the definition of the preprocessing block.
`ExternalSource`	Required to terminate an external source block begun by a matching #ExternalSource Directive.
`If`	Required to terminate a conditional compilation block begun by a matching `#If` directive. See #If...Then...#Else Directives.
`Region`	Required to terminate a source region block begun by a matching #Region Directive.

Remarks

The End Statement, without an additional keyword, terminates execution immediately.

Smart Device Developer Notes

The `End` statement, without an additional keyword, is not supported.

See also

- End Statement

Enum Statement (Visual Basic)

4/26/2019 • 7 minutes to read • Edit Online

Declares an enumeration and defines the values of its members.

Syntax

```
[ <attributelist> ] [ accessmodifier ]  [ Shadows ]
Enum enumerationname [ As datatype ]
   memberlist
End Enum
```

Parts

- `attributelist`

  Optional. List of attributes that apply to this enumeration. You must enclose the attribute list in angle brackets ("`<`" and "`>`").

  The FlagsAttribute attribute indicates that the value of an instance of the enumeration can include multiple enumeration members, and that each member represents a bit field in the enumeration value.

- `accessmodifier`

  Optional. Specifies what code can access this enumeration. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - Protected Friend
  - Private Protected

- `Shadows`

  Optional. Specifies that this enumeration redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can specify Shadows only on the enumeration itself, not on any of its members.

- `enumerationname`

  Required. Name of the enumeration. For information on valid names, see Declared Element Names.

- `datatype`

  Optional. Data type of the enumeration and all its members.

- `memberlist`

  Required. List of member constants being declared in this statement. Multiple members appear on individual source code lines.

Each `member` has the following syntax and parts: `[<attribute list>] member name [ = initializer ]`

PART	DESCRIPTION
`membername`	Required. Name of this member.
`initializer`	Optional. Expression that is evaluated at compile time and assigned to this member.

- `End` `Enum`

  Terminates the `Enum` block.

Remarks

If you have a set of unchanging values that are logically related to each other, you can define them together in an enumeration. This provides meaningful names for the enumeration and its members, which are easier to remember than their values. You can then use the enumeration members in many places in your code.

The benefits of using enumerations include the following:

- Reduces errors caused by transposing or mistyping numbers.
- Makes it easy to change values in the future.
- Makes code easier to read, which means it is less likely that errors will be introduced.
- Ensures forward compatibility. If you use enumerations, your code is less likely to fail if in the future someone changes the values corresponding to the member names.

An enumeration has a name, an underlying data type, and a set of members. Each member represents a constant.

An enumeration declared at class, structure, module, or interface level, outside any procedure, is a *member enumeration*. It is a member of the class, structure, module, or interface that declares it.

Member enumerations can be accessed from anywhere within their class, structure, module, or interface. Code outside a class, structure, or module must qualify a member enumeration's name with the name of that class, structure, or module. You can avoid the need to use fully qualified names by adding an Imports statement to the source file.

An enumeration declared at namespace level, outside any class, structure, module, or interface, is a member of the namespace in which it appears.

The *declaration context* for an enumeration must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure. For more information, see Declaration Contexts and Default Access Levels.

You can apply attributes to an enumeration as a whole, but not to its members individually. An attribute contributes information to the assembly's metadata.

Data Type

The `Enum` statement can declare the data type of an enumeration. Each member takes the enumeration's data type. You can specify `Byte`, `Integer`, `Long`, `SByte`, `Short`, `UInteger`, `ULong`, or `UShort`.

If you do not specify `datatype` for the enumeration, each member takes the data type of its `initializer`. If you specify both `datatype` and `initializer`, the data type of `initializer` must be convertible to `datatype`.

If neither `datatype` nor `initializer` is present, the data type defaults to `Integer`.

Initializing Members

The `Enum` statement can initialize the contents of selected members in `memberlist`. You use `initializer` to supply an expression to be assigned to the member.

If you do not specify `initializer` for a member, Visual Basic initializes it either to zero (if it is the first `member` in `memberlist`), or to a value greater by one than that of the immediately preceding `member`.

The expression supplied in each `initializer` can be any combination of literals, other constants that are already defined, and enumeration members that are already defined, including a previous member of this enumeration. You can use arithmetic and logical operators to combine such elements.

You cannot use variables or functions in `initializer`. However, you can use conversion keywords such as `CByte` and `CShort`. You can also use `AscW` if you call it with a constant `String` or `Char` argument, since that can be evaluated at compile time.

Enumerations cannot have floating-point values. If a member is assigned a floating-point value and `Option Strict` is set to on, a compiler error occurs. If `Option Strict` is off, the value is automatically converted to the `Enum` type.

If the value of a member exceeds the allowable range for the underlying data type, or if you initialize any member to the maximum value allowed by the underlying data type, the compiler reports an error.

Modifiers

Class, structure, module, and interface member enumerations default to public access. You can adjust their access levels with the access modifiers. Namespace member enumerations default to friend access. You can adjust their access levels to public, but not to private or protected. For more information, see Access levels in Visual Basic.

All enumeration members have public access, and you cannot use any access modifiers on them. However, if the enumeration itself has a more restricted access level, the specified enumeration access level takes precedence.

By default, all enumerations are types and their fields are constants. Therefore the `Shared`, `Static`, and `ReadOnly` keywords cannot be used when declaring an enumeration or its members.

Assigning Multiple Values

Enumerations typically represent mutually exclusive values. By including the FlagsAttribute attribute in the `Enum` declaration, you can instead assign multiple values to an instance of the enumeration. The FlagsAttribute attribute specifies that the enumeration be treated as a bit field, that is, a set of flags. These are called *bitwise* enumerations.

When you declare an enumeration by using the FlagsAttribute attribute, we recommend that you use powers of 2, that is, 1, 2, 4, 8, 16, and so on, for the values. We also recommend that "None" be the name of a member whose value is 0. For additional guidelines, see FlagsAttribute and Enum.

Example

The following example shows how to use the `Enum` statement. Note that the member is referred to as `EggSizeEnum.Medium`, and not as `Medium`.

```
Public Class Egg
    Enum EggSizeEnum
        Jumbo
        ExtraLarge
        Large
        Medium
        Small
    End Enum

    Public Sub Poach()
        Dim size As EggSizeEnum

        size = EggSizeEnum.Medium
        ' Continue processing...
    End Sub
End Class
```

Example

The method in the following example is outside the `Egg` class. Therefore, `EggSizeEnum` is fully qualified as `Egg.EggSizeEnum`.

```
Public Sub Scramble(ByVal size As Egg.EggSizeEnum)
    ' Process for the three largest sizes.
    ' Throw an exception for any other size.
    Select Case size
        Case Egg.EggSizeEnum.Jumbo
            ' Process.
        Case Egg.EggSizeEnum.ExtraLarge
            ' Process.
        Case Egg.EggSizeEnum.Large
            ' Process.
        Case Else
            Throw New ApplicationException("size is invalid: " & size.ToString)
    End Select
End Sub
```

Example

The following example uses the `Enum` statement to define a related set of named constant values. In this case, the values are colors you might choose to design data entry forms for a database.

```
Public Enum InterfaceColors
    MistyRose = &HE1E4FF&
    SlateGray = &H908070&
    DodgerBlue = &HFF901E&
    DeepSkyBlue = &HFFBF00&
    SpringGreen = &H7FFF00&
    ForestGreen = &H228B22&
    Goldenrod = &H20A5DA&
    Firebrick = &H2222B2&
End Enum
```

Example

The following example shows values that include both positive and negative numbers.

```
Enum SecurityLevel
    IllegalEntry = -1
    MinimumSecurity = 0
    MaximumSecurity = 1
End Enum
```

Example

In the following example, an `As` clause is used to specify the `datatype` of an enumeration.

```
Public Enum MyEnum As Byte
  Zero
  One
  Two
End Enum
```

Example

The following example shows how to use a bitwise enumeration. Multiple values can be assigned to an instance of a bitwise enumeration. The `Enum` declaration includes the FlagsAttribute attribute, which indicates that the enumeration can be treated as a set of flags.

```
' Apply the Flags attribute, which allows an instance
' of the enumeration to have multiple values.
<Flags()> Public Enum FilePermissions As Integer
    None = 0
    Create = 1
    Read = 2
    Update = 4
    Delete = 8
End Enum

Public Sub ShowBitwiseEnum()

    ' Declare the non-exclusive enumeration object and
    ' set it to multiple values.
    Dim perm As FilePermissions
    perm = FilePermissions.Read Or FilePermissions.Update

    ' Show the values in the enumeration object.
    Console.WriteLine(perm.ToString)
    ' Output: Read, Update

    ' Show the total integer value of all values
    ' in the enumeration object.
    Console.WriteLine(CInt(perm))
    ' Output: 6

    ' Show whether the enumeration object contains
    ' the specified flag.
    Console.WriteLine(perm.HasFlag(FilePermissions.Update))
    ' Output: True
End Sub
```

Example

The following example iterates through an enumeration. It uses the GetNames method to retrieve an array of member names from the enumeration, and GetValues to retrieve an array of member values.

```
Enum EggSizeEnum
    Jumbo
    ExtraLarge
    Large
    Medium
    Small
End Enum

Public Sub Iterate()
    Dim names = [Enum].GetNames(GetType(EggSizeEnum))
    For Each name In names
        Console.Write(name & " ")
    Next
    Console.WriteLine()
    ' Output: Jumbo ExtraLarge Large Medium Small

    Dim values = [Enum].GetValues(GetType(EggSizeEnum))
    For Each value In values
        Console.Write(value & " ")
    Next
    Console.WriteLine()
    ' Output: 0 1 2 3 4
End Sub
```

See also

- Enum
- AscW
- Const Statement
- Dim Statement
- Implicit and Explicit Conversions
- Type Conversion Functions
- Constants and Enumerations

Erase Statement (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Used to release array variables and deallocate the memory used for their elements.

Syntax

```
Erase arraylist
```

Parts

`arraylist`
Required. List of array variables to be erased. Multiple variables are separated by commas.

Remarks

The `Erase` statement can appear only at procedure level. This means you can release arrays inside a procedure but not at class or module level.

The `Erase` statement is equivalent to assigning `Nothing` to each array variable.

Example

The following example uses the `Erase` statement to clear two arrays and free their memory (1000 and 100 storage elements, respectively). The `ReDim` statement then assigns a new array instance to the three-dimensional array.

```
Dim threeDimArray(9, 9, 9), twoDimArray(9, 9) As Integer
Erase threeDimArray, twoDimArray
ReDim threeDimArray(4, 4, 9)
```

See also

- Nothing
- ReDim Statement

Error Statement

8/22/2019 • 2 minutes to read • Edit Online

Simulates the occurrence of an error.

Syntax

```
Error errornumber
```

Parts

`errornumber`
Required. Can be any valid error number.

Remarks

The `Error` statement is supported for backward compatibility. In new code, especially when creating objects, use the `Err` object's `Raise` method to generate run-time errors.

If `errornumber` is defined, the `Error` statement calls the error handler after the properties of the `Err` object are assigned the following default values:

PROPERTY	VALUE
`Number`	Value specified as argument to `Error` statement. Can be any valid error number.
`Source`	Name of the current Visual Basic project.
`Description`	String expression corresponding to the return value of the `Error` function for the specified `Number`, if this string exists. If the string does not exist, `Description` contains a zero-length string ("").
`HelpFile`	The fully qualified drive, path, and file name of the appropriate Visual Basic Help file.
`HelpContext`	The appropriate Visual Basic Help file context ID for the error corresponding to the `Number` property.
`LastDLLError`	Zero.

If no error handler exists, or if none is enabled, an error message is created and displayed from the `Err` object properties.

> **NOTE**
> Some Visual Basic host applications cannot create objects. See your host application's documentation to determine whether it can create classes and objects.

Example

This example uses the `Error` statement to generate error number 11.

```
On Error Resume Next   ' Defer error handling.
Error 11   ' Simulate the "Division by zero" error.
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- Clear
- Err
- Raise
- On Error Statement
- Resume Statement
- Error Messages

Event Statement

8/22/2019 • 6 minutes to read • Edit Online

Declares a user-defined event.

Syntax

```
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname[(parameterlist)] _
[ Implements implementslist ]
' -or-
[ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Event eventname As delegatename _
[ Implements implementslist ]
' -or-
 [ <attrlist> ] [ accessmodifier ] _
[ Shared ] [ Shadows ] Custom Event eventname As delegatename _
[ Implements implementslist ]
   [ <attrlist> ] AddHandler(ByVal value As delegatename)
      [ statements ]
   End AddHandler
   [ <attrlist> ] RemoveHandler(ByVal value As delegatename)
      [ statements ]
   End RemoveHandler
   [ <attrlist> ] RaiseEvent(delegatesignature)
      [ statements ]
   End RaiseEvent
End Event
```

Parts

PART	DESCRIPTION
`attrlist`	Optional. List of attributes that apply to this event. Multiple attributes are separated by commas. You must enclose the Attribute List in angle brackets ("`<`" and "`>`").
`accessmodifier`	Optional. Specifies what code can access the event. Can be one of the following: - Public—any code that can access the element that declares it can access it. - Protected—only code within its class or a derived class can access it. - Friend—only code in the same assembly can access it. - Private—only code in the element that declares it can access it. - Protected Friend-only code in the event's class, a derived class, or the same assembly can access it. - Private Protected-only code in the event's class or a derived class in the same assembly can access it.
`Shared`	Optional. Specifies that this event is not associated with a specific instance of a class or structure.

PART	DESCRIPTION
`Shadows`	Optional. Indicates that this event redeclares and hides an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, except from where the shadowing element is inaccessible. For example, if a `Private` element shadows a base-class element, code that does not have permission to access the `Private` element accesses the base-class element instead.
`eventname`	Required. Name of the event; follows standard variable naming conventions.
`parameterlist`	Optional. List of local variables that represent the parameters of this event. You must enclose the Parameter List in parentheses.
`Implements`	Optional. Indicates that this event implements an event of an interface.
`implementslist`	Required if `Implements` is supplied. List of `Sub` procedures being implemented. Multiple procedures are separated by commas: *implementedprocedure* [, *implementedprocedure* ...] Each `implementedprocedure` has the following syntax and parts: `interface` . `definedname` - `interface` - Required. Name of an interface that this procedure's containing class or structure is implementing. - `Definedname` - Required. Name by which the procedure is defined in `interface`. This does not have to be the same as `name`, the name that this procedure is using to implement the defined procedure.
`Custom`	Required. Events declared as `Custom` must define custom `AddHandler`, `RemoveHandler`, and `RaiseEvent` accessors.
`delegatename`	Optional. The name of a delegate that specifies the event-handler signature.
`AddHandler`	Required. Declares an `AddHandler` accessor, which specifies the statements to execute when an event handler is added, either explicitly by using the `AddHandler` statement or implicitly by using the `Handles` clause.
`End AddHandler`	Required. Terminates the `AddHandler` block.
`value`	Required. Parameter name.

PART	DESCRIPTION
`RemoveHandler`	Required. Declares a `RemoveHandler` accessor, which specifies the statements to execute when an event handler is removed using the `RemoveHandler` statement.
`End RemoveHandler`	Required. Terminates the `RemoveHandler` block.
`RaiseEvent`	Required. Declares a `RaiseEvent` accessor, which specifies the statements to execute when the event is raised using the `RaiseEvent` statement. Typically, this invokes a list of delegates maintained by the `AddHandler` and `RemoveHandler` accessors.
`End RaiseEvent`	Required. Terminates the `RaiseEvent` block.
`delegatesignature`	Required. List of parameters that matches the parameters required by the `delegatename` delegate. You must enclose the Parameter List in parentheses.
`statements`	Optional. Statements that contain the bodies of the `AddHandler`, `RemoveHandler`, and `RaiseEvent` methods.
`End Event`	Required. Terminates the `Event` block.

Remarks

Once the event has been declared, use the `RaiseEvent` statement to raise the event. A typical event might be declared and raised as shown in the following fragments:

```
Public Class EventSource
    ' Declare an event.
    Public Event LogonCompleted(ByVal UserName As String)
    Sub CauseEvent()
        ' Raise an event on successful logon.
        RaiseEvent LogonCompleted("AustinSteele")
    End Sub
End Class
```

> **NOTE**
>
> You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, `ParamArray` arguments, or `Optional` arguments. Events do not have return values.

To handle an event, you must associate it with an event handler subroutine using either the `Handles` or `AddHandler` statement. The signatures of the subroutine and the event must match. To handle a shared event, you must use the `AddHandler` statement.

You can use `Event` only at module level. This means the *declaration context* for an event must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see Declaration Contexts and Default Access Levels.

In most circumstances, you can use the first syntax in the Syntax section of this topic for declaring events. However, some scenarios require that you have more control over the detailed behavior of the event. The last syntax in the Syntax section of this topic, which uses the `Custom` keyword, provides that control by enabling you to define custom events. In a custom event, you specify exactly what occurs when code adds or removes an event handler to or from the event, or when code raises the event. For examples, see How to: Declare Custom Events To Conserve Memory and How to: Declare Custom Events To Avoid Blocking.

Example

The following example uses events to count down seconds from 10 to 0. The code illustrates several of the event-related methods, properties, and statements. This includes the `RaiseEvent` statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Button1`) and a text box (`TextBox1`). When you click the button, the first text box displays a countdown from 10 to 0 seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Forms project. Then add a button named `Button1` and a text box named `TextBox1` to the main form, named `Form1` . Then right-click the form and click **View Code** to open the code editor.

Add a `WithEvents` variable to the declarations section of the `Form1` class:

```
Private WithEvents mText As TimerState
```

Add the following code to the code for `Form1` . Replace any duplicate procedures that may exist, such as `Form_Load` or `Button_Click` .

```
Private Sub Form1_Load() Handles MyBase.Load
    Button1.Text = "Start"
    mText = New TimerState
End Sub
Private Sub Button1_Click() Handles Button1.Click
    mText.StartCountdown(10.0, 0.1)
End Sub

Private Sub mText_ChangeText() Handles mText.Finished
    TextBox1.Text = "Done"
End Sub

Private Sub mText_UpdateTime(ByVal Countdown As Double
  ) Handles mText.UpdateTime

    TextBox1.Text = Format(Countdown, "##0.0")
    ' Use DoEvents to allow the display to refresh.
    My.Application.DoEvents()
End Sub

Class TimerState
    Public Event UpdateTime(ByVal Countdown As Double)
    Public Event Finished()
    Public Sub StartCountdown(ByVal Duration As Double,
                              ByVal Increment As Double)
        Dim Start As Double = DateAndTime.Timer
        Dim ElapsedTime As Double = 0

        Dim SoFar As Double = 0
        Do While ElapsedTime < Duration
            If ElapsedTime > SoFar + Increment Then
                SoFar += Increment
                RaiseEvent UpdateTime(Duration - SoFar)
            End If
            ElapsedTime = DateAndTime.Timer - Start
        Loop
        RaiseEvent Finished()
    End Sub
End Class
```

Press F5 to run the previous example, and click the button labeled **Start**. The first text box starts to count down the seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

> **NOTE**
>
> The `My.Application.DoEvents` method does not process events in the same way the form does. To enable the form to handle the events directly, you can use multithreading. For more information, see Managed Threading.

See also

- RaiseEvent Statement
- Implements Statement
- Events
- AddHandler Statement
- RemoveHandler Statement
- Handles
- Delegate Statement
- How to: Declare Custom Events To Conserve Memory
- How to: Declare Custom Events To Avoid Blocking

- Shared
- Shadows

Exit Statement (Visual Basic)

4/2/2019 • 3 minutes to read • Edit Online

Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

Syntax

```
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

Statements

`Exit Do`
Immediately exits the `Do` loop in which it appears. Execution continues with the statement following the `Loop` statement. `Exit Do` can be used only inside a `Do` loop. When used within nested `Do` loops, `Exit Do` exits the innermost loop and transfers control to the next higher level of nesting.

`Exit For`
Immediately exits the `For` loop in which it appears. Execution continues with the statement following the `Next` statement. `Exit For` can be used only inside a `For` ... `Next` or `For Each` ... `Next` loop. When used within nested `For` loops, `Exit For` exits the innermost loop and transfers control to the next higher level of nesting.

`Exit Function`
Immediately exits the `Function` procedure in which it appears. Execution continues with the statement following the statement that called the `Function` procedure. `Exit Function` can be used only inside a `Function` procedure.

To specify a return value, you can assign the value to the function name on a line before the `Exit Function` statement. To assign the return value and exit the function in one statement, you can instead use the Return Statement.

`Exit Property`
Immediately exits the `Property` procedure in which it appears. Execution continues with the statement that called the `Property` procedure, that is, with the statement requesting or setting the property's value. `Exit Property` can be used only inside a property's `Get` or `Set` procedure.

To specify a return value in a `Get` procedure, you can assign the value to the function name on a line before the `Exit Property` statement. To assign the return value and exit the `Get` procedure in one statement, you can instead use the `Return` statement.

In a `Set` procedure, the `Exit Property` statement is equivalent to the `Return` statement.

`Exit Select`
Immediately exits the `Select Case` block in which it appears. Execution continues with the statement following the `End Select` statement. `Exit Select` can be used only inside a `Select Case` statement.

`Exit Sub`
Immediately exits the `Sub` procedure in which it appears. Execution continues with the statement following the statement that called the `Sub` procedure. `Exit Sub` can be used only inside a `Sub` procedure.

In a `Sub` procedure, the `Exit Sub` statement is equivalent to the `Return` statement.

`Exit Try`
Immediately exits the `Try` or `Catch` block in which it appears. Execution continues with the `Finally` block if there is one, or with the statement following the `End Try` statement otherwise. `Exit Try` can be used only inside a `Try` or `Catch` block, and not inside a `Finally` block.

`Exit While`
Immediately exits the `While` loop in which it appears. Execution continues with the statement following the `End While` statement. `Exit While` can be used only inside a `While` loop. When used within nested `While` loops, `Exit While` transfers control to the loop that is one nested level above the loop where `Exit While` occurs.

Remarks

Do not confuse `Exit` statements with `End` statements. `Exit` does not define the end of a statement.

Example

In the following example, the loop condition stops the loop when the `index` variable is greater than 100. The `If` statement in the loop, however, causes the `Exit Do` statement to stop the loop when the index variable is greater than 10.

```
Dim index As Integer = 0
Do While index <= 100
    If index > 10 Then
        Exit Do
    End If

    Debug.Write(index.ToString & " ")
    index += 1
Loop

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example assigns the return value to the function name `myFunction`, and then uses `Exit Function` to return from the function.

```
Function MyFunction(ByVal j As Integer) As Double
    MyFunction = 3.87 * j
    Exit Function
End Function
```

Example

The following example uses the Return Statement to assign the return value and exit the function.

```
Function MyFunction(ByVal j As Integer) As Double
    Return 3.87 * j
End Function
```

See also

- Continue Statement
- Do...Loop Statement
- End Statement
- For Each...Next Statement
- For...Next Statement
- Function Statement
- Return Statement
- Stop Statement
- Sub Statement
- Try...Catch...Finally Statement

F-P Statements

4/2/2019 • 2 minutes to read • Edit Online

The following table contains a listing of Visual Basic language statements.

For Each...Next	For...Next	Function	Get
GoTo	If...Then...Else	Implements	Imports (.NET Namespace and Type)
Imports (XML Namespace)	Inherits	Interface	Mid
Module	Namespace	On Error	Operator
Option <keyword>	Option Compare	Option Explicit	Option Infer
Option Strict	Property		

See also

- A-E Statements
- Q-Z Statements
- Visual Basic Language Reference

For Each...Next Statement (Visual Basic)

8/27/2019 • 13 minutes to read • Edit Online

Repeats a group of statements for each element in a collection.

Syntax

```
For Each element [ As datatype ] In group
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ element ]
```

Parts

TERM	DEFINITION
`element`	Required in the `For Each` statement. Optional in the `Next` statement. Variable. Used to iterate through the elements of the collection.
`datatype`	Optional if `Option Infer` is on (the default) or `element` is already declared; required if `Option Infer` is off and `element` isn't already declared. The data type of `element`.
`group`	Required. A variable with a type that's a collection type or Object. Refers to the collection over which the `statements` are to be repeated.
`statements`	Optional. One or more statements between `For Each` and `Next` that run on each item in `group`.
`Continue For`	Optional. Transfers control to the start of the `For Each` loop.
`Exit For`	Optional. Transfers control out of the `For Each` loop.
`Next`	Required. Terminates the definition of the `For Each` loop.

Simple Example

Use a `For Each` ... `Next` loop when you want to repeat a set of statements for each element of a collection or array.

TIP

A For...Next Statement works well when you can associate each iteration of a loop with a control variable and determine that variable's initial and final values. However, when you are dealing with a collection, the concept of initial and final values isn't meaningful, and you don't necessarily know how many elements the collection has. In this kind of case, a `For Each` ... `Next` loop is often a better choice.

In the following example, the `For Each` ... `Next` statement iterates through all the elements of a List collection.

```
' Create a list of strings by using a
' collection initializer.
Dim lst As New List(Of String) _
    From {"abc", "def", "ghi"}

' Iterate through the list.
For Each item As String In lst
    Debug.Write(item & " ")
Next
Debug.WriteLine("")
'Output: abc def ghi
```

For more examples, see Collections and Arrays.

Nested Loops

You can nest `For Each` loops by putting one loop within another.

The following example demonstrates nested `For Each` ... `Next` structures.

```
' Create lists of numbers and letters
' by using array initializers.
Dim numbers() As Integer = {1, 4, 7}
Dim letters() As String = {"a", "b", "c"}

' Iterate through the list by using nested loops.
For Each number As Integer In numbers
    For Each letter As String In letters
        Debug.Write(number.ToString & letter & " ")
    Next
Next
Debug.WriteLine("")
'Output: 1a 1b 1c 4a 4b 4c 7a 7b 7c
```

When you nest loops, each loop must have a unique `element` variable.

You can also nest different kinds of control structures within each other. For more information, see Nested Control Structures.

Exit For and Continue For

The Exit For statement causes execution to exit the `For` ... `Next` loop and transfers control to the statement that follows the `Next` statement.

The `Continue For` statement transfers control immediately to the next iteration of the loop. For more information, see Continue Statement.

The following example shows how to use the `Continue For` and `Exit For` statements.

```
Dim numberSeq() As Integer =
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

For Each number As Integer In numberSeq
    ' If number is between 5 and 7, continue
    ' with the next iteration.
    If number >= 5 And number <= 8 Then
        Continue For
    End If

    ' Display the number.
    Debug.Write(number.ToString & " ")

    ' If number is 10, exit the loop.
    If number = 10 Then
        Exit For
    End If
Next
Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

You can put any number of `Exit For` statements in a `For Each` loop. When used within nested `For Each` loops, `Exit For` causes execution to exit the innermost loop and transfers control to the next higher level of nesting.

`Exit For` is often used after an evaluation of some condition, for example, in an `If`...`Then`...`Else` structure. You might want to use `Exit For` for the following conditions:

- Continuing to iterate is unnecessary or impossible. This might be caused by an erroneous value or a termination request.
- An exception is caught in a `Try`...`Catch`...`Finally`. You might use `Exit For` at the end of the `Finally` block.
- There an endless loop, which is a loop that could run a large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop. For more information, see Do...Loop Statement.

Iterators

You use an *iterator* to perform a custom iteration over a collection. An iterator can be a function or a `Get` accessor. It uses a `Yield` statement to return each element of the collection one at a time.

You call an iterator by using a `For Each...Next` statement. Each iteration of the `For Each` loop calls the iterator. When a `Yield` statement is reached in the iterator, the expression in the `Yield` statement is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

The following example uses an iterator function. The iterator function has a `Yield` statement that's inside a For...Next loop. In the `ListEvenNumbers` method, each iteration of the `For Each` statement body creates a call to the iterator function, which proceeds to the next `Yield` statement.

```
Public Sub ListEvenNumbers()
    For Each number As Integer In EvenSequence(5, 18)
        Debug.Write(number & " ")
    Next
    Debug.WriteLine("")
    ' Output: 6 8 10 12 14 16 18
End Sub

Private Iterator Function EvenSequence(
ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

For more information, see Iterators, Yield Statement, and Iterator.

Technical Implementation

When a `For Each` ... `Next` statement runs, Visual Basic evaluates the collection only one time, before the loop starts. If your statement block changes `element` or `group`, these changes don't affect the iteration of the loop.

When all the elements in the collection have been successively assigned to `element`, the `For Each` loop stops and control passes to the statement following the `Next` statement.

If Option Infer is on (its default setting), the Visual Basic compiler can infer the data type of `element`. If it is off and `element` hasn't been declared outside the loop, you must declare it in the `For Each` statement. To declare the data type of `element` explicitly, use an `As` clause. Unless the data type of element is defined outside the `For Each` ... `Next` construct, its scope is the body of the loop. Note that you cannot declare `element` both outside and inside the loop.

You can optionally specify `element` in the `Next` statement. This improves the readability of your program, especially if you have nested `For Each` loops. You must specify the same variable as the one that appears in the corresponding `For Each` statement.

You might want to avoid changing the value of `element` inside a loop. Doing this can make it more difficult to read and debug your code. Changing the value of `group` doesn't affect the collection or its elements, which were determined when the loop was first entered.

When you're nesting loops, if a `Next` statement of an outer nesting level is encountered before the `Next` of an inner level, the compiler signals an error. However, the compiler can detect this overlapping error only if you specify `element` in every `Next` statement.

If your code depends on traversing a collection in a particular order, a `For Each` ... `Next` loop isn't the best choice, unless you know the characteristics of the enumerator object the collection exposes. The order of traversal isn't determined by Visual Basic, but by the MoveNext method of the enumerator object. Therefore, you might not be able to predict which element of the collection is the first to be returned in `element`, or which is the next to be returned after a given element. You might achieve more reliable results using a different loop structure, such as `For` ... `Next` or `Do` ... `Loop`.

The runtime must be able to convert the elements in `group` to `element`. The [`Option Strict`] statement controls whether both widening and narrowing conversions are allowed (`Option Strict` is off, its default

value), or whether only widening conversions are allowed (`Option Strict` is on). For more information, see Narrowing conversions.

The data type of `group` must be a reference type that refers to a collection or an array that's enumerable. Most commonly this means that `group` refers to an object that implements the IEnumerable interface of the `System.Collections` namespace or the IEnumerable<T> interface of the `System.Collections.Generic` namespace. `System.Collections.IEnumerable` defines the GetEnumerator method, which returns an enumerator object for the collection. The enumerator object implements the `System.Collections.IEnumerator` interface of the `System.Collections` namespace and exposes the Current property and the Reset and MoveNext methods. Visual Basic uses these to traverse the collection.

Narrowing Conversions

When `Option Strict` is set to `On`, narrowing conversions ordinarily cause compiler errors. In a `For Each` statement, however, conversions from the elements in `group` to `element` are evaluated and performed at run time, and compiler errors caused by narrowing conversions are suppressed.

In the following example, the assignment of `m` as the initial value for `n` doesn't compile when `Option Strict` is on because the conversion of a `Long` to an `Integer` is a narrowing conversion. In the `For Each` statement, however, no compiler error is reported, even though the assignment to `number` requires the same conversion from `Long` to `Integer`. In the `For Each` statement that contains a large number, a run-time error occurs when ToInteger is applied to the large number.

```
Option Strict On

Module Module1
    Sub Main()
        ' The assignment of m to n causes a compiler error when
        ' Option Strict is on.
        Dim m As Long = 987
        'Dim n As Integer = m

        ' The For Each loop requires the same conversion but
        ' causes no errors, even when Option Strict is on.
        For Each number As Integer In New Long() {45, 3, 987}
            Console.Write(number & " ")
        Next
        Console.WriteLine()
        ' Output: 45 3 987

        ' Here a run-time error is raised because 9876543210
        ' is too large for type Integer.
        'For Each number As Integer In New Long() {45, 3, 9876543210}
        '    Console.Write(number & " ")
        'Next

        Console.ReadKey()
    End Sub
End Module
```

IEnumerator Calls

When execution of a `For Each` ... `Next` loop starts, Visual Basic verifies that `group` refers to a valid collection object. If not, it throws an exception. Otherwise, it calls the MoveNext method and the Current property of the enumerator object to return the first element. If `MoveNext` indicates that there is no next element, that is, if the collection is empty, the `For Each` loop stops and control passes to the statement following the `Next` statement. Otherwise, Visual Basic sets `element` to the first element and runs the statement block.

Each time Visual Basic encounters the `Next` statement, it returns to the `For Each` statement. Again it calls `MoveNext` and `Current` to return the next element, and again it either runs the block or stops the loop

depending on the result. This process continues until `MoveNext` indicates that there is no next element or an `Exit For` statement is encountered.

Modifying the Collection. The enumerator object returned by GetEnumerator normally doesn't let you change the collection by adding, deleting, replacing, or reordering any elements. If you change the collection after you have initiated a `For Each` ... `Next` loop, the enumerator object becomes invalid, and the next attempt to access an element causes an InvalidOperationException exception.

However, this blocking of modification isn't determined by Visual Basic, but rather by the implementation of the IEnumerable interface. It is possible to implement `IEnumerable` in a way that allows for modification during iteration. If you are considering doing such dynamic modification, make sure that you understand the characteristics of the `IEnumerable` implementation on the collection you are using.

Modifying Collection Elements. The Current property of the enumerator object is ReadOnly, and it returns a local copy of each collection element. This means that you cannot modify the elements themselves in a `For Each` ... `Next` loop. Any modification you make affects only the local copy from `Current` and isn't reflected back into the underlying collection. However, if an element is a reference type, you can modify the members of the instance to which it points. The following example modifies the `BackColor` member of each `thisControl` element. You cannot, however, modify `thisControl` itself.

```
Sub lightBlueBackground(ByVal thisForm As System.Windows.Forms.Form)
    For Each thisControl As System.Windows.Forms.Control In thisForm.Controls
        thisControl.BackColor = System.Drawing.Color.LightBlue
    Next thisControl
End Sub
```

The previous example can modify the `BackColor` member of each `thisControl` element, although it cannot modify `thisControl` itself.

Traversing Arrays. Because the Array class implements the IEnumerable interface, all arrays expose the GetEnumerator method. This means that you can iterate through an array with a `For Each` ... `Next` loop. However, you can only read the array elements. You cannot change them.

Example

The following example lists all the folders in the C:\ directory by using the DirectoryInfo class.

```
Dim dInfo As New System.IO.DirectoryInfo("c:\")
For Each dir As System.IO.DirectoryInfo In dInfo.GetDirectories()
    Debug.WriteLine(dir.Name)
Next
```

Example

The following example illustrates a procedure for sorting a collection. The example sorts instances of a `Car` class that are stored in a List<T>. The `Car` class implements the IComparable<T> interface, which requires that the CompareTo method be implemented.

Each call to the CompareTo method makes a single comparison that's used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the Sort method of the

List<T> causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```
Public Sub ListCars()

    ' Create some new cars.
    Dim cars As New List(Of Car) From
    {
        New Car With {.Name = "car1", .Color = "blue", .Speed = 20},
        New Car With {.Name = "car2", .Color = "red", .Speed = 50},
        New Car With {.Name = "car3", .Color = "green", .Speed = 10},
        New Car With {.Name = "car4", .Color = "blue", .Speed = 50},
        New Car With {.Name = "car5", .Color = "blue", .Speed = 30},
        New Car With {.Name = "car6", .Color = "red", .Speed = 60},
        New Car With {.Name = "car7", .Color = "green", .Speed = 50}
    }

    ' Sort the cars by color alphabetically, and then by speed
    ' in descending order.
    cars.Sort()

    ' View all of the cars.
    For Each thisCar As Car In cars
        Debug.Write(thisCar.Color.PadRight(5) & " ")
        Debug.Write(thisCar.Speed.ToString & " ")
        Debug.Write(thisCar.Name)
        Debug.WriteLine("")
    Next

    ' Output:
    '  blue  50 car4
    '  blue  30 car5
    '  blue  20 car1
    '  green 50 car7
    '  green 10 car3
    '  red   60 car6
    '  red   50 car2
End Sub

Public Class Car
    Implements IComparable(Of Car)

    Public Property Name As String
    Public Property Speed As Integer
    Public Property Color As String

    Public Function CompareTo(ByVal other As Car) As Integer _
        Implements System.IComparable(Of Car).CompareTo
        ' A call to this method makes a single comparison that is
        ' used for sorting.

        ' Determine the relative order of the objects being compared.
        ' Sort by color alphabetically, and then by speed in
        ' descending order.

        ' Compare the colors.
        Dim compare As Integer
        compare = String.Compare(Me.Color, other.Color, True)

        ' If the colors are the same, compare the speeds.
        If compare = 0 Then
            compare = Me.Speed.CompareTo(other.Speed)

            ' Use descending order for speed.
            compare = -compare
        End If

        Return compare
    End Function
```

```
End Class
```

See also

- Collections
- For...Next Statement
- Loop Structures
- While...End While Statement
- Do...Loop Statement
- Widening and Narrowing Conversions
- Object Initializers: Named and Anonymous Types
- Collection Initializers
- Arrays

For...Next Statement (Visual Basic)

8/27/2019 • 9 minutes to read • Edit Online

Repeats a group of statements a specified number of times.

Syntax

```
For counter [ As datatype ] = start To end [ Step step ]
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ counter ]
```

Parts

PART	DESCRIPTION
`counter`	Required in the `For` statement. Numeric variable. The control variable for the loop. For more information, see Counter Argument later in this topic.
`datatype`	Optional. Data type of `counter`. For more information, see Counter Argument later in this topic.
`start`	Required. Numeric expression. The initial value of `counter`.
`end`	Required. Numeric expression. The final value of `counter`.
`step`	Optional. Numeric expression. The amount by which `counter` is incremented each time through the loop.
`statements`	Optional. One or more statements between `For` and `Next` that run the specified number of times.
`Continue For`	Optional. Transfers control to the next loop iteration.
`Exit For`	Optional. Transfers control out of the `For` loop.
`Next`	Required. Terminates the definition of the `For` loop.

NOTE

The `To` keyword is used in this statement to specify the range for the counter. You can also use this keyword in the Select...Case Statement and in array declarations. For more information about array declarations, see Dim Statement.

Simple Examples

You use a `For`...`Next` structure when you want to repeat a set of statements a set number of times.

In the following example, the `index` variable starts with a value of 1 and is incremented with each iteration of the loop, ending after the value of `index` reaches 5.

```
For index As Integer = 1 To 5
    Debug.Write(index.ToString & " ")
Next
Debug.WriteLine("")
' Output: 1 2 3 4 5
```

In the following example, the `number` variable starts at 2 and is reduced by 0.25 on each iteration of the loop, ending after the value of `number` reaches 0. The `Step` argument of `-.25` reduces the value by 0.25 on each iteration of the loop.

```
For number As Double = 2 To 0 Step -0.25
    Debug.Write(number.ToString & " ")
Next
Debug.WriteLine("")
' Output: 2 1.75 1.5 1.25 1 0.75 0.5 0.25 0
```

TIP

A While...End While Statement or Do...Loop Statement works well when you don't know in advance how many times to run the statements in the loop. However, when you expect to run the loop a specific number of times, a `For`...`Next` loop is a better choice. You determine the number of iterations when you first enter the loop.

Nesting Loops

You can nest `For` loops by putting one loop within another. The following example demonstrates nested `For`...`Next` structures that have different step values. The outer loop creates a string for every iteration of the loop. The inner loop decrements a loop counter variable for every iteration of the loop.

```
For indexA = 1 To 3
    ' Create a new StringBuilder, which is used
    ' to efficiently build strings.
    Dim sb As New System.Text.StringBuilder()

    ' Append to the StringBuilder every third number
    ' from 20 to 1 descending.
    For indexB = 20 To 1 Step -3
        sb.Append(indexB.ToString)
        sb.Append(" ")
    Next indexB

    ' Display the line.
    Debug.WriteLine(sb.ToString)
Next indexA
' Output:
'  20 17 14 11 8 5 2
'  20 17 14 11 8 5 2
'  20 17 14 11 8 5 2
```

When nesting loops, each loop must have a unique `counter` variable.

You can also nest different kinds control structures within each other. For more information, see Nested Control Structures.

Exit For and Continue For

The `Exit For` statement immediately exits the `For` ... `Next` loop and transfers control to the statement that follows the `Next` statement.

The `Continue For` statement transfers control immediately to the next iteration of the loop. For more information, see Continue Statement.

The following example illustrates the use of the `Continue For` and `Exit For` statements.

```
For index As Integer = 1 To 100000
    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 AndAlso index <= 8 Then
        Continue For
    End If

    ' Display the index.
    Debug.Write(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit For
    End If
Next
Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

You can put any number of `Exit For` statements in a `For` ... `Next` loop. When used within nested `For` ... `Next` loops, `Exit For` exits the innermost loop and transfers control to the next higher level of nesting.

`Exit For` is often used after you evaluate some condition (for example, in an `If` ... `Then` ... `Else` structure). You might want to use `Exit For` for the following conditions:

- Continuing to iterate is unnecessary or impossible. An erroneous value or a termination request might create this condition.
- A `Try` ... `Catch` ... `Finally` statement catches an exception. You might use `Exit For` at the end of the `Finally` block.
- You have an endless loop, which is a loop that could run a large or even infinite number of times. If you detect such a condition, you can use `Exit For` to escape the loop. For more information, see Do...Loop Statement.

Technical Implementation

When a `For` ... `Next` loop starts, Visual Basic evaluates `start`, `end`, and `step`. Visual Basic evaluates these values only at this time and then assigns `start` to `counter`. Before the statement block runs, Visual Basic compares `counter` to `end`. If `counter` is already larger than the `end` value (or smaller if `step` is negative), the `For` loop ends and control passes to the statement that follows the `Next` statement. Otherwise, the statement block runs.

Each time Visual Basic encounters the `Next` statement, it increments `counter` by `step` and returns to the `For` statement. Again it compares `counter` to `end`, and again it either runs the block or exits the loop, depending on the result. This process continues until `counter` passes `end` or an `Exit For` statement is

encountered.

The loop doesn't stop until `counter` has passed `end`. If `counter` is equal to `end`, the loop continues. The comparison that determines whether to run the block is `counter` <= `end` if `step` is positive and `counter` >= `end` if `step` is negative.

If you change the value of `counter` while inside a loop, your code might be more difficult to read and debug. Changing the value of `start`, `end`, or `step` doesn't affect the iteration values that were determined when the loop was first entered.

If you nest loops, the compiler signals an error if it encounters the `Next` statement of an outer nesting level before the `Next` statement of an inner level. However, the compiler can detect this overlapping error only if you specify `counter` in every `Next` statement.

Step Argument

The value of `step` can be either positive or negative. This parameter determines loop processing according to the following table:

STEP VALUE	LOOP EXECUTES IF
Positive or zero	`counter` <= `end`
Negative	`counter` >= `end`

The default value of `step` is 1.

Counter Argument

The following table indicates whether `counter` defines a new local variable that's scoped to the entire `For...Next` loop. This determination depends on whether `datatype` is present and whether `counter` is already defined.

IS `DATATYPE` PRESENT?	IS `COUNTER` ALREADY DEFINED?	RESULT (WHETHER `COUNTER` DEFINES A NEW LOCAL VARIABLE THAT'S SCOPED TO THE ENTIRE `FOR...NEXT` LOOP)
No	Yes	No, because `counter` is already defined. If the scope of `counter` isn't local to the procedure, a compile-time warning occurs.
No	No	Yes. The data type is inferred from the `start`, `end`, and `step` expressions. For information about type inference, see Option Infer Statement and Local Type Inference.
Yes	Yes	Yes, but only if the existing `counter` variable is defined outside the procedure. That variable remains separate. If the scope of the existing `counter` variable is local to the procedure, a compile-time error occurs.
Yes	No	Yes.

The data type of `counter` determines the type of the iteration, which must be one of the following types:

- A `Byte`, `SByte`, `UShort`, `Short`, `UInteger`, `Integer`, `ULong`, `Long`, `Decimal`, `Single`, or `Double`.
- An enumeration that you declare by using an Enum Statement.
- An `Object`.
- A type `T` that has the following operators, where `B` is a type that can be used in a `Boolean` expression.

  ```
  Public Shared Operator >= (op1 As T, op2 As T) As B
  ```

  ```
  Public Shared Operator <= (op1 As T, op2 As T) As B
  ```

  ```
  Public Shared Operator - (op1 As T, op2 As T) As T
  ```

  ```
  Public Shared Operator + (op1 As T, op2 As T) As T
  ```

You can optionally specify the `counter` variable in the `Next` statement. This syntax improves the readability of your program, especially if you have nested `For` loops. You must specify the variable that appears in the corresponding `For` statement.

The `start`, `end`, and `step` expressions can evaluate to any data type that widens to the type of `counter`. If you use a user-defined type for `counter`, you might have to define the `CType` conversion operator to convert the types of `start`, `end`, or `step` to the type of `counter`.

Example

The following example removes all elements from a generic list. Instead of a For Each...Next Statement, the example shows a `For` ... `Next` statement that iterates in descending order. The example uses this technique because the `removeAt` method causes elements after the removed element to have a lower index value.

```
Dim lst As New List(Of Integer) From {10, 20, 30, 40}

For index As Integer = lst.Count - 1 To 0 Step -1
    lst.RemoveAt(index)
Next

Debug.WriteLine(lst.Count.ToString)
' Output: 0
```

Example

The following example iterates through an enumeration that's declared by using an Enum Statement.

```
Public Enum Mammals
    Buffalo
    Gazelle
    Mongoose
    Rhinoceros
    Whale
End Enum


Public Sub ListSomeMammals()
    For mammal As Mammals = Mammals.Gazelle To Mammals.Rhinoceros
        Debug.Write(mammal.ToString & " ")
    Next
    Debug.WriteLine("")
    ' Output: Gazelle Mongoose Rhinoceros
End Sub
```

Example

In the following example, the statement parameters use a class that has operator overloads for the `+`, `-`, `>=`, and `<=` operators.

```
Private Class Distance
    Public Property Number() As Double

    Public Sub New(ByVal number As Double)
        Me.Number = number
    End Sub

    ' Define operator overloads to support For...Next statements.
    Public Shared Operator +(ByVal op1 As Distance, ByVal op2 As Distance) As Distance
        Return New Distance(op1.Number + op2.Number)
    End Operator

    Public Shared Operator -(ByVal op1 As Distance, ByVal op2 As Distance) As Distance
        Return New Distance(op1.Number - op2.Number)
    End Operator

    Public Shared Operator >=(ByVal op1 As Distance, ByVal op2 As Distance) As Boolean
        Return (op1.Number >= op2.Number)
    End Operator

    Public Shared Operator <=(ByVal op1 As Distance, ByVal op2 As Distance) As Boolean
        Return (op1.Number <= op2.Number)
    End Operator
End Class


Public Sub ListDistances()
    Dim distFrom As New Distance(10)
    Dim distTo As New Distance(25)
    Dim distStep As New Distance(4)

    For dist As Distance = distFrom To distTo Step distStep
        Debug.Write(dist.Number.ToString & " ")
    Next
    Debug.WriteLine("")

    ' Output: 10 14 18 22
End Sub
```

See also

- List<T>
- Loop Structures
- While...End While Statement
- Do...Loop Statement
- Nested Control Structures
- Exit Statement
- Collections

Function Statement (Visual Basic)

4/26/2019 • 8 minutes to read • Edit Online

Declares the name, parameters, and code that define a `Function` procedure.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ proceduremodifiers ] [ Shared ] [ Shadows ] [ Async | Iterator ]
Function name [ (Of typeparamlist) ] [ (parameterlist) ] [ As returntype ] [ Implements implementslist | Handles eventlist ]
    [ statements ]
    [ Exit Function ]
    [ statements ]
End Function
```

Parts

- `attributelist`

  Optional. See Attribute List.

- `accessmodifier`

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - Protected Friend
  - Private Protected

  See Access levels in Visual Basic.

- `proceduremodifiers`

  Optional. Can be one of the following:

  - Overloads
  - Overrides
  - Overridable
  - NotOverridable
  - MustOverride
  - `MustOverride Overrides`
  - `NotOverridable Overrides`

- `Shared`

  Optional. See Shared.

- `Shadows`

  Optional. See Shadows.

- `Async`

  Optional. See Async.

- `Iterator`

  Optional. See Iterator.

- `name`

  Required. Name of the procedure. See Declared Element Names.

- `typeparamlist`

  Optional. List of type parameters for a generic procedure. See Type List.

- `parameterlist`

  Optional. List of local variable names representing the parameters of this procedure. See Parameter List.

- `returntype`

  Required if `Option Strict` is `On`. Data type of the value returned by this procedure.

- `Implements`

  Optional. Indicates that this procedure implements one or more `Function` procedures, each one defined in an interface implemented by this procedure's containing class or structure. See Implements Statement.

- `implementslist`

  Required if `Implements` is supplied. List of `Function` procedures being implemented.

  `implementedprocedure [ , implementedprocedure ... ]`

  Each `implementedprocedure` has the following syntax and parts:

  `interface.definedname`

PART	DESCRIPTION
`interface`	Required. Name of an interface implemented by this procedure's containing class or structure.
`definedname`	Required. Name by which the procedure is defined in `interface`.

- `Handles`

  Optional. Indicates that this procedure can handle one or more specific events. See Handles.

- `eventlist`

Required if `Handles` is supplied. List of events this procedure handles.

`eventspecifier [ , eventspecifier ... ]`

Each `eventspecifier` has the following syntax and parts:

`eventvariable.event`

PART	DESCRIPTION
`eventvariable`	Required. Object variable declared with the data type of the class or structure that raises the event.
`event`	Required. Name of the event this procedure handles.

- `statements`

  Optional. Block of statements to be executed within this procedure.

- `End Function`

  Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Each procedure, in turn, is declared within a class, a structure, or a module that is referred to as the containing class, structure, or module.

To return a value to the calling code, use a `Function` procedure; otherwise, use a `Sub` procedure.

Defining a Function

You can define a `Function` procedure only at the module level. Therefore, the declaration context for a function must be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see Declaration Contexts and Default Access Levels.

`Function` procedures default to public access. You can adjust their access levels with the access modifiers.

A `Function` procedure can declare the data type of the value that the procedure returns. You can specify any data type or the name of an enumeration, a structure, a class, or an interface. If you don't specify the `returntype` parameter, the procedure returns `Object`.

If this procedure uses the `Implements` keyword, the containing class or structure must also have an `Implements` statement that immediately follows its `Class` or `Structure` statement. The `Implements` statement must include each interface that's specified in `implementslist`. However, the name by which an interface defines the `Function` (in `definedname`) doesn't need to match the name of this procedure (in `name`).

> **NOTE**
>
> You can use lambda expressions to define function expressions inline. For more information, see Function Expression and Lambda Expressions.

Returning from a Function

When the `Function` procedure returns to the calling code, execution continues with the statement that follows the statement that called the procedure.

To return a value from a function, you can either assign the value to the function name or include it in a `Return` statement.

The `Return` statement simultaneously assigns the return value and exits the function, as the following example shows.

```
Function MyFunction(ByVal j As Integer) As Double
    Return 3.87 * j
End Function
```

The following example assigns the return value to the function name `myFunction` and then uses the `Exit Function` statement to return.

```
Function MyFunction(ByVal j As Integer) As Double
    MyFunction = 3.87 * j
    Exit Function
End Function
```

The `Exit Function` and `Return` statements cause an immediate exit from a `Function` procedure. Any number of `Exit Function` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Function` and `Return` statements.

If you use `Exit Function` without assigning a value to `name`, the procedure returns the default value for the data type that's specified in `returntype`. If `returntype` isn't specified, the procedure returns `Nothing`, which is the default value for `Object`.

Calling a Function

You call a `Function` procedure by using the procedure name, followed by the argument list in parentheses, in an expression. You can omit the parentheses only if you aren't supplying any arguments. However, your code is more readable if you always include the parentheses.

You call a `Function` procedure the same way that you call any library function such as `Sqrt`, `Cos`, or `ChrW`.

You can also call a function by using the `Call` keyword. In that case, the return value is ignored. Use of the `Call` keyword isn't recommended in most cases. For more information, see Call Statement.

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, you shouldn't use a `Function` procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Async Functions

The *Async* feature allows you to invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a function with the Async modifier, you can use the Await operator in the function. When control reaches an `Await` expression in the `Async` function, control returns to the caller, and progress in the function is suspended until the awaited task completes. When the task is complete, execution can resume in the function.

> **NOTE**
> An `Async` procedure returns to the caller when either it encounters the first awaited object that's not yet complete, or it gets to the end of the `Async` procedure, whichever occurs first.

An `Async` function can have a return type of Task<TResult> or Task. An example of an `Async` function that has a return type of Task<TResult> is provided below.

An `Async` function cannot declare any ByRef parameters.

A Sub Statement can also be marked with the `Async` modifier. This is primarily used for event handlers, where a value cannot be returned. An `Async` `Sub` procedure can't be awaited, and the caller of an `Async` `Sub` procedure can't catch exceptions that are thrown by the `Sub` procedure.

For more information about `Async` functions, see Asynchronous Programming with Async and Await, Control Flow in Async Programs, and Async Return Types.

Iterator Functions

An *iterator* function performs a custom iteration over a collection, such as a list or array. An iterator function uses the Yield statement to return each element one at a time. When a Yield statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You call an iterator from client code by using a For Each...Next statement.

The return type of an iterator function can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

Example

The following example uses the `Function` statement to declare the name, parameters, and code that form the body of a `Function` procedure. The `ParamArray` modifier enables the function to accept a variable number of arguments.

```
Public Function CalcSum(ByVal ParamArray args() As Double) As Double
    CalcSum = 0
    If args.Length <= 0 Then Exit Function
    For i As Integer = 0 To UBound(args, 1)
        CalcSum += args(i)
    Next i
End Function
```

Example

The following example invokes the function declared in the preceding example.

```
Module Module1

    Sub Main()
        ' In the following function call, CalcSum's local variables
        ' are assigned the following values: args(0) = 4, args(1) = 3,
        ' and so on. The displayed sum is 10.
        Dim returnedValue As Double = CalcSum(4, 3, 2, 1)
        Console.WriteLine("Sum: " & returnedValue)
        ' Parameter args accepts zero or more arguments. The sum
        ' displayed by the following statements is 0.
        returnedValue = CalcSum()
        Console.WriteLine("Sum: " & returnedValue)
    End Sub

    Public Function CalcSum(ByVal ParamArray args() As Double) As Double
        CalcSum = 0
        If args.Length <= 0 Then Exit Function
        For i As Integer = 0 To UBound(args, 1)
            CalcSum += args(i)
        Next i
    End Function

End Module
```

Example

In the following example, `DelayAsync` is an `Async` `Function` that has a return type of Task<TResult>. `DelayAsync` has a `Return` statement that returns an integer. Therefore the function declaration of `DelayAsync` needs to have a return type of `Task(Of Integer)`. Because the return type is `Task(Of Integer)`, the evaluation of the `Await` expression in `DoSomethingAsync` produces an integer. This is demonstrated in this statement: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an `Async Sub` procedure. Because `DoSomethingAsync` is an `Async` function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement demonstrates: `Await DoSomethingAsync()`. The `startButton_Click` `Sub` procedure must be defined with the `Async` modifier because it has an `Await` expression.

```
' Imports System.Diagnostics
' Imports System.Threading.Tasks

' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

'  Output:
'    Result: 5
```

See also

- Sub Statement
- Function Procedures
- Parameter List
- Dim Statement
- Call Statement
- Of
- Parameter Arrays
- How to: Use a Generic Class
- Troubleshooting Procedures
- Lambda Expressions
- Function Expression

Get Statement

4/28/2019 • 3 minutes to read • Edit Online

Declares a `Get` property procedure used to retrieve the value of a property.

Syntax

```
[ <attributelist> ] [ accessmodifier ] Get()
    [ statements ]
End Get
```

Parts

TERM	DEFINITION
`attributelist`	Optional. See Attribute List.
`accessmodifier`	Optional on at most one of the `Get` and `Set` statements in this property. Can be one of the following: - Protected - Friend - Private - `Protected Friend` See Access levels in Visual Basic.
`statements`	Optional. One or more statements that run when the `Get` property procedure is called.
`End Get`	Required. Terminates the definition of the `Get` property procedure.

Remarks

Every property must have a `Get` property procedure unless the property is marked `WriteOnly`. The `Get` procedure is used to return the current value of the property.

Visual Basic automatically calls a property's `Get` procedure when an expression requests the property's value.

The body of the property declaration can contain only the property's `Get` and `Set` procedures between the Property Statement and the `End Property` statement. It cannot store anything other than those procedures. In particular, it cannot store the property's current value. You must store this value outside the property, because if you store it inside either of the property procedures, the other property procedure cannot access it. The usual approach is to store the value in a Private variable declared at the same level as the property. You must define a `Get` procedure inside the property to which it applies.

The `Get` procedure defaults to the access level of its containing property unless you use `accessmodifier` in the `Get` statement.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Get` procedure `Private`, but not `Public`.

  If you are defining a `ReadOnly` property, the `Get` procedure represents the entire property. You cannot declare a different access level for `Get`, because that would set two access levels for the property.

- **Return Type.** The Property Statement can declare the data type of the value it returns. The `Get` procedure automatically returns that data type. You can specify any data type or the name of an enumeration, structure, class, or interface.

  If the `Property` statement does not specify `returntype`, the procedure returns `Object`.

Behavior

- **Returning from a Procedure.** When the `Get` procedure returns to the calling code, execution continues within the statement that requested the property value.

  `Get` property procedures can return a value using either the Return Statement or by assigning the return value to the property name. For more information, see "Return Value" in Function Statement.

  The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

- **Return Value.** To return a value from a `Get` procedure, you can either assign the value to the property name or include it in a Return Statement. The `Return` statement simultaneously assigns the `Get` procedure return value and exits the procedure.

  If you use `Exit Property` without assigning a value to the property name, the `Get` procedure returns the default value for the property's data type. For more information, see "Return Value" in Function Statement.

  The following example illustrates two ways the read-only property `quoteForTheDay` can return the value held in the private variable `quoteValue`.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        QuoteForTheDay = quoteValue
        Exit Property
    End Get
End Property
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

Example

The following example uses the `Get` statement to return the value of a property.

```
Class propClass
    ' Define a private local variable to store the property value.
    Private currentTime As String
    ' Define the read-only property.
    Public ReadOnly Property DateAndTime() As String
        Get
            ' The Get procedure is called automatically when the
            ' value of the property is retrieved.
            currentTime = CStr(Now)
            ' Return the date and time As a string.
            Return currentTime
        End Get
    End Property
End Class
```

See also

- Set Statement
- Property Statement
- Exit Statement
- Objects and Classes
- Walkthrough: Defining Classes

GoTo Statement

8/22/2019 • 2 minutes to read • Edit Online

Branches unconditionally to a specified line in a procedure.

Syntax

```
GoTo line
```

Part

`line`
Required. Any line label.

Remarks

The `GoTo` statement can branch only to lines in the procedure in which it appears. The line must have a line label that `GoTo` can refer to. For more information, see How to: Label Statements.

> **NOTE**
> `GoTo` statements can make code difficult to read and maintain. Whenever possible, use a control structure instead. For more information, see Control Flow.

You cannot use a `GoTo` statement to branch from outside a `For` ... `Next`, `For Each` ... `Next`, `SyncLock` ... `End SyncLock`, `Try` ... `Catch` ... `Finally`, `With` ... `End With`, or `Using` ... `End Using` construction to a label inside.

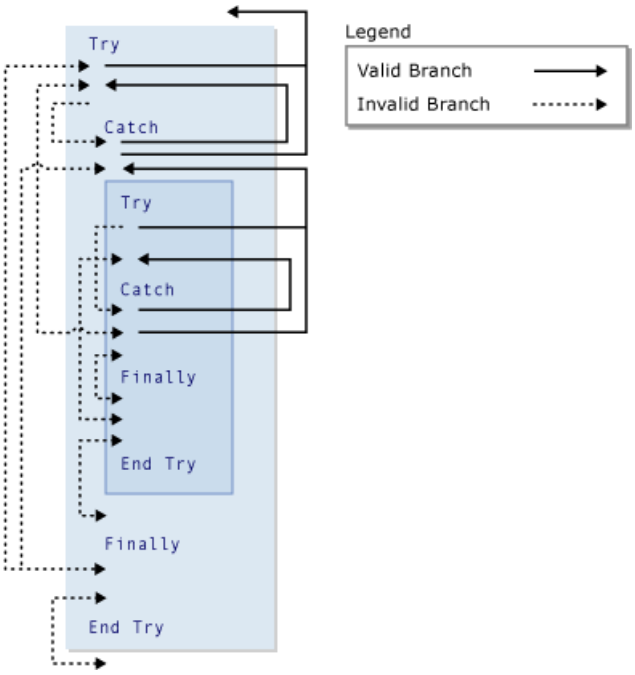
Branching and Try Constructions

Within a `Try` ... `Catch` ... `Finally` construction, the following rules apply to branching with the `GoTo` statement.

BLOCK OR REGION	BRANCHING IN FROM OUTSIDE	BRANCHING OUT FROM INSIDE
`Try` block	Only from a `Catch` block of the same construction [1]	Only to outside the whole construction
`Catch` block	Never allowed	Only to outside the whole construction, or to the `Try` block of the same construction [1]
`Finally` block	Never allowed	Never allowed

[1] If one `Try` ... `Catch` ... `Finally` construction is nested within another, a `Catch` block can branch into the `Try` block at its own nesting level, but not into any other `Try` block. A nested `Try` ... `Catch` ... `Finally` construction must be contained completely in a `Try` or `Catch` block of the construction within which it is nested.

The following illustration shows one `Try` construction nested within another. Various branches among the blocks of the two constructions are indicated as valid or invalid.



Example

The following example uses the `GoTo` statement to branch to line labels in a procedure.

```
    Sub GoToStatementDemo()
        Dim number As Integer = 1
        Dim sampleString As String
        ' Evaluate number and branch to appropriate label.
        If number = 1 Then GoTo Line1 Else GoTo Line2
Line1:
        sampleString = "Number equals 1"
        GoTo LastLine
Line2:
        ' The following statement never gets executed because number = 1.
        sampleString = "Number equals 2"
LastLine:
        ' Write "Number equals 1" in the Debug window.
        Debug.WriteLine(sampleString)
    End Sub
```

See also

- Do...Loop Statement
- For...Next Statement
- For Each...Next Statement
- If...Then...Else Statement
- Select...Case Statement
- Try...Catch...Finally Statement
- While...End While Statement
- With...End With Statement

If...Then...Else Statement (Visual Basic)

8/27/2019 • 5 minutes to read • Edit Online

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

```
' Multiline syntax:
If condition [ Then ]
    [ statements ]
[ ElseIf elseifcondition [ Then ]
    [ elseifstatements ] ]
[ Else
    [ elsestatements ] ]
End If

' Single-line syntax:
If condition Then [ statements ] [ Else [ elsestatements ] ]
```

Quick links to example code

This article includes several examples that illustrate uses of the `If` ... `Then` ... `Else` statement:

- Multiline syntax example
- Nested syntax example
- Single-line syntax example

Parts

`condition`
Required. Expression. Must evaluate to `True` or `False`, or to a data type that is implicitly convertible to `Boolean`.

If the expression is a Nullable `Boolean` variable that evaluates to Nothing, the condition is treated as if the expression is `False` and the `Else` block is executed.

`Then`
Required in the single-line syntax; optional in the multiline syntax.

`statements`
Optional. One or more statements following `If` ... `Then` that are executed if `condition` evaluates to `True`.

`elseifcondition`
Required if `ElseIf` is present. Expression. Must evaluate to `True` or `False`, or to a data type that is implicitly convertible to `Boolean`.

`elseifstatements`
Optional. One or more statements following `ElseIf` ... `Then` that are executed if `elseifcondition` evaluates to `True`.

`elsestatements`
Optional. One or more statements that are executed if no previous `condition` or `elseifcondition` expression

evaluates to `True`.

`End If`
Terminates the multiline version of `If` ... `Then` ... `Else` block.

Remarks

Multiline syntax

When an `If` ... `Then` ... `Else` statement is encountered, `condition` is tested. If `condition` is `True`, the statements following `Then` are executed. If `condition` is `False`, each `ElseIf` statement (if there are any) is evaluated in order. When a `True` `elseifcondition` is found, the statements immediately following the associated `ElseIf` are executed. If no `elseifcondition` evaluates to `True`, or if there are no `ElseIf` statements, the statements following `Else` are executed. After executing the statements following `Then`, `ElseIf`, or `Else`, execution continues with the statement following `End If`.

The `ElseIf` and `Else` clauses are both optional. You can have as many `ElseIf` clauses as you want in an `If` ... `Then` ... `Else` statement, but no `ElseIf` clause can appear after an `Else` clause. `If` ... `Then` ... `Else` statements can be nested within each other.

In the multiline syntax, the `If` statement must be the only statement on the first line. The `ElseIf`, `Else`, and `End If` statements can be preceded only by a line label. The `If` ... `Then` ... `Else` block must end with an `End If` statement.

> **TIP**
> The Select...Case Statement might be more useful when you evaluate a single expression that has several possible values.

Single-Line syntax

You can use the single-line syntax for a single condition with code to execute if it's true. However, the multiple-line syntax provides more structure and flexibility and is easier to read, maintain, and debug.

What follows the `Then` keyword is examined to determine whether a statement is a single-line `If`. If anything other than a comment appears after `Then` on the same line, the statement is treated as a single-line `If` statement. If `Then` is absent, it must be the start of a multiple-line `If` ... `Then` ... `Else`.

In the single-line syntax, you can have multiple statements executed as the result of an `If` ... `Then` decision. All statements must be on the same line and be separated by colons.

Multiline syntax example

The following example illustrates the use of the multiline syntax of the `If` ... `Then` ... `Else` statement.

```
Module Multiline
    Public Sub Main()
        'Create a Random object to seed our starting value
        Dim randomizer As New Random()
        'set our variable
        Dim count As Integer = randomizer.Next(0, 5)

        Dim message As String

        'If count is zero, output will be no items
        If count = 0 Then
            message = "There are no items."
        'If count is 1, output will be "There is 1 item.".
        ElseIf count = 1 Then
            message = "There is 1 item."
        'If count is greater than 1, output will be "There are {count} items.", where {count} is replaced by
the value of count.
        Else
            message = $"There are {count} items."
        End If

        Console.WriteLine(message)
    End Sub
End Module
'This example displays output like the following:
' There are 4 items.
```

Nested syntax example

The following example contains nested `If` ... `Then` ... `Else` statements.

```
Module Nested
    Public Sub Main()
        ' Run the function as part of the WriteLine output.
        Console.WriteLine("Time Check is " & CheckIfTime() & ".")
    End Sub

    Private Function CheckIfTime() As Boolean
        ' Determine the current day of week and hour of day.
        Dim dayW As DayOfWeek = DateTime.Now.DayOfWeek
        Dim hour As Integer = DateTime.Now.Hour

        ' Return True if Wednesday from 2 to 3:59 P.M.,
        ' or if Thursday from noon to 12:59 P.M.
        If dayW = DayOfWeek.Wednesday Then
            If hour = 14 Or hour = 15 Then
                Return True
            Else
                Return False
            End If
        ElseIf dayW = DayOfWeek.Thursday Then
            If hour = 12 Then
                Return True
            Else
                Return False
            End If
        Else
            Return False
        End If
    End Function
End Module
'This example displays output like the following:
'Time Check is False.
```

Single-Line syntax example

The following example illustrates the use of the single-line syntax.

```
Module SingleLine
    Public Sub Main()

        'Create a Random object to seed our starting values
        Dim randomizer As New Random()

        Dim A As Integer = randomizer.Next(10, 20)
        Dim B As Integer = randomizer.Next(0, 20)
        Dim C As Integer = randomizer.Next(0, 5)

        'Let's display the initial values for comparison
        Console.WriteLine($"A value before If: {A}")
        Console.WriteLine($"B value before If: {B}")
        Console.WriteLine($"C value before If: {C}")

        ' If A > 10, execute the three colon-separated statements in the order
        ' that they appear
        If A > 10 Then A = A + 1 : B = B + A : C = C + B

        'If the condition is true, the values will be different
        Console.WriteLine($"A value after If: {A}")
        Console.WriteLine($"B value after If: {B}")
        Console.WriteLine($"C value after If: {C}")

    End Sub
End Module
'This example displays output like the following:
'A value before If: 11
'B value before If: 6
'C value before If: 3
'A value after If: 12
'B value after If: 18
'C value after If: 21
```

See also

- Choose
- Switch
- #If...Then...#Else Directives
- Select...Case Statement
- Nested Control Structures
- Decision Structures
- Logical and Bitwise Operators in Visual Basic
- If Operator

Implements Statement

4/2/2019 • 2 minutes to read • Edit Online

Specifies one or more interfaces, or interface members, that must be implemented in the class or structure definition in which it appears.

Syntax

```
Implements interfacename [, ...]
-or-
Implements interfacename.interfacemember [, ...]
```

Parts

`interfacename`
Required. An interface whose properties, procedures, and events are to be implemented by corresponding members in the class or structure.

`interfacemember`
Required. The member of an interface that is being implemented.

Remarks

An interface is a collection of prototypes representing the members (properties, procedures, and events) the interface encapsulates. Interfaces contain only the declarations for members; classes and structures implement these members. For more information, see Interfaces.

The `Implements` statement must immediately follow the `Class` or `Structure` statement.

When you implement an interface, you must implement all the members declared in the interface. Omitting any member is considered to be a syntax error. To implement an individual member, you specify the Implements keyword (which is separate from the `Implements` statement) when you declare the member in the class or structure. For more information, see Interfaces.

Classes can use Private implementations of properties and procedures, but these members are accessible only by casting an instance of the implementing class into a variable declared to be of the type of the interface.

Example

The following example shows how to use the `Implements` statement to implement members of an interface. It defines an interface named `ICustomerInfo` with an event, a property, and a procedure. The class `customerInfo` implements all the members defined in the interface.

```
Public Interface ICustomerInfo
    Event UpdateComplete()
    Property CustomerName() As String
    Sub UpdateCustomerStatus()
End Interface

Public Class customerInfo
    Implements ICustomerInfo
    ' Storage for the property value.
    Private customerNameValue As String
    Public Event UpdateComplete() Implements ICustomerInfo.UpdateComplete
    Public Property CustomerName() As String _
        Implements ICustomerInfo.CustomerName
        Get
            Return customerNameValue
        End Get
        Set(ByVal value As String)
            ' The value parameter is passed to the Set procedure
            ' when the contents of this property are modified.
            customerNameValue = value
        End Set
    End Property

    Public Sub UpdateCustomerStatus() _
        Implements ICustomerInfo.UpdateCustomerStatus
        ' Add code here to update the status of this account.
        ' Raise an event to indicate that this procedure is done.
        RaiseEvent UpdateComplete()
    End Sub
End Class
```

Note that the class `customerInfo` uses the `Implements` statement on a separate source code line to indicate that the class implements all the members of the `ICustomerInfo` interface. Then each member in the class uses the `Implements` keyword as part of its member declaration to indicate that it implements that interface member.

Example

The following two procedures show how you could use the interface implemented in the preceding example. To test the implementation, add these procedures to your project and call the `testImplements` procedure.

```
Public Sub TestImplements()
    ' This procedure tests the interface implementation by
    ' creating an instance of the class that implements ICustomerInfo.
    Dim cust As ICustomerInfo = New customerInfo()
    ' Associate an event handler with the event that is raised by
    ' the cust object.
    AddHandler cust.UpdateComplete, AddressOf HandleUpdateComplete
    ' Set the CustomerName Property
    cust.CustomerName = "Fred"
    ' Retrieve and display the CustomerName property.
    MsgBox("Customer name is: " & cust.CustomerName)
    ' Call the UpdateCustomerStatus procedure, which raises the
    ' UpdateComplete event.
    cust.UpdateCustomerStatus()
End Sub

Sub HandleUpdateComplete()
    ' This is the event handler for the UpdateComplete event.
    MsgBox("Update is complete.")
End Sub
```

See also

- Implements
- Interface Statement
- Interfaces

Imports Statement (.NET Namespace and Type)

8/22/2019 • 4 minutes to read • Edit Online

Enables type names to be referenced without namespace qualification.

Syntax

```
Imports [ aliasname = ] namespace
-or-
Imports [ aliasname = ] namespace.element
```

Parts

TERM	DEFINITION
`aliasname`	Optional. An *import alias* or name by which code can refer to `namespace` instead of the full qualification string. See Declared Element Names.
`namespace`	Required. The fully qualified name of the namespace being imported. Can be a string of namespaces nested to any level.
`element`	Optional. The name of a programming element declared in the namespace. Can be any container element.

Remarks

The `Imports` statement enables types that are contained in a given namespace to be referenced directly.

You can supply a single namespace name or a string of nested namespaces. Each nested namespace is separated from the next higher level namespace by a period (`.`), as the following example illustrates.

```
Imports System.Collections.Generic
```

Each source file can contain any number of `Imports` statements. These must follow any option declarations, such as the `Option Strict` statement, and they must precede any programming element declarations, such as `Module` or `Class` statements.

You can use `Imports` only at file level. This means the declaration context for importation must be a source file, and cannot be a namespace, class, structure, module, interface, procedure, or block.

Note that the `Imports` statement does not make elements from other projects and assemblies available to your project. Importing does not take the place of setting a reference. It only removes the need to qualify names that are already available to your project. For more information, see "Importing Containing Elements" in References to Declared Elements.

NOTE

You can define implicit `Imports` statements by using the References Page, Project Designer (Visual Basic). For more information, see How to: Add or Remove Imported Namespaces (Visual Basic).

Import Aliases

An *import alias* defines the alias for a namespace or type. Import aliases are useful when you need to use items with the same name that are declared in one or more namespaces. For more information and an example, see "Qualifying an Element Name" in References to Declared Elements.

You should not declare a member at module level with the same name as `aliasname`. If you do, the Visual Basic compiler uses `aliasname` only for the declared member and no longer recognizes it as an import alias.

Although the syntax used for declaring an import alias is like that used for importing an XML namespace prefix, the results are different. An import alias can be used as an expression in your code, whereas an XML namespace prefix can be used only in XML literals or XML axis properties as the prefix for a qualified element or attribute name.

Element Names

If you supply `element`, it must represent a *container element*, that is, a programming element that can contain other elements. Container elements include classes, structures, modules, interfaces, and enumerations.

The scope of the elements made available by an `Imports` statement depends on whether you specify `element`. If you specify only `namespace`, all uniquely named members of that namespace, and members of container elements within that namespace, are available without qualification. If you specify both `namespace` and `element`, only the members of that element are available without qualification.

Example

The following example returns all the folders in the C:\ directory by using the DirectoryInfo class.

The code has no `Imports` statements at the top of the file. Therefore, the `DirectoryInfo`, StringBuilder, and CrLf references are all fully qualified with the namespaces.

```
Public Function GetFolders() As String
    ' Create a new StringBuilder, which is used
    ' to efficiently build strings.
    Dim sb As New System.Text.StringBuilder

    Dim dInfo As New System.IO.DirectoryInfo("c:\")

    ' Obtain an array of directories, and iterate through
    ' the array.
    For Each dir As System.IO.DirectoryInfo In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(Microsoft.VisualBasic.ControlChars.CrLf)
    Next

    Return sb.ToString
End Function
```

Example

The following example includes `Imports` statements for the referenced namespaces. Therefore, the types do not have to be fully qualified with the namespaces.

```
' Place Imports statements at the top of your program.
Imports System.Text
Imports System.IO
Imports Microsoft.VisualBasic.ControlChars
```

```
Public Function GetFolders() As String
    Dim sb As New StringBuilder

    Dim dInfo As New DirectoryInfo("c:\")
    For Each dir As DirectoryInfo In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(CrLf)
    Next

    Return sb.ToString
End Function
```

Example

The following example includes `Imports` statements that create aliases for the referenced namespaces. The types are qualified with the aliases.

```
Imports systxt = System.Text
Imports sysio = System.IO
Imports ch = Microsoft.VisualBasic.ControlChars
```

```
Public Function GetFolders() As String
    Dim sb As New systxt.StringBuilder

    Dim dInfo As New sysio.DirectoryInfo("c:\")
    For Each dir As sysio.DirectoryInfo In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(ch.CrLf)
    Next

    Return sb.ToString
End Function
```

Example

The following example includes `Imports` statements that create aliases for the referenced types. Aliases are used to specify the types.

```
Imports strbld = System.Text.StringBuilder
Imports dirinf = System.IO.DirectoryInfo
```

```
Public Function GetFolders() As String
    Dim sb As New strbld

    Dim dInfo As New dirinf("c:\")
    For Each dir As dirinf In dInfo.GetDirectories()
        sb.Append(dir.Name)
        sb.Append(ControlChars.CrLf)
    Next

    Return sb.ToString
End Function
```

See also

- Namespace Statement
- Namespaces in Visual Basic
- References and the Imports Statement
- Imports Statement (XML Namespace)
- References to Declared Elements

Imports Statement (XML Namespace)

4/2/2019 • 4 minutes to read • Edit Online

Imports XML namespace prefixes for use in XML literals and XML axis properties.

Syntax

```
Imports <xmlns:xmlNamespacePrefix = "xmlNamespaceName">
```

Parts

`xmlNamespacePrefix`
Optional. The string by which XML elements and attributes can refer to `xmlNamespaceName`. If no `xmlNamespacePrefix` is supplied, the imported XML namespace is the default XML namespace. Must be a valid XML identifier. For more information, see Names of Declared XML Elements and Attributes.

`xmlNamespaceName`
Required. The string identifying the XML namespace being imported.

Remarks

You can use the `Imports` statement to define global XML namespaces that you can use with XML literals and XML axis properties, or as parameters passed to the `GetXmlNamespace` operator. (For information about using the `Imports` statement to import an alias that can be used where type names are used in your code, see Imports Statement (.NET Namespace and Type).) The syntax for declaring an XML namespace by using the `Imports` statement is identical to the syntax used in XML. Therefore, you can copy a namespace declaration from an XML file and use it in an `Imports` statement.

XML namespace prefixes are useful when you want to repeatedly create XML elements that are from the same namespace. The XML namespace prefix declared with the `Imports` statement is global in the sense that it is available to all code in the file. You can use it when you create XML element literals and when you access XML axis properties. For more information, see XML Element Literal and XML Axis Properties.

If you define a global XML namespace without a namespace prefix (for example, `Imports <xmlns="http://SomeNameSpace>"`), that namespace is considered the default XML namespace. The default XML namespace is used for any XML element literals or XML attribute axis properties that do not explicitly specify a namespace. The default namespace is also used if the specified namespace is the empty namespace (that is, `xmlns=""`). The default XML namespace does not apply to XML attributes in XML literals or to XML attribute axis properties that do not have a namespace.

XML namespaces that are defined in an XML literal, which are called *local XML namespaces*, take precedence over XML namespaces that are defined by the `Imports` statement as global. XML namespaces that are defined by the `Imports` statement take precedence over XML namespaces imported for a Visual Basic project. If an XML literal defines an XML namespace, that local namespace does not apply to embedded expressions.

Global XML namespaces follow the same scoping and definition rules as .NET Framework namespaces. As a result, you can include an `Imports` statement to define a global XML namespace anywhere you can import a .NET Framework namespace. This includes both code files and project-level imported namespaces. For information about project-level imported namespaces, see References Page, Project Designer (Visual Basic).

Each source file can contain any number of `Imports` statements. These must follow option declarations, such as the `Option Strict` statement, and they must precede programming element declarations, such as `Module` or `Class` statements.

Example

The following example imports a default XML namespace and an XML namespace identified with the prefix `ns`. It then creates XML literals that use both namespaces.

```
' Place Imports statements at the top of your program.
Imports <xmlns="http://DefaultNamespace">
Imports <xmlns:ns="http://NewNamespace">

Module Module1

  Sub Main()
    ' Create element by using the default global XML namespace.
    Dim inner = <innerElement/>

    ' Create element by using both the default global XML namespace
    ' and the namespace identified with the "ns" prefix.
    Dim outer = <ns:outer>
                  <ns:innerElement></ns:innerElement>
                  <siblingElement></siblingElement>
                  <%= inner %>
                </ns:outer>

    ' Display element to see its final form.
    Console.WriteLine(outer)
  End Sub

End Module
```

This code displays the following text:

```
<ns:outer xmlns="http://DefaultNamespace"
          xmlns:ns="http://NewNamespace">
  <ns:innerElement></ns:innerElement>
  <siblingElement></siblingElement>
  <innerElement />
</ns:outer>
```

Example

The following example imports the XML namespace prefix `ns`. It then creates an XML literal that uses the namespace prefix and displays the element's final form.

```
' Place Imports statements at the top of your program.
Imports <xmlns:ns="http://SomeNamespace">

Class TestClass1

    Shared Sub TestPrefix()
        ' Create test using a global XML namespace prefix.
        Dim inner2 = <ns:inner2/>

        Dim test =
        <ns:outer>
            <ns:middle xmlns:ns="http://NewNamespace">
                <ns:inner1/>
                <%= inner2 %>
            </ns:middle>
        </ns:outer>

        ' Display test to see its final form.
        Console.WriteLine(test)
    End Sub

End Class
```

This code displays the following text:

```
<ns:outer xmlns:ns="http://SomeNamespace">
  <ns:middle xmlns:ns="http://NewNamespace">
    <ns:inner1 />
    <inner2 xmlns="http://SomeNamespace" />
  </ns:middle>
</ns:outer>
```

Notice that the compiler converted the XML namespace prefix from a global prefix to a local prefix definition.

Example

The following example imports the XML namespace prefix `ns`. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name `ns:name`.

```
Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass4

    Shared Sub TestPrefix()
        Dim contact = <ns:contact>
                        <ns:name>Patrick Hines</ns:name>
                      </ns:contact>
        Console.WriteLine(contact.<ns:name>.Value)
    End Sub

End Class
```

This code displays the following text:

```
Patrick Hines
```

See also

- XML Element Literal
- XML Axis Properties

- Names of Declared XML Elements and Attributes
- GetXmlNamespace Operator

Inherits Statement

4/28/2019 • 2 minutes to read • Edit Online

Causes the current class or interface to inherit the attributes, variables, properties, procedures, and events from another class or set of interfaces.

Syntax

```
Inherits basetypenames
```

Parts

TERM	DEFINITION
`basetypenames`	Required. The name of the class from which this class derives. -or- The names of the interfaces from which this interface derives. Use commas to separate multiple names.

Remarks

If used, the `Inherits` statement must be the first non-blank, non-comment line in a class or interface definition. It should immediately follow the `Class` or `Interface` statement.

You can use `Inherits` only in a class or interface. This means the declaration context for an inheritance cannot be a source file, namespace, structure, module, procedure, or block.

Rules

- **Class Inheritance.** If a class uses the `Inherits` statement, you can specify only one base class.

  A class cannot inherit from a class nested within it.

- **Interface Inheritance.** If an interface uses the `Inherits` statement, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

  An interface cannot inherit from another interface with a more restrictive access level. For example, a `Public` interface cannot inherit from a `Friend` interface.

  An interface cannot inherit from an interface nested within it.

An example of class inheritance in the .NET Framework is the ArgumentException class, which inherits from the SystemException class. This provides to ArgumentException all the predefined properties and procedures required by system exceptions, such as the Message property and the ToString method.

An example of interface inheritance in the .NET Framework is the ICollection interface, which inherits from the

IEnumerable interface. This causes ICollection to inherit the definition of the enumerator required to traverse a collection.

Example

The following example uses the `Inherits` statement to show how a class named `thisClass` can inherit all the members of a base class named `anotherClass`.

```
Public Class thisClass
    Inherits anotherClass
    ' Add code to override, overload, or extend members
    ' inherited from the base class.
    ' Add new variable, property, procedure, and event declarations.
End Class
```

Example

The following example shows inheritance of multiple interfaces.

```
Public Interface thisInterface
    Inherits IComparable, IDisposable, IFormattable
    ' Add new property, procedure, and event definitions.
End Interface
```

The interface named `thisInterface` now includes all the definitions in the IComparable, IDisposable, and IFormattable interfaces The inherited members provide respectively for type-specific comparison of two objects, releasing allocated resources, and expressing the value of an object as a `String`. A class that implements `thisInterface` must implement every member of every base interface.

See also

- MustInherit
- NotInheritable
- Objects and Classes
- Inheritance Basics
- Interfaces

Interface Statement (Visual Basic)

4/28/2019 • 5 minutes to read • Edit Online

Declares the name of an interface and introduces the definitions of the members that the interface comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] _
Interface name [ ( Of typelist ) ]
    [ Inherits interfacenames ]
    [ [ modifiers ] Property membername ]
    [ [ modifiers ] Function membername ]
    [ [ modifiers ] Sub membername ]
    [ [ modifiers ] Event membername ]
    [ [ modifiers ] Interface membername ]
    [ [ modifiers ] Class membername ]
    [ [ modifiers ] Structure membername ]
End Interface
```

Parts

TERM	DEFINITION
`attributelist`	Optional. See Attribute List.
`accessmodifier`	Optional. Can be one of the following: - Public - Protected - Friend - Private - Protected Friend - Private Protected See Access levels in Visual Basic.
`Shadows`	Optional. See Shadows.
`name`	Required. Name of this interface. See Declared Element Names.
`Of`	Optional. Specifies that this is a generic interface.
`typelist`	Required if you use the Of keyword. List of type parameters for this interface. Optionally, each type parameter can be declared variant by using `In` and `Out` generic modifiers. See Type List.
`Inherits`	Optional. Indicates that this interface inherits the attributes and members of another interface or interfaces. See Inherits Statement.

TERM	DEFINITION
`interfacenames`	Required if you use the `Inherits` statement. The names of the interfaces from which this interface derives.
`modifiers`	Optional. Appropriate modifiers for the interface member being defined.
`Property`	Optional. Defines a property that is a member of the interface.
`Function`	Optional. Defines a `Function` procedure that is a member of the interface.
`Sub`	Optional. Defines a `Sub` procedure that is a member of the interface.
`Event`	Optional. Defines an event that is a member of the interface.
`Interface`	Optional. Defines an interface that is a nested within this interface. The nested interface definition must terminate with an `End Interface` statement.
`Class`	Optional. Defines a class that is a member of the interface. The member class definition must terminate with an `End Class` statement.
`Structure`	Optional. Defines a structure that is a member of the interface. The member structure definition must terminate with an `End Structure` statement.
`membername`	Required for each property, procedure, event, interface, class, or structure defined as a member of the interface. The name of the member.
`End Interface`	Terminates the `Interface` definition.

Remarks

An *interface* defines a set of members, such as properties and procedures, that classes and structures can implement. The interface defines only the signatures of the members and not their internal workings.

A class or structure implements the interface by supplying code for every member defined by the interface. Finally, when the application creates an instance from that class or structure, an object exists and runs in memory. For more information, see Objects and Classes and Interfaces.

You can use `Interface` only at namespace or module level. This means the *declaration context* for an interface must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see Declaration Contexts and Default Access Levels.

Interfaces default to Friend access. You can adjust their access levels with the access modifiers. For more information, see Access levels in Visual Basic.

Rules

- **Nesting Interfaces.** You can define one interface within another. The outer interface is called the *containing interface*, and the inner interface is called a *nested interface*.

- **Member Declaration.** When you declare a property or procedure as a member of an interface, you are defining only the *signature* of that property or procedure. This includes the element type (property or procedure), its parameters and parameter types, and its return type. Because of this, the member definition uses only one line of code, and terminating statements such as `End Function` or `End Property` are not valid in an interface.

  In contrast, when you define an enumeration or structure, or a nested class or interface, it is necessary to include their data members.

- **Member Modifiers.** You cannot use any access modifiers when defining module members, nor can you specify Shared or any procedure modifier except Overloads. You can declare any member with Shadows, and you can use Default when defining a property, as well as ReadOnly or WriteOnly.

- **Inheritance.** If the interface uses the Inherits Statement, you can specify one or more base interfaces. You can inherit from two interfaces even if they each define a member with the same name. If you do so, the implementing code must use name qualification to specify which member it is implementing.

  An interface cannot inherit from another interface with a more restrictive access level. For example, a `Public` interface cannot inherit from a `Friend` interface.

  An interface cannot inherit from an interface nested within it.

- **Implementation.** When a class uses the Implements statement to implement this interface, it must implement every member defined within the interface. Furthermore, each signature in the implementing code must exactly match the corresponding signature defined in this interface. However, the name of the member in the implementing code does not have to match the member name as defined in the interface.

  When a class is implementing a procedure, it cannot designate the procedure as `Shared`.

- **Default Property.** An interface can specify at most one property as its *default property*, which can be referenced without using the property name. You specify such a property by declaring it with the Default modifier.

  Notice that this means that an interface can define a default property only if it inherits none.

Behavior

- **Access Level.** All interface members implicitly have Public access. You cannot use any access modifier when defining a member. However, a class implementing the interface can declare an access level for each implemented member.

  If you assign a class instance to a variable, the access level of its members can depend on whether the data type of the variable is the underlying interface or the implementing class. The following example illustrates this.

```
Public Interface IDemo
    Sub DoSomething()
End Interface
Public Class implementIDemo
    Implements IDemo
    Private Sub DoSomething() Implements IDemo.DoSomething
    End Sub
End Class
Dim varAsInterface As IDemo = New implementIDemo()
Dim varAsClass As implementIDemo = New implementIDemo()
```

If you access class members through `varAsInterface`, they all have public access. However, if you access members through `varAsClass`, the `Sub` procedure `doSomething` has private access.

- **Scope.** An interface is in scope throughout its namespace, class, structure, or module.

  The scope of every interface member is the entire interface.

- **Lifetime.** An interface does not itself have a lifetime, nor do its members. When a class implements an interface and an object is created as an instance of that class, the object has a lifetime within the application in which it is running. For more information, see "Lifetime" in Class Statement.

Example

The following example uses the `Interface` statement to define an interface named `thisInterface`, which must be implemented with a `Property` statement and a `Function` statement.

```
Public Interface thisInterface
    Property ThisProp(ByVal thisStr As String) As Char
    Function ThisFunc(ByVal thisInt As Integer) As Integer
End Interface
```

Note that the `Property` and `Function` statements do not introduce blocks ending with `End Property` and `End Function` within the interface. The interface defines only the signatures of its members. The full `Property` and `Function` blocks appear in a class that implements `thisInterface`.

See also

- Interfaces
- Class Statement
- Module Statement
- Structure Statement
- Property Statement
- Function Statement
- Sub Statement
- Generic Types in Visual Basic
- Variance in Generic Interfaces
- In
- Out

Mid Statement

8/22/2019 • 2 minutes to read • Edit Online

Replaces a specified number of characters in a `String` variable with characters from another string.

Syntax

```
Mid( _
   ByRef Target As String, _
   ByVal Start As Integer, _
   Optional ByVal Length As Integer _
) = StringExpression
```

Parts

`Target`
Required. Name of the `String` variable to modify.

`Start`
Required. `Integer` expression. Character position in `Target` where the replacement of text begins. `Start` uses a one-based index.

`Length`
Optional. `Integer` expression. Number of characters to replace. If omitted, all of `String` is used.

`StringExpression`
Required. `String` expression that replaces part of `Target`.

Exceptions

EXCEPTION TYPE	CONDITION
ArgumentException	`Start` <= 0 or `Length` < 0.

Remarks

The number of characters replaced is always less than or equal to the number of characters in `Target`.

Visual Basic has a Mid function and a `Mid` statement. These elements both operate on a specified number of characters in a string, but the `Mid` function returns the characters while the `Mid` statement replaces the characters. For more information, see Mid.

> **NOTE**
> The `MidB` statement of earlier versions of Visual Basic replaces a substring in bytes, rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic strings are in Unicode, and `MidB` is no longer supported.

Example

This example uses the `Mid` statement to replace a specified number of characters in a string variable with characters from another string.

```
Dim testString As String
' Initializes string.
testString = "The dog jumps"
' Returns "The fox jumps".
Mid(testString, 5, 3) = "fox"
' Returns "The cow jumps".
Mid(testString, 5) = "cow"
' Returns "The cow jumpe".
Mid(testString, 5) = "cow jumped over"
' Returns "The duc jumpe".
Mid(testString, 5, 3) = "duck"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: `Strings`

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- Mid
- Strings
- Introduction to Strings in Visual Basic

Module Statement

7/26/2019 • 3 minutes to read • Edit Online

Declares the name of a module and introduces the definition of the variables, properties, events, and procedures that the module comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ]  Module name
    [ statements ]
End Module
```

Parts

`attributelist`
Optional. See Attribute List.

`accessmodifier`
Optional. Can be one of the following:

- Public
- Friend

See Access levels in Visual Basic.

`name`
Required. Name of this module. See Declared Element Names.

`statements`
Optional. Statements which define the variables, properties, events, procedures, and nested types of this module.

`End Module`
Terminates the `Module` definition.

Remarks

A `Module` statement defines a reference type available throughout its namespace. A *module* (sometimes called a *standard module*) is similar to a class but with some important distinctions. Every module has exactly one instance and does not need to be created or assigned to a variable. Modules do not support inheritance or implement interfaces. Notice that a module is not a *type* in the sense that a class or structure is — you cannot declare a programming element to have the data type of a module.

You can use `Module` only at namespace level. This means the *declaration context* for a module must be a source file or namespace, and cannot be a class, structure, module, interface, procedure, or block. You cannot nest a module within another module, or within any type. For more information, see Declaration Contexts and Default Access Levels.

A module has the same lifetime as your program. Because its members are all `Shared`, they also have lifetimes equal to that of the program.

Modules default to Friend access. You can adjust their access levels with the access modifiers. For more information, see Access levels in Visual Basic.

All members of a module are implicitly `Shared`.

Classes and Modules

These elements have many similarities, but there are some important differences as well.

- **Terminology.** Previous versions of Visual Basic recognize two types of modules: *class modules* (.cls files) and *standard modules* (.bas files). The current version calls these *classes* and *modules*, respectively.
- **Shared Members.** You can control whether a member of a class is a shared or instance member.
- **Object Orientation.** Classes are object-oriented, but modules are not. So only classes can be instantiated as objects. For more information, see Objects and Classes.

Rules

- **Modifiers.** All module members are implicitly Shared. You cannot use the `Shared` keyword when declaring a member, and you cannot alter the shared status of any member.
- **Inheritance.** A module cannot inherit from any type other than Object, from which all modules inherit. In particular, one module cannot inherit from another.

  You cannot use the Inherits Statement in a module definition, even to specify Object.
- **Default Property.** You cannot define any default properties in a module. For more information, see Default.

Behavior

- **Access Level.** Within a module, you can declare each member with its own access level. Module members default to Public access, except variables and constants, which default to Private access. When a module has more restricted access than one of its members, the specified module access level takes precedence.
- **Scope.** A module is in scope throughout its namespace.

  The scope of every module member is the entire module. Notice that all members undergo *type promotion*, which causes their scope to be promoted to the namespace containing the module. For more information, see Type Promotion.
- **Qualification.** You can have multiple modules in a project, and you can declare members with the same name in two or more modules. However, you must qualify any reference to such a member with the appropriate module name if the reference is from outside that module. For more information, see References to Declared Elements.

Example

```
Public Module thisModule
    Sub Main()
        Dim userName As String = InputBox("What is your name?")
        MsgBox("User name is " & userName)
    End Sub
    ' Insert variable, property, procedure, and event declarations.
End Module
```

See also

- Class Statement
- Namespace Statement
- Structure Statement
- Interface Statement
- Property Statement
- Type Promotion

Namespace Statement

4/2/2019 • 4 minutes to read • Edit Online

Declares the name of a namespace and causes the source code that follows the declaration to be compiled within that namespace.

Syntax

```
Namespace [Global.] { name | name.name }
    [ componenttypes ]
End Namespace
```

Parts

Global
Optional. Allows you to define a namespace out of the root namespace of your project. See Namespaces in Visual Basic.

`name`
Required. A unique name that identifies the namespace. Must be a valid Visual Basic identifier. For more information, see Declared Element Names.

`componenttypes`
Optional. Elements that make up the namespace. These include, but are not limited to, enumerations, structures, interfaces, classes, modules, delegates, and other namespaces.

`End Namespace`
Terminates a `Namespace` block.

Remarks

Namespaces are used as an organizational system. They provide a way to classify and present programming elements that are exposed to other programs and applications. Note that a namespace is not a *type* in the sense that a class or structure is—you cannot declare a programming element to have the data type of a namespace.

All programming elements declared after a `Namespace` statement belong to that namespace. Visual Basic continues to compile elements into the last declared namespace until it encounters either an `End Namespace` statement or another `Namespace` statement.

If a namespace is already defined, even outside your project, you can add programming elements to it. To do this, you use a `Namespace` statement to direct Visual Basic to compile elements into that namespace.

You can use a `Namespace` statement only at the file or namespace level. This means the *declaration context* for a namespace must be a source file or another namespace, and cannot be a class, structure, module, interface, or procedure. For more information, see Declaration Contexts and Default Access Levels.

You can declare one namespace within another. There is no strict limit to the levels of nesting you can declare, but remember that when other code accesses the elements declared in the innermost namespace, it must use a qualification string that contains all the namespace names in the nesting hierarchy.

Access Level

Namespaces are treated as if they have a `Public` access level. A namespace can be accessed from code anywhere in the same project, from other projects that reference the project, and from any assembly built from the project.

Programming elements declared at namespace level, meaning in a namespace but not inside any other element, can have `Public` or `Friend` access. If unspecified, the access level of such an element uses `Friend` by default. Elements you can declare at namespace level include classes, structures, modules, interfaces, enumerations, and delegates. For more information, see Declaration Contexts and Default Access Levels.

Root Namespace

All namespace names in your project are based on a *root namespace*. Visual Studio assigns your project name as the default root namespace for all code in your project. For example, if your project is named `Payroll`, its programming elements belong to namespace `Payroll`. If you declare `Namespace funding`, the full name of that namespace is `Payroll.funding`.

If you want to specify an existing namespace in a `Namespace` statement, such as in the generic list class example, you can set your root namespace to a null value. To do this, click **Project Properties** from the **Project** menu and then clear the **Root namespace** entry so that the box is empty. If you did not do this in the generic list class example, the Visual Basic compiler would take `System.Collections.Generic` as a new namespace within project `Payroll`, with the full name of `Payroll.System.Collections.Generic`.

Alternatively, you can use the `Global` keyword to refer to elements of namespaces defined outside your project. Doing so lets you retain your project name as the root namespace. This reduces the chance of unintentionally merging your programming elements together with those of existing namespaces. For more information, see the "Global Keyword in Fully Qualified Names" section in Namespaces in Visual Basic.

The `Global` keyword can also be used in a Namespace statement. This lets you define a namespace out of the root namespace of your project. For more information, see the "Global Keyword in Namespace Statements" section in Namespaces in Visual Basic.

Troubleshooting. The root namespace can lead to unexpected concatenations of namespace names. If you make reference to namespaces defined outside your project, the Visual Basic compiler can construe them as nested namespaces in the root namespace. In such a case, the compiler does not recognize any types that have been already defined in the external namespaces. To avoid this, either set your root namespace to a null value as described in "Root Namespace," or use the `Global` keyword to access elements of external namespaces.

Attributes and Modifiers

You cannot apply attributes to a namespace. An attribute contributes information to the assembly's metadata, which is not meaningful for source classifiers such as namespaces.

You cannot apply any access or procedure modifiers, or any other modifiers, to a namespace. Because it is not a type, these modifiers are not meaningful.

Example

The following example declares two namespaces, one nested in the other.

```
Namespace n1
    Namespace n2
        Class a
            ' Insert class definition.
        End Class
    End Namespace
End Namespace
```

Example

The following example declares multiple nested namespaces on a single line, and it is equivalent to the previous example.

```
Namespace n1.n2
    Class a
        ' Insert class definition.
    End Class
End Namespace
```

Example

The following example accesses the class defined in the previous examples.

```
Dim instance As New n1.n2.a
```

Example

The following example defines the skeleton of a new generic list class and adds it to the System.Collections.Generic namespace.

```
Namespace System.Collections.Generic
    Class specialSortedList(Of T)
        Inherits List(Of T)
        ' Insert code to define the special generic list class.
    End Class
End Namespace
```

See also

- Imports Statement (.NET Namespace and Type)
- Declared Element Names
- Namespaces in Visual Basic

On Error Statement (Visual Basic)

8/27/2019 • 7 minutes to read • Edit Online

Enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine. The `On Error` statement is used in unstructured error handling and can be used instead of structured exception handling. Structured exception handling is built into .NET, is generally more efficient, and so is recommended when handling runtime errors in your application.

Without error handling or exception handling, any run-time error that occurs is fatal: an error message is displayed, and execution stops.

> **NOTE**
> The `Error` keyword is also used in the Error Statement, which is supported for backward compatibility.

Syntax

```
On Error { GoTo [ line | 0 | -1 ] | Resume Next }
```

Parts

TERM	DEFINITION
`GoTo` *line*	Enables the error-handling routine that starts at the line specified in the required *line* argument. The *line* argument is any line label or line number. If a run-time error occurs, control branches to the specified line, making the error handler active. The specified line must be in the same procedure as the `On Error` statement or a compile-time error will occur.
`GoTo 0`	Disables enabled error handler in the current procedure and resets it to `Nothing`.
`GoTo -1`	Disables enabled exception in the current procedure and resets it to `Nothing`.
`Resume Next`	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred, and execution continues from that point. Use this form rather than `On Error GoTo` when accessing objects.

Remarks

> **NOTE**
> We recommend that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` statement. For more information, see Try...Catch...Finally Statement.

An "enabled" error handler is one that is turned on by an `On Error` statement. An "active" error handler is an enabled handler that is in the process of handling an error.

If an error occurs while an error handler is active (between the occurrence of the error and a `Resume`, `Exit Sub`, `Exit Function`, or `Exit Property` statement), the current procedure's error handler cannot handle the error. Control returns to the calling procedure.

If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no such error handler is found, the error is fatal at the point at which it actually occurred.

Each time the error handler passes control back to a calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the `Resume` statement.

> **NOTE**
> An error-handling routine is not a `Sub` procedure or a `Function` procedure. It is a section of code marked by a line label or a line number.

Number Property

Error-handling routines rely on the value in the `Number` property of the `Err` object to determine the cause of the error. The routine should test or save relevant property values in the `Err` object before any other error can occur or before a procedure that might cause an error is called. The property values in the `Err` object reflect only the most recent error. The error message associated with `Err.Number` is contained in `Err.Description`.

Throw Statement

An error that is raised with the `Err.Raise` method sets the `Exception` property to a newly created instance of the Exception class. In order to support the raising of exceptions of derived exception types, a `Throw` statement is supported in the language. This takes a single parameter that is the exception instance to be thrown. The following example shows how these features can be used with the existing exception handling support:

```
    On Error GoTo Handler
    Throw New DivideByZeroException()
Handler:
    If (TypeOf Err.GetException() Is DivideByZeroException) Then
    ' Code for handling the error is entered here.
    End If
```

Notice that the `On Error GoTo` statement traps all errors, regardless of the exception class.

On Error Resume Next

`On Error Resume Next` causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure

containing the `On Error Resume Next` statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur rather than transferring control to another location within the procedure. An `On Error Resume Next` statement becomes inactive when another procedure is called, so you should execute an `On Error Resume Next` statement in each called routine if you want inline error handling within that routine.

> **NOTE**
> The `On Error Resume Next` construct may be preferable to `On Error GoTo` when handling errors generated during access to other objects. Checking `Err` after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in `Err.Number`, as well as which object originally generated the error (the object specified in `Err.Source`).

On Error GoTo 0

`On Error GoTo 0` disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an `On Error GoTo 0` statement, an error handler is automatically disabled when a procedure is exited.

On Error GoTo -1

`On Error GoTo -1` disables the exception in the current procedure. It does not specify line -1 as the start of the error-handling code, even if the procedure contains a line numbered -1. Without an `On Error GoTo -1` statement, an exception is automatically disabled when a procedure is exited.

To prevent error-handling code from running when no error has occurred, place an `Exit Sub`, `Exit Function`, or `Exit Property` statement immediately before the error-handling routine, as in the following fragment:

```
Public Sub InitializeMatrix(ByVal Var1 As Object, ByVal Var2 As Object)
   On Error GoTo ErrorHandler
   ' Insert code that might generate an error here
   Exit Sub
ErrorHandler:
   ' Insert code to handle the error here
   Resume Next
End Sub
```

Here, the error-handling code follows the `Exit Sub` statement and precedes the `End Sub` statement to separate it from the procedure flow. You can place error-handling code anywhere in a procedure.

Untrapped Errors

Untrapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, untrapped errors are returned to the controlling application only if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle any unhandled errors they pass back. If you cannot, map the error codes in `Err.Number` to one of your own errors and then pass them back to the caller of your object. You should specify your error by adding your error code to the `VbObjectError` constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = vbObjectError + 1052
```

Caution

System errors during calls to Windows dynamic-link libraries (DLLs) do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the `Err` object's `LastDLLError` property.

Example

This example first uses the `On Error GoTo` statement to specify the location of an error-handling routine within a procedure. In the example, an attempt to divide by zero generates error number 6. The error is handled in the error-handling routine, and control is then returned to the statement that caused the error. The `On Error GoTo 0` statement turns off error trapping. Then the `On Error Resume Next` statement is used to defer error trapping so that the context for the error generated by the next statement can be known for certain. Note that `Err.Clear` is used to clear the `Err` object's properties after the error is handled.

```
Public Sub OnErrorDemo()
   On Error GoTo ErrorHandler   ' Enable error-handling routine.
   Dim x As Integer = 32
   Dim y As Integer = 0
   Dim z As Integer
   z = x / y   ' Creates a divide by zero error
   On Error GoTo 0   ' Turn off error trapping.
   On Error Resume Next   ' Defer error trapping.
   z = x / y   ' Creates a divide by zero error again
   If Err.Number = 6 Then
      ' Tell user what happened. Then clear the Err object.
      Dim Msg As String
      Msg = "There was an error attempting to divide by zero!"
      MsgBox(Msg, , "Divide by zero error")
      Err.Clear() ' Clear Err object fields.
   End If
Exit Sub      ' Exit to avoid handler.
ErrorHandler:  ' Error-handling routine.
   Select Case Err.Number   ' Evaluate error number.
      Case 6   ' Divide by zero error
         MsgBox("You attempted to divide by zero!")
         ' Insert code to handle this error
      Case Else
         ' Insert code to handle other situations here...
   End Select
   Resume Next  ' Resume execution at the statement immediately
                ' following the statement where the error occurred.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- Err
- Number
- Description
- LastDllError
- End Statement
- Exit Statement

- Resume Statement
- Error Messages
- Try...Catch...Finally Statement

Operator Statement

4/28/2019 • 5 minutes to read • Edit Online

Declares the operator symbol, operands, and code that define an operator procedure on a class or structure.

Syntax

```
[ <attrlist> ] Public [ Overloads ] Shared [ Shadows ] [ Widening | Narrowing ]
Operator operatorsymbol ( operand1 [, operand2 ]) [ As [ <attrlist> ] type ]
    [ statements ]
    [ statements ]
    Return returnvalue
    [ statements ]
End Operator
```

Parts

`attrlist`
Optional. See Attribute List.

`Public`
Required. Indicates that this operator procedure has Public access.

`Overloads`
Optional. See Overloads.

`Shared`
Required. Indicates that this operator procedure is a Shared procedure.

`Shadows`
Optional. See Shadows.

`Widening`
Required for a conversion operator unless you specify `Narrowing`. Indicates that this operator procedure defines a Widening conversion. See "Widening and Narrowing Conversions" on this Help page.

`Narrowing`
Required for a conversion operator unless you specify `Widening`. Indicates that this operator procedure defines a Narrowing conversion. See "Widening and Narrowing Conversions" on this Help page.

`operatorsymbol`
Required. The symbol or identifier of the operator that this operator procedure defines.

`operand1`
Required. The name and type of the single operand of a unary operator (including a conversion operator) or the left operand of a binary operator.

`operand2`
Required for binary operators. The name and type of the right operand of a binary operator.

`operand1` and `operand2` have the following syntax and parts:

`[ ByVal ] operandname [ As operandtype ]`

PART	DESCRIPTION
`ByVal`	Optional, but the passing mechanism must be ByVal.
`operandname`	Required. Name of the variable representing this operand. See Declared Element Names.
`operandtype`	Optional unless `Option Strict` is `On`. Data type of this operand.

`type`
Optional unless `Option Strict` is `On`. Data type of the value the operator procedure returns.

`statements`
Optional. Block of statements that the operator procedure runs.

`returnvalue`
Required. The value that the operator procedure returns to the calling code.

`End` `Operator`
Required. Terminates the definition of this operator procedure.

Remarks

You can use `Operator` only in a class or structure. This means the *declaration context* for an operator cannot be a source file, namespace, module, interface, procedure, or block. For more information, see Declaration Contexts and Default Access Levels.

All operators must be `Public Shared`. You cannot specify `ByRef`, `Optional`, or `ParamArray` for either operand.

You cannot use the operator symbol or identifier to hold a return value. You must use the `Return` statement, and it must specify a value. Any number of `Return` statements can appear anywhere in the procedure.

Defining an operator in this way is called *operator overloading*, whether or not you use the `Overloads` keyword. The following table lists the operators you can define.

TYPE	OPERATORS
Unary	`+`, `-`, `IsFalse`, `IsTrue`, `Not`
Binary	`+`, `-`, `*`, `/`, `\`, `&`, `^`, `>>`, `<<`, `=`, `<>`, `>`, `>=`, `<`, `<=`, `And`, `Like`, `Mod`, `Or`, `Xor`
Conversion (unary)	`CType`

Note that the `=` operator in the binary list is the comparison operator, not the assignment operator.

When you define `CType`, you must specify either `Widening` or `Narrowing`.

Matched Pairs

You must define certain operators as matched pairs. If you define either operator of such a pair, you must define the other as well. The matched pairs are the following:

- `=` and `<>`

- `>` and `<`
- `>=` and `<=`
- `IsTrue` and `IsFalse`

Data Type Restrictions

Every operator you define must involve the class or structure on which you define it. This means that the class or structure must appear as the data type of the following:

- The operand of a unary operator.
- At least one of the operands of a binary operator.
- Either the operand or the return type of a conversion operator.

Certain operators have additional data type restrictions, as follows:

- If you define the `IsTrue` and `IsFalse` operators, they must both return the `Boolean` type.
- If you define the `<<` and `>>` operators, they must both specify the `Integer` type for the `operandtype` of `operand2`.

The return type does not have to correspond to the type of either operand. For example, a comparison operator such as `=` or `<>` can return `Boolean` even if neither operand is `Boolean`.

Logical and Bitwise Operators

The `And`, `Or`, `Not`, and `Xor` operators can perform either logical or bitwise operations in Visual Basic. However, if you define one of these operators on a class or structure, you can define only its bitwise operation.

You cannot define the `AndAlso` operator directly with an `Operator` statement. However, you can use `AndAlso` if you have fulfilled the following conditions:

- You have defined `And` on the same operand types you want to use for `AndAlso`.
- Your definition of `And` returns the same type as the class or structure on which you have defined it.
- You have defined the `IsFalse` operator on the class or structure on which you have defined `And`.

Similarly, you can use `OrElse` if you have defined `Or` on the same operands, with the return type of the class or structure, and you have defined `IsTrue` on the class or structure.

Widening and Narrowing Conversions

A *widening conversion* always succeeds at run time, while a *narrowing conversion* can fail at run time. For more information, see Widening and Narrowing Conversions.

If you declare a conversion procedure to be `Widening`, your procedure code must not generate any failures. This means the following:

- It must always return a valid value of type `type`.
- It must handle all possible exceptions and other error conditions.
- It must handle any error returns from any procedures it calls.

If there is any possibility that a conversion procedure might not succeed, or that it might cause an unhandled

exception, you must declare it to be `Narrowing`.

Example

The following code example uses the `Operator` statement to define the outline of a structure that includes operator procedures for the `And`, `Or`, `IsFalse`, and `IsTrue` operators. `And` and `Or` each take two operands of type `abc` and return type `abc`. `IsFalse` and `IsTrue` each take a single operand of type `abc` and return `Boolean`. These definitions allow the calling code to use `And`, `AndAlso`, `Or`, and `OrElse` with operands of type `abc`.

```
Public Structure abc
    Dim d As Date
    Public Shared Operator And(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate And of x and y.
        Return r
    End Operator
    Public Shared Operator Or(ByVal x As abc, ByVal y As abc) As abc
        Dim r As New abc
        ' Insert code to calculate Or of x and y.
        Return r
    End Operator
    Public Shared Operator IsFalse(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsFalse of z.
        Return b
    End Operator
    Public Shared Operator IsTrue(ByVal z As abc) As Boolean
        Dim b As Boolean
        ' Insert code to calculate IsTrue of z.
        Return b
    End Operator
End Structure
```

See also

- IsFalse Operator
- IsTrue Operator
- Widening
- Narrowing
- Widening and Narrowing Conversions
- Operator Procedures
- How to: Define an Operator
- How to: Define a Conversion Operator
- How to: Call an Operator Procedure
- How to: Use a Class that Defines Operators

Option <keyword> Statement

4/2/2019 • 2 minutes to read • Edit Online

Introduces a statement that specifies a compiler option that applies to the entire source file.

Remarks

The compiler options can control whether all variables must be explicitly declared, whether narrowing type conversions must be explicit, or whether strings should be compared as text or as binary quantities.

The `Option` keyword can be used in these contexts:

Option Compare Statement

Option Explicit Statement

Option Infer Statement

Option Strict Statement

See also

- Keywords

Option Compare Statement

4/28/2019 • 3 minutes to read • Edit Online

Declares the default comparison method to use when comparing string data.

Syntax

```
Option Compare { Binary | Text }
```

Parts

TERM	DEFINITION
`Binary`	Optional. Results in string comparisons based on a sort order derived from the internal binary representations of the characters. This type of comparison is useful especially if the strings can contain characters that are not to be interpreted as text. In this case, you do not want to bias comparisons with alphabetical equivalences, such as case insensitivity.
`Text`	Optional. Results in string comparisons based on a case-insensitive text sort order determined by your system's locale. This type of comparison is useful if your strings contain all text characters, and you want to compare them taking into account alphabetic equivalences such as case insensitivity and closely related letters. For example, you might want to consider `A` and `a` to be equal, and `Ä` and `ä` to come before `B` and `b`.

Remarks

If used, the `Option Compare` statement must appear in a file before any other source code statements.

The `Option Compare` statement specifies the string comparison method (`Binary` or `Text`). The default text comparison method is `Binary` .

A `Binary` comparison compares the numeric Unicode value of each character in each string. A `Text` comparison compares each Unicode character based on its lexical meaning in the current culture.

In Microsoft Windows, sort order is determined by the code page. For more information, see Code Pages.

In the following example, characters in the English/European code page (ANSI 1252) are sorted by using `Option Compare Binary` , which produces a typical binary sort order.

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

When the same characters in the same code page are sorted by using `Option Compare Text` , the following text sort order is produced.

```
(A=a) < (À = à) < (B=b) < (E=e) < (Ê = ê) < (Z=z) < (Ø = ø)
```

When an Option Compare Statement Is Not Present

If the source code does not contain an `Option Compare` statement, the **Option Compare** setting on the Compile Page, Project Designer (Visual Basic) is used. If you use the command-line compiler, the setting specified by the /optioncompare compiler option is used.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To set Option Compare in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option Compare** box.

When you create a project, the **Option Compare** setting on the **Compile** tab is set to the **Option Compare** setting in the **Options** dialog box. To change this setting, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is **Binary**.

To set Option Compare on the command line

- Include the /optioncompare compiler option in the **vbc** command.

Example

The following example uses the `Option Compare` statement to set the binary comparison as the default string comparison method. To use this code, uncomment the `Option Compare Binary` statement, and put it at the top of the source file.

```
' Option Compare Binary

Console.WriteLine("A" < "a")
' Output: True
```

Example

The following example uses the `Option Compare` statement to set the case-insensitive text sort order as the default string comparison method. To use this code, uncomment the `Option Compare Text` statement, and put it at the top of the source file.

```
' Option Compare Text

Console.WriteLine("A" = "a")
' Output: True
```

See also

- InStr

- InStrRev
- Replace
- Split
- StrComp
- /optioncompare
- Comparison Operators
- Comparison Operators in Visual Basic
- Like Operator
- String Functions
- Option Explicit Statement
- Option Strict Statement

Option Explicit Statement (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Forces explicit declaration of all variables in a file, or allows implicit declarations of variables.

Syntax

```
Option Explicit { On | Off }
```

Parts

`On`
Optional. Enables `Option Explicit` checking. If `On` or `Off` is not specified, the default is `On`.

`Off`
Optional. Disables `Option Explicit` checking.

Remarks

When `Option Explicit On` or `Option Explicit` appears in a file, you must explicitly declare all variables by using the `Dim` or `ReDim` statements. If you try to use an undeclared variable name, an error occurs at compile time. The `Option Explicit Off` statement allows implicit declaration of variables.

If used, the `Option Explicit` statement must appear in a file before any other source code statements.

> **NOTE**
> Setting `Option Explicit` to `Off` is generally not a good practice. You could misspell a variable name in one or more locations, which would cause unexpected results when the program is run.

When an Option Explicit Statement Is Not Present

If the source code does not contain an `Option Explicit` statement, the **Option Explicit** setting on the Compile Page, Project Designer (Visual Basic) is used. If the command-line compiler is used, the /optionexplicit compiler option is used.

To set Option Explicit in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option Explicit** box.

When you create a new project, the **Option Explicit** setting on the **Compile** tab is set to the **Option Explicit** setting in the **VB Defaults** dialog box. To access the **VB Defaults** dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `On`.

To set Option Explicit on the command line

- Include the /optionexplicit compiler option in the **vbc** command.

Example

The following example uses the `Option Explicit` statement to force explicit declaration of all variables. Attempting to use an undeclared variable causes an error at compile time.

```
' Force explicit variable declaration.
Option Explicit On
```

```
Dim thisVar As Integer
thisVar = 10
' The following assignment produces a COMPILER ERROR because
' the variable is not declared and Option Explicit is On.
thisInt = 10 ' causes ERROR
```

See also

- Dim Statement
- ReDim Statement
- Option Compare Statement
- Option Strict Statement
- /optioncompare
- /optionexplicit
- /optionstrict
- Visual Basic Defaults, Projects, Options Dialog Box

Option Infer Statement

8/22/2019 • 4 minutes to read • Edit Online

Enables the use of local type inference in declaring variables.

Syntax

```
Option Infer { On | Off }
```

Parts

TERM	DEFINITION
`On`	Optional. Enables local type inference.
`Off`	Optional. Disables local type inference.

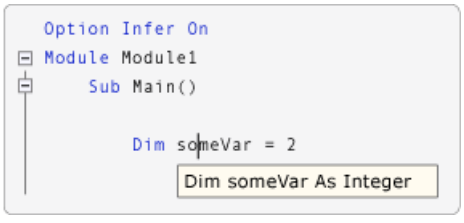
Remarks

To set `Option Infer` in a file, type `Option Infer On` or `Option Infer Off` at the top of the file, before any other source code. If the value set for `Option Infer` in a file conflicts with the value set in the IDE or on the command line, the value in the file has precedence.

When you set `Option Infer` to `On`, you can declare local variables without explicitly stating a data type. The compiler infers the data type of a variable from the type of its initialization expression.

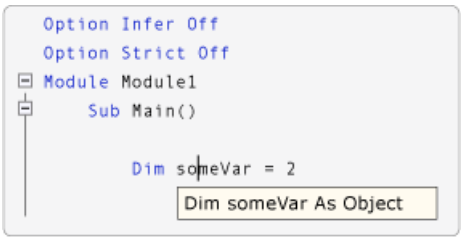
In the following illustration, `Option Infer` is turned on. The variable in the declaration `Dim someVar = 2` is declared as an integer by type inference.

The following screenshot shows IntelliSense when Option Infer is on:



In the following illustration, `Option Infer` is turned off. The variable in the declaration `Dim someVar = 2` is declared as an `Object` by type inference. In this example, the **Option Strict** setting is set to **Off** on the Compile Page, Project Designer (Visual Basic).

The following screenshot shows IntelliSense when Option Infer is off:



> **NOTE**
>
> When a variable is declared as an `Object`, the run-time type can change while the program is running. Visual Basic performs operations called *boxing* and *unboxing* to convert between an `Object` and a value type, which makes execution slower. For information about boxing and unboxing, see the Visual Basic Language Specification.

Type inference applies at the procedure level, and does not apply outside a procedure in a class, structure, module, or interface.

For additional information, see Local Type Inference.

When an Option Infer Statement Is Not Present

If the source code does not contain an `Option Infer` statement, the **Option Infer** setting on the Compile Page, Project Designer (Visual Basic) is used. If the command-line compiler is used, the /optioninfer compiler option is used.

To set Option Infer in the IDE

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Option infer** box.

When you create a new project, the **Option Infer** setting on the **Compile** tab is set to the **Option Infer** setting in the **VB Defaults** dialog box. To access the **VB Defaults** dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `On`.

To set Option Infer on the command line

- Include the /optioninfer compiler option in the **vbc** command.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a `Dim` statement.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	`Dim qty`	If `Option Strict` is off (the default), the variable is set to `Nothing`. If `Option Strict` is on, a compile-time error occurs.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	Yes	`Dim qty = 5`	If `Option Infer` is on (the default), the variable takes the data type of the initializer. See Local Type Inference. If `Option Infer` is off and `Option Strict` is off, the variable takes the data type of `Object`. If `Option Infer` is off and `Option Strict` is on, a compile-time error occurs.
Yes	No	`Dim qty As Integer`	The variable is initialized to the default value for the data type. For more information, see Dim Statement.
Yes	Yes	`Dim qty As Integer = 5`	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

Example

The following examples demonstrate how the `Option Infer` statement enables local type inference.

```
' Enable Option Infer before trying these examples.

' Variable num is an Integer.
Dim num = 5

' Variable dbl is a Double.
Dim dbl = 4.113

' Variable str is a String.
Dim str = "abc"

' Variable pList is an array of Process objects.
Dim pList = Process.GetProcesses()

' Variable i is an Integer.
For i = 1 To 10
    Console.WriteLine(i)
Next

' Variable item is a string.
Dim lst As New List(Of String) From {"abc", "def", "ghi"}

For Each item In lst
    Console.WriteLine(item)
Next

' Variable namedCust is an instance of the Customer class.
Dim namedCust = New Customer With {.Name = "Blue Yonder Airlines",
                                   .City = "Snoqualmie"}

' Variable product is an instance of an anonymous type.
Dim product = New With {Key .Name = "paperclips", .Price = 1.29}

' If customers is a collection of Customer objects in the following
' query, the inferred type of cust is Customer, and the inferred type
' of custs is IEnumerable(Of Customer).
Dim custs = From cust In customers
            Where cust.City = "Seattle"
            Select cust.Name, cust.ID
```

Example

The following example demonstrates that the run-time type can differ when a variable is identified as an `Object`.

```
' Disable Option Infer when trying this example.

Dim someVar = 5
Console.WriteLine(someVar.GetType.ToString)

' If Option Infer is instead enabled, the following
' statement causes a run-time error. This is because
' someVar was implicitly defined as an integer.
someVar = "abc"
Console.WriteLine(someVar.GetType.ToString)

' Output:
'   System.Int32
'   System.String
```

See also

- Dim Statement
- Local Type Inference
- Option Compare Statement
- Option Explicit Statement
- Option Strict Statement
- Visual Basic Defaults, Projects, Options Dialog Box
- /optioninfer
- Boxing and Unboxing

Option Strict Statement

8/22/2019 • 9 minutes to read • Edit Online

Restricts implicit data type conversions to only widening conversions, disallows late binding, and disallows implicit typing that results in an `Object` type.

Syntax

```
Option Strict { On | Off }
```

Parts

TERM	DEFINITION
`On`	Optional. Enables `Option Strict` checking.
`Off`	Optional. Disables `Option Strict` checking.

Remarks

When `Option Strict On` or `Option Strict` appears in a file, the following conditions cause a compile-time error:

- Implicit narrowing conversions
- Late binding
- Implicit typing that results in an `Object` type

> **NOTE**
> In the warning configurations that you can set on the Compile Page, Project Designer (Visual Basic), there are three settings that correspond to the three conditions that cause a compile-time error. For information about how to use these settings, see To set warning configurations in the IDE later in this topic.

The `Option Strict Off` statement turns off error and warning checking for all three conditions, even if the associated IDE settings specify to turn on these errors or warnings. The `Option Strict On` statement turns on error and warning checking for all three conditions, even if the associated IDE settings specify to turn off these errors or warnings.

If used, the `Option Strict` statement must appear before any other code statements in a file.

When you set `Option Strict` to `On`, Visual Basic checks that data types are specified for all programming elements. Data types can be specified explicitly, or specified by using local type inference. Specifying data types for all your programming elements is recommended, for the following reasons:

- It enables IntelliSense support for your variables and parameters. This enables you to see their properties and other members as you type code.
- It enables the compiler to perform type checking. Type checking helps you find statements that can

fail at run time because of type conversion errors. It also identifies calls to methods on objects that do not support those methods.

- It speeds up the execution of code. One reason for this is that if you do not specify a data type for a programming element, the Visual Basic compiler assigns it the `Object` type. Compiled code might have to convert back and forth between `Object` and other data types, which reduces performance.

Implicit Narrowing Conversion Errors

Implicit narrowing conversion errors occur when there is an implicit data type conversion that is a narrowing conversion.

Visual Basic can convert many data types to other data types. Data loss can occur when the value of one data type is converted to a data type that has less precision or a smaller capacity. A run-time error occurs if such a narrowing conversion fails. `Option Strict` ensures compile-time notification of these narrowing conversions so that you can avoid them. For more information, see Implicit and Explicit Conversions and Widening and Narrowing Conversions.

Conversions that can cause errors include implicit conversions that occur in expressions. For more information, see the following topics:

- + Operator
- += Operator
- \ Operator (Visual Basic)
- /= Operator (Visual Basic)
- Char Data Type

When you concatenate strings by using the & Operator, all conversions to the strings are considered to be widening. So these conversions do not generate an implicit narrowing conversion error, even if `Option Strict` is on.

When you call a method that has an argument that has a data type different from the corresponding parameter, a narrowing conversion causes a compile-time error if `Option Strict` is on. You can avoid the compile-time error by using a widening conversion or an explicit conversion.

Implicit narrowing conversion errors are suppressed at compile-time for conversions from the elements in a `For Each…Next` collection to the loop control variable. This occurs even if `Option Strict` is on. For more information, see the "Narrowing Conversions" section in For Each...Next Statement.

Late Binding Errors

An object is late bound when it is assigned to a property or method of a variable that is declared to be of type `Object`. For more information, see Early and Late Binding.

Implicit Object Type Errors

Implicit object type errors occur when an appropriate type cannot be inferred for a declared variable, so a type of `Object` is inferred. This primarily occurs when you use a `Dim` statement to declare a variable without using an `As` clause, and `Option Infer` is off. For more information, see Option Infer Statement and the Visual Basic Language Specification.

For method parameters, the `As` clause is optional if `Option Strict` is off. However, if any one parameter uses an `As` clause, they all must use it. If `Option Strict` is on, the `As` clause is required for every

parameter definition.

If you declare a variable without using an `As` clause and set it to `Nothing`, the variable has a type of `Object`. No compile-time error occurs in this case when `Option Strict` is on and `Option Infer` is on. An example of this is `Dim something = Nothing`.

Default Data Types and Values

The following table describes the results of various combinations of specifying the data type and initializer in a Dim Statement.

DATA TYPE SPECIFIED?	INITIALIZER SPECIFIED?	EXAMPLE	RESULT
No	No	`Dim qty`	If `Option Strict` is off (the default), the variable is set to `Nothing`. If `Option Strict` is on, a compile-time error occurs.
No	Yes	`Dim qty = 5`	If `Option Infer` is on (the default), the variable takes the data type of the initializer. See Local Type Inference. If `Option Infer` is off and `Option Strict` is off, the variable takes the data type of `Object`. If `Option Infer` is off and `Option Strict` is on, a compile-time error occurs.
Yes	No	`Dim qty As Integer`	The variable is initialized to the default value for the data type. For more information, see Dim Statement.
Yes	Yes	`Dim qty As Integer = 5`	If the data type of the initializer is not convertible to the specified data type, a compile-time error occurs.

When an Option Strict Statement Is Not Present

If the source code does not contain an `Option Strict` statement, the **Option strict** setting on the Compile Page, Project Designer (Visual Basic) is used. The **Compile Page** has settings that provide additional control over the conditions that generate an error.

If you are using the command-line compiler, you can use the /optionstrict compiler option to specify a setting for `Option Strict`.

To set Option Strict in the IDE

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

1. In **Solution Explorer**, select a project. On the **Project** menu, click **Properties**.
2. On the **Compile** tab, set the value in the **Option Strict** box.

To set warning configurations in the IDE

When you use the Compile Page, Project Designer (Visual Basic) instead of an `Option Strict` statement, you have additional control over the conditions that generate errors. The **Warning configurations** section of the **Compile Page** has settings that correspond to the three conditions that cause a compile-time error when `Option Strict` is on. Following are these settings:

- **Implicit conversion**
- **Late binding; call could fail at run time**
- **Implicit type; object assumed**

When you set **Option Strict** to **On**, all three of these warning configuration settings are set to **Error**. When you set **Option Strict** to **Off**, all three settings are set to **None**.

You can individually change each warning configuration setting to **None**, **Warning**, or **Error**. If all three warning configuration settings are set to **Error**, `On` appears in the `Option strict` box. If all three are set to **None**, `Off` appears in this box. For any other combination of these settings, **(custom)** appears.

To set the Option Strict default setting for new projects

When you create a project, the **Option Strict** setting on the **Compile** tab is set to the **Option Strict** setting in the **Options** dialog box.

To set `Option Strict` in this dialog box, on the **Tools** menu, click **Options**. In the **Options** dialog box, expand **Projects and Solutions**, and then click **VB Defaults**. The initial default setting in **VB Defaults** is `Off`.

To set Option Strict on the command line

Include the /optionstrict compiler option in the **vbc** command.

Example

The following examples demonstrate compile-time errors caused by implicit type conversions that are narrowing conversions. This category of errors corresponds to the **Implicit conversion** condition on the **Compile Page**.

```
' If Option Strict is on, this implicit narrowing
' conversion causes a compile-time error.
' The commented statements below use explicit
' conversions to avoid a compile-time error.
Dim cyclists As Long = 5
Dim bicycles As Integer = cyclists
'Dim bicycles As Integer = CType(cyclists, Integer)
'Dim bicycles As Integer = CInt(cyclists)
'Dim bicycles As Integer = Convert.ToInt32(cyclists)


' If Option Strict is on, this implicit narrowing
' conversion causes a compile-time error.
' The commented statements below use explicit
' conversions to avoid a compile-time error.
Dim charVal As Char = "a"
'Dim charVal As Char = "a"c
'Dim charVal As Char = CType("a", Char)


' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the string is implicitly converted
' to a Double, and then is added to the other number.
Dim myAge As Integer = "34" + 6


' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the floating-point number
' is implicitly converted to a Long.
Dim num = 123.45 \ 10
```

Example

The following example demonstrates a compile-time error caused by late binding. This category of errors corresponds to the **Late binding; call could fail at run time** condition on the **Compile Page**.

```
' If Option Strict is on, this late binding
' causes a compile-time error. If Option Strict
' is off, the late binding instead causes a
' run-time error.
Dim punchCard As New Object
punchCard.Column = 5
```

Example

The following examples demonstrate errors caused by variables that are declared with an implicit type of `Object`. This category of errors corresponds to the **Implicit type; object assumed** condition on the **Compile Page**.

```
' If Option Strict is on and Option Infer is off,
' this Dim statement without an As clause
' causes a compile-time error.
Dim cardReaders = 5

' If Option Strict is on, a compile-time error occurs.
' If Option Strict is off, the variable is set to Nothing.
Dim dryWall
```

```
' If Option Strict is on, this parameter without an
' As clause causes a compile-time error.
Private Sub DetectIntergalacticRange(ByVal photonAttenuation)

End Sub
```

See also

- Widening and Narrowing Conversions
- Implicit and Explicit Conversions
- Compile Page, Project Designer (Visual Basic)
- Option Explicit Statement
- Type Conversion Functions
- How to: Access Members of an Object
- Embedded Expressions in XML
- Relaxed Delegate Conversion
- Late Binding in Office Solutions
- /optionstrict
- Visual Basic Defaults, Projects, Options Dialog Box

Property Statement

4/26/2019 • 5 minutes to read • Edit Online

Declares the name of a property, and the property procedures used to store and retrieve the value of the property.

Syntax

```
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ] [ Iterator ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
    [ <attributelist> ] [ accessmodifier ] Get
        [ statements ]
    End Get
    [ <attributelist> ] [ accessmodifier ] Set ( ByVal value As returntype [, parameterlist ] )
        [ statements ]
    End Set
End Property
- or -
[ <attributelist> ] [ Default ] [ accessmodifier ]
[ propertymodifiers ] [ Shared ] [ Shadows ] [ ReadOnly | WriteOnly ]
Property name ( [ parameterlist ] ) [ As returntype ] [ Implements implementslist ]
```

Parts

- `attributelist`

  Optional. List of attributes that apply to this property or `Get` or `Set` procedure. See Attribute List.

- `Default`

  Optional. Specifies that this property is the default property for the class or structure on which it is defined. Default properties must accept parameters and can be set and retrieved without specifying the property name. If you declare the property as `Default`, you cannot use `Private` on the property or on either of its property procedures.

- `accessmodifier`

  Optional on the `Property` statement and on at most one of the `Get` and `Set` statements. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - Protected Friend
  - Private Protected

  See Access levels in Visual Basic.

- `propertymodifiers`

  Optional. Can be one of the following:

  - Overloads
  - Overrides
  - Overridable
  - NotOverridable
  - MustOverride
  - `MustOverride Overrides`
  - `NotOverridable Overrides`

- `Shared`

  Optional. See Shared.

- `Shadows`

  Optional. See Shadows.

- `ReadOnly`

  Optional. See ReadOnly.

- `WriteOnly`

  Optional. See WriteOnly.

- `Iterator`

  Optional. See Iterator.

- `name`

  Required. Name of the property. See Declared Element Names.

- `parameterlist`

  Optional. List of local variable names representing the parameters of this property, and possible additional parameters of the `Set` procedure. See Parameter List.

- `returntype`

  Required if `Option Strict` is `On`. Data type of the value returned by this property.

- `Implements`

  Optional. Indicates that this property implements one or more properties, each one defined in an interface implemented by this property's containing class or structure. See Implements Statement.

- `implementslist`

  Required if `Implements` is supplied. List of properties being implemented.

  ```
  implementedproperty [ , implementedproperty ... ]
  ```

  Each `implementedproperty` has the following syntax and parts:

`interface.definedname`

PART	DESCRIPTION
`interface`	Required. Name of an interface implemented by this property's containing class or structure.
`definedname`	Required. Name by which the property is defined in `interface`.

- `Get`

  Optional. Required if the property is marked `WriteOnly`. Starts a `Get` property procedure that is used to return the value of the property.

- `statements`

  Optional. Block of statements to run within the `Get` or `Set` procedure.

- `End Get`

  Terminates the `Get` property procedure.

- `Set`

  Optional. Required if the property is marked `ReadOnly`. Starts a `Set` property procedure that is used to store the value of the property.

- `End Set`

  Terminates the `Set` property procedure.

- `End Property`

  Terminates the definition of this property.

Remarks

The `Property` statement introduces the declaration of a property. A property can have a `Get` procedure (read only), a `Set` procedure (write only), or both (read-write). You can omit the `Get` and `Set` procedure when using an auto-implemented property. For more information, see Auto-Implemented Properties.

You can use `Property` only at class level. This means the *declaration context* for a property must be a class, structure, module, or interface, and cannot be a source file, namespace, procedure, or block. For more information, see Declaration Contexts and Default Access Levels.

By default, properties use public access. You can adjust a property's access level with an access modifier on the `Property` statement, and you can optionally adjust one of its property procedures to a more restrictive access level.

Visual Basic passes a parameter to the `Set` procedure during property assignments. If you do not supply a parameter for `Set`, the integrated development environment (IDE) uses an implicit parameter named `value`. This parameter holds the value to be assigned to the property. You typically store this value in a private local variable and return it whenever the `Get` procedure is called.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Set` procedure `Private`, but not `Public`.

  If you are defining a `ReadOnly` or `WriteOnly` property, the single property procedure (`Get` or `Set`, respectively) represents all of the property. You cannot declare a different access level for such a procedure, because that would set two access levels for the property.

- **Return Type.** The `Property` statement can declare the data type of the value it returns. You can specify any data type or the name of an enumeration, structure, class, or interface.

  If you do not specify `returntype`, the property returns `Object`.

- **Implementation.** If this property uses the `Implements` keyword, the containing class or structure must have an `Implements` statement immediately following its `Class` or `Structure` statement. The `Implements` statement must include each interface specified in `implementslist`. However, the name by which an interface defines the `Property` (in `definedname`) does not have to be the same as the name of this property (in `name`).

Behavior

- **Returning from a Property Procedure.** When the `Get` or `Set` procedure returns to the calling code, execution continues with the statement following the statement that invoked it.

  The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

- **Return Value.** To return a value from a `Get` procedure, you can either assign the value to the property name or include it in a `Return` statement. The following example assigns the return value to the property name `quoteForTheDay` and then uses the `Exit Property` statement to return.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        QuoteForTheDay = quoteValue
        Exit Property
    End Get
End Property
```

  If you use `Exit Property` without assigning a value to `name`, the `Get` procedure returns the default value for the property's data type.

  The `Return` statement at the same time assigns the `Get` procedure return value and exits the procedure. The following example shows this.

```
Private quoteValue As String = "No quote assigned yet."
```

```
ReadOnly Property QuoteForTheDay() As String
    Get
        Return quoteValue
    End Get
End Property
```

Example

The following example declares a property in a class.

```
Class Class1
    ' Define a local variable to store the property value.
    Private propertyValue As String
    ' Define the property.
    Public Property Prop1() As String
        Get
            ' The Get property procedure is called when the value
            ' of a property is retrieved.
            Return propertyValue
        End Get
        Set(ByVal value As String)
            ' The Set property procedure is called when the value
            ' of a property is modified.  The value to be assigned
            ' is passed in the argument to Set.
            propertyValue = value
        End Set
    End Property
End Class
```

See also

- Auto-Implemented Properties
- Objects and Classes
- Get Statement
- Set Statement
- Parameter List
- Default

Q-Z Statements

4/2/2019 • 2 minutes to read • Edit Online

The following table contains a listing of Visual Basic language statements.

RaiseEvent	ReDim	REM	RemoveHandler
Resume	Return	Select...Case	Set
Stop	Structure	Sub	SyncLock
Then	Throw	Try...Catch...Finally	Using
While...End While	With...End With	Yield	

See also

- A-E Statements
- F-P Statements
- Visual Basic Language Reference

RaiseEvent Statement

8/22/2019 • 3 minutes to read • Edit Online

Triggers an event declared at module level within a class, form, or document.

Syntax

```
RaiseEvent eventname[( argumentlist )]
```

Parts

`eventname`
Required. Name of the event to trigger.

`argumentlist`
Optional. Comma-delimited list of variables, arrays, or expressions. The `argumentlist` argument must be enclosed by parentheses. If there are no arguments, the parentheses must be omitted.

Remarks

The required `eventname` is the name of an event declared within the module. It follows Visual Basic variable naming conventions.

If the event has not been declared within the module in which it is raised, an error occurs. The following code fragment illustrates an event declaration and a procedure in which the event is raised.

```
' Declare an event at module level.
Event LogonCompleted(ByVal UserName As String)

Sub Logon(ByVal UserName As String)
    ' Raise the event.
    RaiseEvent LogonCompleted(UserName)
End Sub
```

You cannot use `RaiseEvent` to raise events that are not explicitly declared in the module. For example, all forms inherit a Click event from System.Windows.Forms.Form, it cannot be raised using `RaiseEvent` in a derived form. If you declare a `Click` event in the form module, it shadows the form's own Click event. You can still invoke the form's Click event by calling the OnClick method.

By default, an event defined in Visual Basic raises its event handlers in the order that the connections are established. Because events can have `ByRef` parameters, a process that connects late may receive parameters that have been changed by an earlier event handler. After the event handlers execute, control is returned to the subroutine that raised the event.

> **NOTE**
> Non-shared events should not be raised within the constructor of the class in which they are declared. Although such events do not cause run-time errors, they may fail to be caught by associated event handlers. Use the `Shared` modifier to create a shared event if you need to raise an event from a constructor.

NOTE

You can change the default behavior of events by defining a custom event. For custom events, the `RaiseEvent` statement invokes the event's `RaiseEvent` accessor. For more information on custom events, see Event Statement.

Example

The following example uses events to count down seconds from 10 to 0. The code illustrates several of the event-related methods, properties, and statements, including the `RaiseEvent` statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Button1`) and a text box (`TextBox1`). When you click the button, the first text box displays a countdown from 10 to 0 seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Application project, add a button named `Button1` and a text box named `TextBox1` to the main form, named `Form1`. Then right-click the form and click **View Code** to open the Code Editor.

Add a `WithEvents` variable to the declarations section of the `Form1` class.

```
Private WithEvents mText As TimerState
```

Example

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load`, or `Button_Click`.

```
Private Sub Form1_Load() Handles MyBase.Load
    Button1.Text = "Start"
    mText = New TimerState
End Sub
Private Sub Button1_Click() Handles Button1.Click
    mText.StartCountdown(10.0, 0.1)
End Sub

Private Sub mText_ChangeText() Handles mText.Finished
    TextBox1.Text = "Done"
End Sub

Private Sub mText_UpdateTime(ByVal Countdown As Double
  ) Handles mText.UpdateTime

    TextBox1.Text = Format(Countdown, "##0.0")
    ' Use DoEvents to allow the display to refresh.
    My.Application.DoEvents()
End Sub

Class TimerState
    Public Event UpdateTime(ByVal Countdown As Double)
    Public Event Finished()
    Public Sub StartCountdown(ByVal Duration As Double,
                              ByVal Increment As Double)
        Dim Start As Double = DateAndTime.Timer
        Dim ElapsedTime As Double = 0

        Dim SoFar As Double = 0
        Do While ElapsedTime < Duration
            If ElapsedTime > SoFar + Increment Then
                SoFar += Increment
                RaiseEvent UpdateTime(Duration - SoFar)
            End If
            ElapsedTime = DateAndTime.Timer - Start
        Loop
        RaiseEvent Finished()
    End Sub
End Class
```

Press F5 to run the preceding example, and click the button labeled **Start**. The first text box starts to count down the seconds. When the full time (10 seconds) has elapsed, the first text box displays "Done".

> **NOTE**
>
> The `My.Application.DoEvents` method does not process events in exactly the same way as the form does. To allow the form to handle the events directly, you can use multithreading. For more information, see Managed Threading.

See also

- Events
- Event Statement
- AddHandler Statement
- RemoveHandler Statement
- Handles

ReDim Statement (Visual Basic)

4/28/2019 • 4 minutes to read • Edit Online

Reallocates storage space for an array variable.

Syntax

```
ReDim [ Preserve ] name(boundlist) [ ,  name(boundlist) [, ... ] ]
```

Parts

TERM	DEFINITION
`Preserve`	Optional. Modifier used to preserve the data in the existing array when you change the size of only the last dimension.
`name`	Required. Name of the array variable. See Declared Element Names.
`boundlist`	Required. List of bounds of each dimension of the redefined array.

Remarks

You can use the `ReDim` statement to change the size of one or more dimensions of an array that has already been declared. If you have a large array and you no longer need some of its elements, `ReDim` can free up memory by reducing the array size. On the other hand, if your array needs more elements, `ReDim` can add them.

The `ReDim` statement is intended only for arrays. It's not valid on scalars (variables that contain only a single value), collections, or structures. Note that if you declare a variable to be of type `Array`, the `ReDim` statement doesn't have sufficient type information to create the new array.

You can use `ReDim` only at procedure level. Therefore, the declaration context for the variable must be a procedure; it can't be a source file, a namespace, an interface, a class, a structure, a module, or a block. For more information, see Declaration Contexts and Default Access Levels.

Rules

- **Multiple Variables.** You can resize several array variables in the same declaration statement and specify the `name` and `boundlist` parts for each variable. Multiple variables are separated by commas.
- **Array Bounds.** Each entry in `boundlist` can specify the lower and upper bounds of that dimension. The lower bound is always 0 (zero). The upper bound is the highest possible index value for that dimension, not the length of the dimension (which is the upper bound plus one). The index for each dimension can vary from 0 through its upper bound value.

  The number of dimensions in `boundlist` must match the original number of dimensions (rank) of the array.

- **Data Types.** The `ReDim` statement cannot change the data type of an array variable or its elements.
- **Initialization.** The `ReDim` statement cannot provide new initialization values for the array elements.
- **Rank.** The `ReDim` statement cannot change the rank (the number of dimensions) of the array.
- **Resizing with Preserve.** If you use `Preserve`, you can resize only the last dimension of the array. For every other dimension, you must specify the bound of the existing array.

  For example, if your array has only one dimension, you can resize that dimension and still preserve all the contents of the array, because you are changing the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension if you use `Preserve`.
- **Properties.** You can use `ReDim` on a property that holds an array of values.

Behavior

- **Array Replacement.** `ReDim` releases the existing array and creates a new array with the same rank. The new array replaces the released array in the array variable.
- **Initialization without Preserve.** If you do not specify `Preserve`, `ReDim` initializes the elements of the new array by using the default value for their data type.
- **Initialization with Preserve.** If you specify `Preserve`, Visual Basic copies the elements from the existing array to the new array.

Example

The following example increases the size of the last dimension of a dynamic array without losing any existing data in the array, and then decreases the size with partial data loss. Finally, it decreases the size back to its original value and reinitializes all the array elements.

```
Dim intArray(10, 10, 10) As Integer
ReDim Preserve intArray(10, 10, 20)
ReDim Preserve intArray(10, 10, 15)
ReDim intArray(10, 10, 10)
```

The `Dim` statement creates a new array with three dimensions. Each dimension is declared with a bound of 10, so the array index for each dimension can range from 0 through 10. In the following discussion, the three dimensions are referred to as layer, row, and column.

The first `ReDim` creates a new array which replaces the existing array in variable `intArray`. `ReDim` copies all the elements from the existing array into the new array. It also adds 10 more columns to the end of every row in every layer and initializes the elements in these new columns to 0 (the default value of `Integer`, which is the element type of the array).

The second `ReDim` creates another new array and copies all the elements that fit. However, five columns are lost from the end of every row in every layer. This is not a problem if you have finished using these columns. Reducing the size of a large array can free up memory that you no longer need.

The third `ReDim` creates another new array and removes another five columns from the end of every row in every layer. This time it does not copy any existing elements. This statement reverts the array to its original size. Because the statement doesn't include the `Preserve` modifier, it sets all array elements to their original default values.

For additional examples, see Arrays.

See also

- IndexOutOfRangeException
- Const Statement
- Dim Statement
- Erase Statement
- Nothing
- Arrays

REM Statement (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Used to include explanatory remarks in the source code of a program.

Syntax

```
REM comment
' comment
```

Parts

`comment`
Optional. The text of any comment you want to include. A space is required between the `REM` keyword and `comment`.

Remarks

You can put a `REM` statement alone on a line, or you can put it on a line following another statement. The `REM` statement must be the last statement on the line. If it follows another statement, the `REM` must be separated from that statement by a space.

You can use a single quotation mark (`'`) instead of `REM`. This is true whether your comment follows another statement on the same line or sits alone on a line.

> **NOTE**
> You cannot continue a `REM` statement by using a line-continuation sequence (`_`). Once a comment begins, the compiler does not examine the characters for special meaning. For a multiple-line comment, use another `REM` statement or a comment symbol (`'`) on each line.

Example

The following example illustrates the `REM` statement, which is used to include explanatory remarks in a program. It also shows the alternative of using the single quotation-mark character (`'`) instead of `REM`.

```
Dim demoStr1, demoStr2 As String
demoStr1 = "Hello" REM Comment after a statement using REM.
demoStr2 = "Goodbye" ' Comment after a statement using the ' character.
REM This entire line is a comment.
' This entire line is also a comment.
```

See also

- Comments in Code
- How to: Break and Combine Statements in Code

RemoveHandler Statement

8/22/2019 • 2 minutes to read • Edit Online

Removes the association between an event and an event handler.

Syntax

```
RemoveHandler event, AddressOf eventhandler
```

Parts

TERM	DEFINITION
`event`	The name of the event being handled.
`eventhandler`	The name of the procedure currently handling the event.

Remarks

The `AddHandler` and `RemoveHandler` statements allow you to start and stop event handling for a specific event at any time during program execution.

> **NOTE**
> For custom events, the `RemoveHandler` statement invokes the event's `RemoveHandler` accessor. For more information on custom events, see Event Statement.

Example

```
Sub TestEvents()
    Dim Obj As New Class1
    ' Associate an event handler with an event.
    AddHandler Obj.Ev_Event, AddressOf EventHandler
    ' Call the method to raise the event.
    Obj.CauseSomeEvent()
    ' Stop handling events.
    RemoveHandler Obj.Ev_Event, AddressOf EventHandler
    ' This event will not be handled.
    Obj.CauseSomeEvent()
End Sub

Sub EventHandler()
    ' Handle the event.
    MsgBox("EventHandler caught event.")
End Sub

Public Class Class1
    ' Declare an event.
    Public Event Ev_Event()
    Sub CauseSomeEvent()
        ' Raise an event.
        RaiseEvent Ev_Event()
    End Sub
End Class
```

See also

- AddHandler Statement
- Handles
- Event Statement
- Events

Resume Statement

8/22/2019 • 2 minutes to read • Edit Online

Resumes execution after an error-handling routine is finished.

We suggest that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` and `Resume` statements. For more information, see Try...Catch...Finally Statement.

Syntax

```
Resume [ Next | line ]
```

Parts

`Resume`
Required. If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.

`Next`
Optional. If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or `On Error Resume Next` statement).

`line`
Optional. Execution resumes at the line specified in the required `line` argument. The `line` argument is a line label or line number and must be in the same procedure as the error handler.

Remarks

> **NOTE**
> We recommend that you use structured exception handling in your code whenever possible, rather than using unstructured exception handling and the `On Error` and `Resume` statements. For more information, see Try...Catch...Finally Statement.

If you use a `Resume` statement anywhere other than in an error-handling routine, an error occurs.

The `Resume` statement cannot be used in any procedure that contains a `Try...Catch...Finally` statement.

Example

This example uses the `Resume` statement to end error handling in a procedure and then resume execution with the statement that caused the error. Error number 55 is generated to illustrate use of the `Resume` statement.

```
Sub ResumeStatementDemo()
  On Error GoTo ErrorHandler   ' Enable error-handling routine.
  Dim x As Integer = 32
  Dim y As Integer = 0
  Dim z As Integer
  z = x / y   ' Creates a divide by zero error
  Exit Sub   ' Exit Sub to avoid error handler.
ErrorHandler:     ' Error-handling routine.
  Select Case Err.Number   ' Evaluate error number.
      Case 6   ' "Divide by zero" error.
        y = 1 ' Sets the value of y to 1 and tries the calculation again.
      Case Else
        ' Handle other situations here....
  End Select
  Resume   ' Resume execution at same line
  ' that caused the error.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Visual Basic Runtime Library (in Microsoft.VisualBasic.dll)

See also

- Try...Catch...Finally Statement
- Error Statement
- On Error Statement

Return Statement (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Returns control to the code that called a `Function`, `Sub`, `Get`, `Set`, or `Operator` procedure.

Syntax

```
Return
-or-
Return expression
```

Part

`expression`
Required in a `Function`, `Get`, or `Operator` procedure. Expression that represents the value to be returned to the calling code.

Remarks

In a `Sub` or `Set` procedure, the `Return` statement is equivalent to an `Exit Sub` or `Exit Property` statement, and `expression` must not be supplied.

In a `Function`, `Get`, or `Operator` procedure, the `Return` statement must include `expression`, and `expression` must evaluate to a data type that is convertible to the return type of the procedure. In a `Function` or `Get` procedure, you also have the alternative of assigning an expression to the procedure name to serve as the return value, and then executing an `Exit Function` or `Exit Property` statement. In an `Operator` procedure, you must use `Return expression`.

You can include as many `Return` statements as appropriate in the same procedure.

> **NOTE**
> The code in a `Finally` block runs after a `Return` statement in a `Try` or `Catch` block is encountered, but before that `Return` statement executes. A `Return` statement cannot be included in a `Finally` block.

Example

The following example uses the `Return` statement several times to return to the calling code when the procedure does not have to do anything else.

```
Public Function GetAgePhrase(ByVal age As Integer) As String
    If age > 60 Then Return "Senior"
    If age > 40 Then Return "Middle-aged"
    If age > 20 Then Return "Adult"
    If age > 12 Then Return "Teen-aged"
    If age > 4 Then Return "School-aged"
    If age > 1 Then Return "Toddler"
    Return "Infant"
End Function
```

See also

- Function Statement
- Sub Statement
- Get Statement
- Set Statement
- Operator Statement
- Property Statement
- Exit Statement
- Try...Catch...Finally Statement

Select...Case Statement (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Runs one of several groups of statements, depending on the value of an expression.

Syntax

```
Select [ Case ] testexpression
    [ Case expressionlist
        [ statements ] ]
    [ Case Else
        [ elsestatements ] ]
End Select
```

Parts

TERM	DEFINITION
`testexpression`	Required. Expression. Must evaluate to one of the elementary data types (`Boolean` , `Byte` , `Char` , `Date` , `Double` , `Decimal` , `Integer` , `Long` , `Object` , `SByte` , `Short` , `Single` , `String` , `UInteger` , `ULong` , and `UShort`).

TERM	DEFINITION
`expressionlist`	Required in a `Case` statement. List of expression clauses representing match values for `testexpression`. Multiple expression clauses are separated by commas. Each clause can take one of the following forms: - *expression1* `To` *expression2* - [`Is`] *comparisonoperator expression* - *expression* Use the `To` keyword to specify the boundaries of a range of match values for `testexpression`. The value of `expression1` must be less than or equal to the value of `expression2`. Use the `Is` keyword with a comparison operator (`=`, `<>`, `<`, `<=`, `>`, or `>=`) to specify a restriction on the match values for `testexpression`. If the `Is` keyword is not supplied, it is automatically inserted before *comparisonoperator*. The form specifying only `expression` is treated as a special case of the `Is` form where *comparisonoperator* is the equal sign (`=`). This form is evaluated as `testexpression = expression`. The expressions in `expressionlist` can be of any data type, provided they are implicitly convertible to the type of `testexpression` and the appropriate `comparisonoperator` is valid for the two types it is being used with.
`statements`	Optional. One or more statements following `Case` that run if `testexpression` matches any clause in `expressionlist`.
`elsestatements`	Optional. One or more statements following `Case Else` that run if `testexpression` does not match any clause in the `expressionlist` of any of the `Case` statements.
`End Select`	Terminates the definition of the `Select ... Case` construction.

Remarks

If `testexpression` matches any `Case` `expressionlist` clause, the statements following that `Case` statement run up to the next `Case`, `Case Else`, or `End Select` statement. Control then passes to the statement following `End Select`. If `testexpression` matches an `expressionlist` clause in more than one `Case` clause, only the statements following the first match run.

The `Case Else` statement is used to introduce the `elsestatements` to run if no match is found between the `testexpression` and an `expressionlist` clause in any of the other `Case` statements. Although not required, it is a good idea to have a `Case Else` statement in your `Select Case` construction to handle unforeseen `testexpression` values. If no `Case` `expressionlist` clause matches `testexpression` and there is no `Case Else` statement, control passes to the statement following `End Select`.

You can use multiple expressions or ranges in each `Case` clause. For example, the following line is valid.

```
Case 1 To 4, 7 To 9, 11, 13, Is > maxNumber
```

> **NOTE**
> The `Is` keyword used in the `Case` and `Case Else` statements is not the same as the Is Operator, which is used for object reference comparison.

You can specify ranges and multiple expressions for character strings. In the following example, `Case` matches any string that is exactly equal to "apples", has a value between "nuts" and "soup" in alphabetical order, or contains the exact same value as the current value of `testItem`.

```
Case "apples", "nuts" To "soup", testItem
```

The setting of `Option Compare` can affect string comparisons. Under `Option Compare Text`, the strings "Apples" and "apples" compare as equal, but under `Option Compare Binary`, they do not.

> **NOTE**
> A `Case` statement with multiple clauses can exhibit behavior known as *short-circuiting*. Visual Basic evaluates the clauses from left to right, and if one produces a match with `testexpression`, the remaining clauses are not evaluated. Short-circuiting can improve performance, but it can produce unexpected results if you are expecting every expression in `expressionlist` to be evaluated. For more information on short-circuiting, see Boolean Expressions.

If the code within a `Case` or `Case Else` statement block does not need to run any more of the statements in the block, it can exit the block by using the `Exit Select` statement. This transfers control immediately to the statement following `End Select`.

`Select Case` constructions can be nested. Each nested `Select Case` construction must have a matching `End Select` statement and must be completely contained within a single `Case` or `Case Else` statement block of the outer `Select Case` construction within which it is nested.

Example

The following example uses a `Select Case` construction to write a line corresponding to the value of the variable `number`. The second `Case` statement contains the value that matches the current value of `number`, so the statement that writes "Between 6 and 8, inclusive" runs.

```
Dim number As Integer = 8
Select Case number
    Case 1 To 5
        Debug.WriteLine("Between 1 and 5, inclusive")
        ' The following is the only Case clause that evaluates to True.
    Case 6, 7, 8
        Debug.WriteLine("Between 6 and 8, inclusive")
    Case 9 To 10
        Debug.WriteLine("Equal to 9 or 10")
    Case Else
        Debug.WriteLine("Not between 1 and 10, inclusive")
End Select
```

See also

- Choose

- End Statement
- If...Then...Else Statement
- Option Compare Statement
- Exit Statement

Set Statement (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Declares a `Set` property procedure used to assign a value to a property.

Syntax

```
[ <attributelist> ] [ accessmodifier ] Set (ByVal value [ As datatype ])
    [ statements ]
End Set
```

Parts

`attributelist`
Optional. See Attribute List.

`accessmodifier`
Optional on at most one of the `Get` and `Set` statements in this property. Can be one of the following:

- Protected
- Friend
- Private
- `Protected Friend`

See Access levels in Visual Basic.

`value`
Required. Parameter containing the new value for the property.

`datatype`
Required if `Option Strict` is `On`. Data type of the `value` parameter. The data type specified must be the same as the data type of the property where this `Set` statement is declared.

`statements`
Optional. One or more statements that run when the `Set` property procedure is called.

`End Set`
Required. Terminates the definition of the `Set` property procedure.

Remarks

Every property must have a `Set` property procedure unless the property is marked `ReadOnly`. The `Set` procedure is used to set the value of the property.

Visual Basic automatically calls a property's `Set` procedure when an assignment statement provides a value to be stored in the property.

Visual Basic passes a parameter to the `Set` procedure during property assignments. If you do not supply a parameter for `Set`, the integrated development environment (IDE) uses an implicit parameter named `value`. The parameter holds the value to be assigned to the property. You typically store this value in a private local

variable and return it whenever the `Get` procedure is called.

The body of the property declaration can contain only the property's `Get` and `Set` procedures between the Property Statement and the `End Property` statement. It cannot store anything other than those procedures. In particular, it cannot store the property's current value. You must store this value outside the property, because if you store it inside either of the property procedures, the other property procedure cannot access it. The usual approach is to store the value in a Private variable declared at the same level as the property. You must define a `Set` procedure inside the property to which it applies.

The `Set` procedure defaults to the access level of its containing property unless you use `accessmodifier` in the `Set` statement.

Rules

- **Mixed Access Levels.** If you are defining a read-write property, you can optionally specify a different access level for either the `Get` or the `Set` procedure, but not both. If you do this, the procedure access level must be more restrictive than the property's access level. For example, if the property is declared `Friend`, you can declare the `Set` procedure `Private`, but not `Public`.

  If you are defining a `WriteOnly` property, the `Set` procedure represents the entire property. You cannot declare a different access level for `Set`, because that would set two access levels for the property.

Behavior

- **Returning from a Property Procedure.** When the `Set` procedure returns to the calling code, execution continues following the statement that provided the value to be stored.

  `Set` property procedures can return using either the Return Statement or the Exit Statement.

  The `Exit Property` and `Return` statements cause an immediate exit from a property procedure. Any number of `Exit Property` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Property` and `Return` statements.

Example

The following example uses the `Set` statement to set the value of a property.

```
Class propClass
    Private propVal As Integer
    Property Prop1() As Integer
        Get
            Return propVal
        End Get
        Set(ByVal value As Integer)
            propVal = value
        End Set
    End Property
End Class
```

See also

- Get Statement
- Property Statement
- Sub Statement
- Property Procedures

Stop Statement (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Suspends execution.

Syntax

```
Stop
```

Remarks

You can place `Stop` statements anywhere in procedures to suspend execution. Using the `Stop` statement is similar to setting a breakpoint in the code.

The `Stop` statement suspends execution, but unlike `End`, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.

> **NOTE**
> If the `Stop` statement is encountered in code that is running outside of the integrated development environment (IDE), the debugger is invoked. This is true regardless of whether the code was compiled in debug or retail mode.

Example

This example uses the `Stop` statement to suspend execution for each iteration through the `For...Next` loop.

```
Dim i As Integer
For i = 1 To 10
    Debug.WriteLine(i)
    ' Stop during each iteration and wait for user to resume.
    Stop
Next i
```

See also

- End Statement

Structure Statement

4/28/2019 • 5 minutes to read • Edit Online

Declares the name of a structure and introduces the definition of the variables, properties, events, and procedures that the structure comprises.

Syntax

```
[ <attributelist> ] [ accessmodifier ] [ Shadows ] [ Partial ] _
Structure name [ ( Of typelist ) ]
    [ Implements interfacenames ]
    [ datamemberdeclarations ]
    [ methodmemberdeclarations ]
End Structure
```

Parts

TERM	DEFINITION
`attributelist`	Optional. See Attribute List.
`accessmodifier`	Optional. Can be one of the following: - Public - Protected - Friend - Private - Protected Friend - Private Protected See Access levels in Visual Basic.
`Shadows`	Optional. See Shadows.
`Partial`	Optional. Indicates a partial definition of the structure. See Partial.
`name`	Required. Name of this structure. See Declared Element Names.
`Of`	Optional. Specifies that this is a generic structure.
`typelist`	Required if you use the Of keyword. List of type parameters for this structure. See Type List.
`Implements`	Optional. Indicates that this structure implements the members of one or more interfaces. See Implements Statement.
`interfacenames`	Required if you use the `Implements` statement. The names of the interfaces this structure implements.

TERM	DEFINITION
`datamemberdeclarations`	Required. Zero or more `Const`, `Dim`, `Enum`, or `Event` statements declaring *data members* of the structure.
`methodmemberdeclarations`	Optional. Zero or more declarations of `Function`, `Operator`, `Property`, or `Sub` procedures, which serve as *method members* of the structure.
`End Structure`	Required. Terminates the `Structure` definition.

Remarks

The `Structure` statement defines a composite value type that you can customize. A *structure* is a generalization of the user-defined type (UDT) of previous versions of Visual Basic. For more information, see Structures.

Structures support many of the same features as classes. For example, structures can have properties and procedures, they can implement interfaces, and they can have parameterized constructors. However, there are significant differences between structures and classes in areas such as inheritance, declarations, and usage. Also, classes are reference types and structures are value types. For more information, see Structures and Classes.

You can use `Structure` only at namespace or module level. This means the *declaration context* for a structure must be a source file, namespace, class, structure, module, or interface, and cannot be a procedure or block. For more information, see Declaration Contexts and Default Access Levels.

Structures default to Friend access. You can adjust their access levels with the access modifiers. For more information, see Access levels in Visual Basic.

Rules

- **Nesting.** You can define one structure within another. The outer structure is called the *containing structure*, and the inner structure is called a *nested structure*. However, you cannot access a nested structure's members through the containing structure. Instead, you must declare a variable of the nested structure's data type.
- **Member Declaration.** You must declare every member of a structure. A structure member cannot be Protected or `Protected Friend` because nothing can inherit from a structure. The structure itself, however, can be `Protected` or `Protected Friend`.

  You can declare zero or more nonshared variables or nonshared, noncustom events in a structure. You cannot have only constants, properties, and procedures, even if some of them are nonshared.
- **Initialization.** You cannot initialize the value of any nonshared data member of a structure as part of its declaration. You must either initialize such a data member by means of a parameterized constructor on the structure, or assign a value to the member after you have created an instance of the structure.
- **Inheritance.** A structure cannot inherit from any type other than ValueType, from which all structures inherit. In particular, one structure cannot inherit from another.

  You cannot use the Inherits Statement in a structure definition, even to specify ValueType.
- **Implementation.** If the structure uses the Implements Statement, you must implement every member defined by every interface you specify in `interfacenames`.

- **Default Property.** A structure can specify at most one property as its *default property*, using the Default modifier. For more information, see Default.

Behavior

- **Access Level.** Within a structure, you can declare each member with its own access level. All structure members default to Public access. Note that if the structure itself has a more restricted access level, this automatically restricts access to its members, even if you adjust their access levels with the access modifiers.

- **Scope.** A structure is in scope throughout its containing namespace, class, structure, or module.

  The scope of every structure member is the entire structure.

- **Lifetime.** A structure does not itself have a lifetime. Rather, each instance of that structure has a lifetime independent of all other instances.

  The lifetime of an instance begins when it is created by a New Operator clause. It ends when the lifetime of the variable that holds it ends.

  You cannot extend the lifetime of a structure instance. An approximation to static structure functionality is provided by a module. For more information, see Module Statement.

  Structure members have lifetimes depending on how and where they are declared. For more information, see "Lifetime" in Class Statement.

- **Qualification.** Code outside a structure must qualify a member's name with the name of that structure.

  If code inside a nested structure makes an unqualified reference to a programming element, Visual Basic searches for the element first in the nested structure, then in its containing structure, and so on out to the outermost containing element. For more information, see References to Declared Elements.

- **Memory Consumption.** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration. If you need to control the storage layout of a structure, you can apply the StructLayoutAttribute attribute to the `Structure` statement.

Example

The following example uses the `Structure` statement to define a set of related data for an employee. It shows the use of `Public`, `Friend`, and `Private` members to reflect the sensitivity of the data items. It also shows procedure, property, and event members.

```
Public Structure employee
    ' Public members, accessible from throughout declaration region.
    Public firstName As String
    Public middleName As String
    Public lastName As String
    ' Friend members, accessible from anywhere within the same assembly.
    Friend employeeNumber As Integer
    Friend workPhone As Long
    ' Private members, accessible only from within the structure itself.
    Private homePhone As Long
    Private level As Integer
    Private salary As Double
    Private bonus As Double
    ' Procedure member, which can access structure's private members.
    Friend Sub CalculateBonus(ByVal rate As Single)
        bonus = salary * CDbl(rate)
    End Sub
    ' Property member to return employee's eligibility.
    Friend ReadOnly Property Eligible() As Boolean
        Get
            Return level >= 25
        End Get
    End Property
    ' Event member, raised when business phone number has changed.
    Public Event ChangedWorkPhone(ByVal newPhone As Long)
End Structure
```

See also

- Class Statement
- Interface Statement
- Module Statement
- Dim Statement
- Const Statement
- Enum Statement
- Event Statement
- Operator Statement
- Property Statement
- Structures and Classes

Sub Statement (Visual Basic)

4/26/2019 • 6 minutes to read • Edit Online

Declares the name, parameters, and code that define a `Sub` procedure.

Syntax

```
[ <attributelist> ] [ Partial ] [ accessmodifier ] [ proceduremodifiers ] [ Shared ] [ Shadows ] [
Async ]
Sub name [ (Of typeparamlist) ] [ (parameterlist) ] [ Implements implementslist | Handles eventlist
]
    [ statements ]
    [ Exit Sub ]
    [ statements ]
End Sub
```

Parts

- `attributelist`

  Optional. See Attribute List.

- `Partial`

  Optional. Indicates definition of a partial method. See Partial Methods.

- `accessmodifier`

  Optional. Can be one of the following:

  - Public
  - Protected
  - Friend
  - Private
  - Protected Friend
  - Private Protected

  See Access levels in Visual Basic.

- `proceduremodifiers`

  Optional. Can be one of the following:

  - Overloads
  - Overrides
  - Overridable
  - NotOverridable
  - MustOverride

- `MustOverride Overrides`
- `NotOverridable Overrides`

- `Shared`

  Optional. See Shared.

- `Shadows`

  Optional. See Shadows.

- `Async`

  Optional. See Async.

- `name`

  Required. Name of the procedure. See Declared Element Names. To create a constructor procedure for a class, set the name of a `Sub` procedure to the `New` keyword. For more information, see Object Lifetime: How Objects Are Created and Destroyed.

- `typeparamlist`

  Optional. List of type parameters for a generic procedure. See Type List.

- `parameterlist`

  Optional. List of local variable names representing the parameters of this procedure. See Parameter List.

- `Implements`

  Optional. Indicates that this procedure implements one or more `Sub` procedures, each one defined in an interface implemented by this procedure's containing class or structure. See Implements Statement.

- `implementslist`

  Required if `Implements` is supplied. List of `Sub` procedures being implemented.

  ```
  implementedprocedure [ , implementedprocedure ... ]
  ```

  Each `implementedprocedure` has the following syntax and parts:

  ```
  interface.definedname
  ```

PART	DESCRIPTION
`interface`	Required. Name of an interface implemented by this procedure's containing class or structure.
`definedname`	Required. Name by which the procedure is defined in `interface`.

- `Handles`

  Optional. Indicates that this procedure can handle one or more specific events. See Handles.

- `eventlist`

  Required if `Handles` is supplied. List of events this procedure handles.

`eventspecifier [ , eventspecifier ... ]`

Each `eventspecifier` has the following syntax and parts:

`eventvariable.event`

PART	DESCRIPTION
`eventvariable`	Required. Object variable declared with the data type of the class or structure that raises the event.
`event`	Required. Name of the event this procedure handles.

- `statements`

  Optional. Block of statements to run within this procedure.

- `End Sub`

  Terminates the definition of this procedure.

Remarks

All executable code must be inside a procedure. Use a `Sub` procedure when you don't want to return a value to the calling code. Use a `Function` procedure when you want to return a value.

Defining a Sub Procedure

You can define a `Sub` procedure only at the module level. The declaration context for a sub procedure must, therefore, be a class, a structure, a module, or an interface and can't be a source file, a namespace, a procedure, or a block. For more information, see Declaration Contexts and Default Access Levels.

`Sub` procedures default to public access. You can adjust their access levels by using the access modifiers.

If the procedure uses the `Implements` keyword, the containing class or structure must have an `Implements` statement that immediately follows its `Class` or `Structure` statement. The `Implements` statement must include each interface that's specified in `implementslist`. However, the name by which an interface defines the `Sub` (in `definedname`) doesn't have to match the name of this procedure (in `name`).

Returning from a Sub Procedure

When a `Sub` procedure returns to the calling code, execution continues with the statement after the statement that called it.

The following example shows a return from a `Sub` procedure.

```
Sub mySub(ByVal q As String)
    Return
End Sub
```

The `Exit Sub` and `Return` statements cause an immediate exit from a `Sub` procedure. Any number of `Exit Sub` and `Return` statements can appear anywhere in the procedure, and you can mix `Exit Sub` and `Return` statements.

Calling a Sub Procedure

You call a `Sub` procedure by using the procedure name in a statement and then following that name with its argument list in parentheses. You can omit the parentheses only if you don't supply any arguments. However, your code is more readable if you always include the parentheses.

A `Sub` procedure and a `Function` procedure can have parameters and perform a series of statements. However, a `Function` procedure returns a value, and a `Sub` procedure doesn't. Therefore, you can't use a `Sub` procedure in an expression.

You can use the `Call` keyword when you call a `Sub` procedure, but that keyword isn't recommended for most uses. For more information, see Call Statement.

Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, if your argument list includes expressions that call other procedures, you shouldn't assume that those expressions will be called in a particular order.

Async Sub Procedures

By using the Async feature, you can invoke asynchronous functions without using explicit callbacks or manually splitting your code across multiple functions or lambda expressions.

If you mark a procedure with the Async modifier, you can use the Await operator in the procedure. When control reaches an `Await` expression in the `Async` procedure, control returns to the caller, and progress in the procedure is suspended until the awaited task completes. When the task is complete, execution can resume in the procedure.

> **NOTE**
> An `Async` procedure returns to the caller when either the first awaited object that's not yet complete is encountered or the end of the `Async` procedure is reached, whichever occurs first.

You can also mark a Function Statement with the `Async` modifier. An `Async` function can have a return type of Task<TResult> or Task. An example later in this topic shows an `Async` function that has a return type of Task<TResult>.

`Async` `Sub` procedures are primarily used for event handlers, where a value can't be returned. An `Async` `Sub` procedure can't be awaited, and the caller of an `Async` `Sub` procedure can't catch exceptions that the `Sub` procedure throws.

An `Async` procedure can't declare any ByRef parameters.

For more information about `Async` procedures, see Asynchronous Programming with Async and Await, Control Flow in Async Programs, and Async Return Types.

Example

The following example uses the `Sub` statement to define the name, parameters, and code that form the body of a `Sub` procedure.

```
Sub ComputeArea(ByVal length As Double, ByVal width As Double)
    ' Declare local variable.
    Dim area As Double
    If length = 0 Or width = 0 Then
        ' If either argument = 0 then exit Sub immediately.
        Exit Sub
    End If
    ' Calculate area of rectangle.
    area = length * width
    ' Print area to Immediate window.
    Debug.WriteLine(area)
End Sub
```

Example

In the following example, `DelayAsync` is an `Async` `Function` that has a return type of Task<TResult>. `DelayAsync` has a `Return` statement that returns an integer. Therefore, the function declaration of `DelayAsync` must have a return type of `Task(Of Integer)`. Because the return type is `Task(Of Integer)`, the evaluation of the `Await` expression in `DoSomethingAsync` produces an integer, as the following statement shows: `Dim result As Integer = Await delayTask`.

The `startButton_Click` procedure is an example of an `Async Sub` procedure. Because `DoSomethingAsync` is an `Async` function, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `Await DoSomethingAsync()`. The `startButton_Click` `Sub` procedure must be defined with the `Async` modifier because it has an `Await` expression.

```
' Imports System.Diagnostics
' Imports System.Threading.Tasks

' This Click event is marked with the Async modifier.
Private Async Sub startButton_Click(sender As Object, e As RoutedEventArgs) Handles
startButton.Click
    Await DoSomethingAsync()
End Sub

Private Async Function DoSomethingAsync() As Task
    Dim delayTask As Task(Of Integer) = DelayAsync()
    Dim result As Integer = Await delayTask

    ' The previous two statements may be combined into
    ' the following statement.
    ' Dim result As Integer = Await DelayAsync()

    Debug.WriteLine("Result: " & result)
End Function

Private Async Function DelayAsync() As Task(Of Integer)
    Await Task.Delay(100)
    Return 5
End Function

'  Output:
'    Result: 5
```

See also

- Implements Statement
- Function Statement
- Parameter List

- Dim Statement
- Call Statement
- Of
- Parameter Arrays
- How to: Use a Generic Class
- Troubleshooting Procedures
- Partial Methods

SyncLock Statement

4/28/2019 • 5 minutes to read • Edit Online

Acquires an exclusive lock for a statement block before executing the block.

Syntax

```
SyncLock lockobject
    [ block ]
End SyncLock
```

Parts

`lockobject`
Required. Expression that evaluates to an object reference.

`block`
Optional. Block of statements that are to execute when the lock is acquired.

`End SyncLock`
Terminates a `SyncLock` block.

Remarks

The `SyncLock` statement ensures that multiple threads do not execute the statement block at the same time. `SyncLock` prevents each thread from entering the block until no other thread is executing it.

The most common use of `SyncLock` is to protect data from being updated by more than one thread simultaneously. If the statements that manipulate the data must go to completion without interruption, put them inside a `SyncLock` block.

A statement block protected by an exclusive lock is sometimes called a *critical section*.

Rules

- Branching. You cannot branch into a `SyncLock` block from outside the block.
- Lock Object Value. The value of `lockobject` cannot be `Nothing`. You must create the lock object before you use it in a `SyncLock` statement.

  You cannot change the value of `lockobject` while executing a `SyncLock` block. The mechanism requires that the lock object remain unchanged.
- You can't use the Await operator in a `SyncLock` block.

Behavior

- Mechanism. When a thread reaches the `SyncLock` statement, it evaluates the `lockobject` expression and suspends execution until it acquires an exclusive lock on the object returned by the expression. When another thread reaches the `SyncLock` statement, it does not acquire a lock until the first thread executes the `End SyncLock` statement.

- Protected Data. If `lockobject` is a `Shared` variable, the exclusive lock prevents a thread in any instance of the class from executing the `SyncLock` block while any other thread is executing it. This protects data that is shared among all the instances.

  If `lockobject` is an instance variable (not `Shared`), the lock prevents a thread running in the current instance from executing the `SyncLock` block at the same time as another thread in the same instance. This protects data maintained by the individual instance.

- Acquisition and Release. A `SyncLock` block behaves like a `Try...Finally` construction in which the `Try` block acquires an exclusive lock on `lockobject` and the `Finally` block releases it. Because of this, the `SyncLock` block guarantees release of the lock, no matter how you exit the block. This is true even in the case of an unhandled exception.

- Framework Calls. The `SyncLock` block acquires and releases the exclusive lock by calling the `Enter` and `Exit` methods of the `Monitor` class in the System.Threading namespace.

Programming Practices

The `lockobject` expression should always evaluate to an object that belongs exclusively to your class. You should declare a `Private` object variable to protect data belonging to the current instance, or a `Private Shared` object variable to protect data common to all instances.

You should not use the `Me` keyword to provide a lock object for instance data. If code external to your class has a reference to an instance of your class, it could use that reference as a lock object for a `SyncLock` block completely different from yours, protecting different data. In this way, your class and the other class could block each other from executing their unrelated `SyncLock` blocks. Similarly locking on a string can be problematic since any other code in the process using the same string will share the same lock.

You should also not use the `Me.GetType` method to provide a lock object for shared data. This is because `GetType` always returns the same `Type` object for a given class name. External code could call `GetType` on your class and obtain the same lock object you are using. This would result in the two classes blocking each other from their `SyncLock` blocks.

Examples

Description

The following example shows a class that maintains a simple list of messages. It holds the messages in an array and the last used element of that array in a variable. The `addAnotherMessage` procedure increments the last element and stores the new message. Those two operations are protected by the `SyncLock` and `End SyncLock` statements, because once the last element has been incremented, the new message must be stored before any other thread can increment the last element again.

If the `simpleMessageList` class shared one list of messages among all its instances, the variables `messagesList` and `messagesLast` would be declared as `Shared`. In this case, the variable `messagesLock` should also be `Shared`, so that there would be a single lock object used by every instance.

Code

```
Class simpleMessageList
    Public messagesList() As String = New String(50) {}
    Public messagesLast As Integer = -1
    Private messagesLock As New Object
    Public Sub addAnotherMessage(ByVal newMessage As String)
        SyncLock messagesLock
            messagesLast += 1
            If messagesLast < messagesList.Length Then
                messagesList(messagesLast) = newMessage
            End If
        End SyncLock
    End Sub
End Class
```

Description

The following example uses threads and `SyncLock`. As long as the `SyncLock` statement is present, the statement block is a critical section and `balance` never becomes a negative number. You can comment out the `SyncLock` and `End SyncLock` statements to see the effect of leaving out the `SyncLock` keyword.

Code

```
Imports System.Threading

Module Module1

    Class Account
        Dim thisLock As New Object
        Dim balance As Integer

        Dim r As New Random()

        Public Sub New(ByVal initial As Integer)
            balance = initial
        End Sub

        Public Function Withdraw(ByVal amount As Integer) As Integer
            ' This condition will never be true unless the SyncLock statement
            ' is commented out:
            If balance < 0 Then
                Throw New Exception("Negative Balance")
            End If

            ' Comment out the SyncLock and End SyncLock lines to see
            ' the effect of leaving out the SyncLock keyword.
            SyncLock thisLock
                If balance >= amount Then
                    Console.WriteLine("Balance before Withdrawal :  " & balance)
                    Console.WriteLine("Amount to Withdraw        : -" & amount)
                    balance = balance - amount
                    Console.WriteLine("Balance after Withdrawal  :  " & balance)
                    Return amount
                Else
                    ' Transaction rejected.
                    Return 0
                End If
            End SyncLock
        End Function

        Public Sub DoTransactions()
            For i As Integer = 0 To 99
                Withdraw(r.Next(1, 100))
            Next
        End Sub
    End Class

    Sub Main()
        Dim threads(10) As Thread
        Dim acc As New Account(1000)

        For i As Integer = 0 To 9
            Dim t As New Thread(New ThreadStart(AddressOf acc.DoTransactions))
            threads(i) = t
        Next

        For i As Integer = 0 To 9
            threads(i).Start()
        Next
    End Sub

End Module
```

Comments

See also

- System.Threading.Monitor

- System.Threading.Interlocked
- Overview of synchronization primitives

Then Statement

4/2/2019 • 2 minutes to read • Edit Online

Introduces a statement block to be compiled or executed if a tested condition is true.

Remarks

The `Then` keyword can be used in these contexts:

#If...Then...#Else Directive

If...Then...Else Statement

See also

- Keywords

Throw Statement (Visual Basic)

8/7/2019 • 2 minutes to read • Edit Online

Throws an exception within a procedure.

Syntax

```
Throw [ expression ]
```

Part

`expression`
Provides information about the exception to be thrown. Optional when residing in a `Catch` statement, otherwise required.

Remarks

The `Throw` statement throws an exception that you can handle with structured exception-handling code (`Try` ... `Catch` ... `Finally`) or unstructured exception-handling code (`On Error GoTo`). You can use the `Throw` statement to trap errors within your code because Visual Basic moves up the call stack until it finds the appropriate exception-handling code.

A `Throw` statement with no expression can only be used in a `Catch` statement, in which case the statement rethrows the exception currently being handled by the `Catch` statement.

The `Throw` statement resets the call stack for the `expression` exception. If `expression` is not provided, the call stack is left unchanged. You can access the call stack for the exception through the StackTrace property.

Example

The following code uses the `Throw` statement to throw an exception:

```
' Throws a new exception.
Throw New System.Exception("An exception has occurred.")
```

See also

- Try...Catch...Finally Statement
- On Error Statement

Try...Catch...Finally Statement (Visual Basic)

5/15/2019 • 13 minutes to read • Edit Online

Provides a way to handle some or all possible errors that may occur in a given block of code, while still running code.

Syntax

```
Try
    [ tryStatements ]
    [ Exit Try ]
[ Catch [ exception [ As type ] ] [ When expression ]
    [ catchStatements ]
    [ Exit Try ] ]
[ Catch ... ]
[ Finally
    [ finallyStatements ] ]
End Try
```

Parts

TERM	DEFINITION
`tryStatements`	Optional. Statement(s) where an error can occur. Can be a compound statement.
`Catch`	Optional. Multiple `Catch` blocks permitted. If an exception occurs when processing the `Try` block, each `Catch` statement is examined in textual order to determine whether it handles the exception, with `exception` representing the exception that has been thrown.
`exception`	Optional. Any variable name. The initial value of `exception` is the value of the thrown error. Used with `Catch` to specify the error caught. If omitted, the `Catch` statement catches any exception.
`type`	Optional. Specifies the type of class filter. If the value of `exception` is of the type specified by `type` or of a derived type, the identifier becomes bound to the exception object.
`When`	Optional. A `Catch` statement with a `When` clause catches exceptions only when `expression` evaluates to `True`. A `When` clause is applied only after checking the type of the exception, and `expression` may refer to the identifier representing the exception.

TERM	DEFINITION
`expression`	Optional. Must be implicitly convertible to `Boolean`. Any expression that describes a generic filter. Typically used to filter by error number. Used with `When` keyword to specify circumstances under which the error is caught.
`catchStatements`	Optional. Statement(s) to handle errors that occur in the associated `Try` block. Can be a compound statement.
`Exit Try`	Optional. Keyword that breaks out of the `Try...Catch...Finally` structure. Execution resumes with the code immediately following the `End Try` statement. The `Finally` statement will still be executed. Not allowed in `Finally` blocks.
`Finally`	Optional. A `Finally` block is always executed when execution leaves any part of the `Try...Catch` statement.
`finallyStatements`	Optional. Statement(s) that are executed after all other error processing has occurred.
`End Try`	Terminates the `Try...Catch...Finally` structure.

Remarks

If you expect that a particular exception might occur during a particular section of code, put the code in a `Try` block and use a `Catch` block to retain control and handle the exception if it occurs.

A `Try…Catch` statement consists of a `Try` block followed by one or more `Catch` clauses, which specify handlers for various exceptions. When an exception is thrown in a `Try` block, Visual Basic looks for the `Catch` statement that handles the exception. If a matching `Catch` statement is not found, Visual Basic examines the method that called the current method, and so on up the call stack. If no `Catch` block is found, Visual Basic displays an unhandled exception message to the user and stops execution of the program.

You can use more than one `Catch` statement in a `Try…Catch` statement. If you do this, the order of the `Catch` clauses is significant because they are examined in order. Catch the more specific exceptions before the less specific ones.

The following `Catch` statement conditions are the least specific, and will catch all exceptions that derive from the Exception class. You should ordinarily use one of these variations as the last `Catch` block in the `Try...Catch...Finally` structure, after catching all the specific exceptions you expect. Control flow can never reach a `Catch` block that follows either of these variations.

- The `type` is `Exception`, for example: `Catch ex As Exception`
- The statement has no `exception` variable, for example: `Catch`

When a `Try…Catch…Finally` statement is nested in another `Try` block, Visual Basic first examines each `Catch` statement in the innermost `Try` block. If no matching `Catch` statement is found, the search proceeds to the `Catch` statements of the outer `Try…Catch…Finally` block.

Local variables from a `Try` block are not available in a `Catch` block because they are separate blocks. If you want to use a variable in more than one block, declare the variable outside the `Try...Catch...Finally` structure.

TIP

The `Try…Catch…Finally` statement is available as an IntelliSense code snippet. In the Code Snippets Manager, expand **Code Patterns - If, For Each, Try Catch, Property, etc**, and then **Error Handling (Exceptions)**. For more information, see Code Snippets.

Finally block

If you have one or more statements that must run before you exit the `Try` structure, use a `Finally` block. Control passes to the `Finally` block just before it passes out of the `Try…Catch` structure. This is true even if an exception occurs anywhere inside the `Try` structure.

A `Finally` block is useful for running any code that must execute even if there is an exception. Control is passed to the `Finally` block regardless of how the `Try...Catch` block exits.

The code in a `Finally` block runs even if your code encounters a `Return` statement in a `Try` or `Catch` block. Control does not pass from a `Try` or `Catch` block to the corresponding `Finally` block in the following cases:

- An End Statement is encountered in the `Try` or `Catch` block.
- A StackOverflowException is thrown in the `Try` or `Catch` block.

It is not valid to explicitly transfer execution into a `Finally` block. Transferring execution out of a `Finally` block is not valid, except through an exception.

If a `Try` statement does not contain at least one `Catch` block, it must contain a `Finally` block.

TIP

If you do not have to catch specific exceptions, the `Using` statement behaves like a `Try…Finally` block, and guarantees disposal of the resources, regardless of how you exit the block. This is true even with an unhandled exception. For more information, see Using Statement.

Exception argument

The `Catch` block `exception` argument is an instance of the Exception class or a class that derives from the `Exception` class. The `Exception` class instance corresponds to the error that occurred in the `Try` block.

The properties of the `Exception` object help to identify the cause and location of an exception. For example, the StackTrace property lists the called methods that led to the exception, helping you find where the error occurred in the code. Message returns a message that describes the exception. HelpLink returns a link to an associated Help file. InnerException returns the `Exception` object that caused the current exception, or it returns `Nothing` if there is no original `Exception`.

Considerations when using a Try…Catch statement

Use a `Try…Catch` statement only to signal the occurrence of unusual or unanticipated program events. Reasons for this include the following:

- Catching exceptions at runtime creates additional overhead, and is likely to be slower than pre-checking to avoid exceptions.
- If a `Catch` block is not handled correctly, the exception might not be reported correctly to users.

- Exception handling makes a program more complex.

You do not always need a `Try…Catch` statement to check for a condition that is likely to occur. The following example checks whether a file exists before trying to open it. This reduces the need for catching an exception thrown by the OpenText method.

```
Private Sub TextFileExample(ByVal filePath As String)

    ' Verify that the file exists.
    If System.IO.File.Exists(filePath) = False Then
        Console.Write("File Not Found: " & filePath)
    Else
        ' Open the text file and display its contents.
        Dim sr As System.IO.StreamReader =
            System.IO.File.OpenText(filePath)

        Console.Write(sr.ReadToEnd)

        sr.Close()
    End If
End Sub
```

Ensure that code in `Catch` blocks can properly report exceptions to users, whether through thread-safe logging or appropriate messages. Otherwise, exceptions might remain unknown.

Async methods

If you mark a method with the Async modifier, you can use the Await operator in the method. A statement with the `Await` operator suspends execution of the method until the awaited task completes. The task represents ongoing work. When the task that's associated with the `Await` operator finishes, execution resumes in the same method. For more information, see Control Flow in Async Programs.

A task returned by an Async method may end in a faulted state, indicating that it completed due to an unhandled exception. A task may also end in a canceled state, which results in an `OperationCanceledException` being thrown out of the await expression. To catch either type of exception, place the `Await` expression that's associated with the task in a `Try` block, and catch the exception in the `Catch` block. An example is provided later in this topic.

A task can be in a faulted state because multiple exceptions were responsible for its faulting. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, the caught exception is only one of the exceptions, and you can't predict which exception will be caught. An example is provided later in this topic.

An `Await` expression can't be inside a `Catch` block or `Finally` block.

Iterators

An iterator function or `Get` accessor performs a custom iteration over a collection. An iterator uses a Yield statement to return each element of the collection one at a time. You call an iterator function by using a For Each…Next Statement.

A `Yield` statement can be inside a `Try` block. A `Try` block that contains a `Yield` statement can have `Catch` blocks, and can have a `Finally` block. See the "Try Blocks in Visual Basic" section of Iterators for an example.

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (outside of the iterator function) throws an exception, a `Catch` block in the iterator

function is not executed, but a `Finally` block in the iterator function is executed. A `Catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Partial-trust situations

In partial-trust situations, such as an application hosted on a network share, `Try...Catch...Finally` does not catch security exceptions that occur before the method that contains the call is invoked. The following example, when you put it on a server share and run from there, produces the error "System.Security.SecurityException: Request Failed." For more information about security exceptions, see the SecurityException class.

```
Try
    Process.Start("http://www.microsoft.com")
Catch ex As Exception
    MsgBox("Can't load Web page" & vbCrLf & ex.Message)
End Try
```

In such a partial-trust situation, you have to put the `Process.Start` statement in a separate `Sub`. The initial call to the `Sub` will fail. This enables `Try...Catch` to catch it before the `Sub` that contains `Process.Start` is started and the security exception produced.

Examples

The structure of Try...Catch...Finally

The following example illustrates the structure of the `Try...Catch...Finally` statement.

```
Public Sub TryExample()
    ' Declare variables.
    Dim x As Integer = 5
    Dim y As Integer = 0

    ' Set up structured error handling.
    Try
        ' Cause a "Divide by Zero" exception.
        x = x \ y

        ' This statement does not execute because program
        ' control passes to the Catch block when the
        ' exception occurs.
        MessageBox.Show("end of Try block")
    Catch ex As Exception
        ' Show the exception's message.
        MessageBox.Show(ex.Message)

        ' Show the stack trace, which is a list of methods
        ' that are currently executing.
        MessageBox.Show("Stack Trace: " & vbCrLf & ex.StackTrace)
    Finally
        ' This line executes whether or not the exception occurs.
        MessageBox.Show("in Finally block")
    End Try
End Sub
```

Exception in a method called from a Try block

In the following example, the `CreateException` method throws a `NullReferenceException`. The code that generates the exception is not in a `Try` block. Therefore, the `CreateException` method does not handle the exception. The `RunSample` method does handle the exception because the call to the `CreateException` method is in a `Try` block.

The example includes `Catch` statements for several types of exceptions, ordered from the most specific to the most general.

```
Public Sub RunSample()
    Try
        CreateException()
    Catch ex As System.IO.IOException
        ' Code that reacts to IOException.
    Catch ex As NullReferenceException
        MessageBox.Show("NullReferenceException: " & ex.Message)
        MessageBox.Show("Stack Trace: " & vbCrLf & ex.StackTrace)
    Catch ex As Exception
        ' Code that reacts to any other exception.
    End Try
End Sub

Private Sub CreateException()
    ' This code throws a NullReferenceException.
    Dim obj = Nothing
    Dim prop = obj.Name

    ' This code also throws a NullReferenceException.
    'Throw New NullReferenceException("Something happened.")
End Sub
```

The Catch When statement

The following example shows how to use a `Catch When` statement to filter on a conditional expression. If the conditional expression evaluates to `True`, the code in the `Catch` block runs.

```
Private Sub WhenExample()
    Dim i As Integer = 5

    Try
        Throw New ArgumentException()
    Catch e As OverflowException When i = 5
        Console.WriteLine("First handler")
    Catch e As ArgumentException When i = 4
        Console.WriteLine("Second handler")
    Catch When i = 5
        Console.WriteLine("Third handler")
    End Try
End Sub
' Output: Third handler
```

Nested Try statements

The following example has a `Try…Catch` statement that is contained in a `Try` block. The inner `Catch` block throws an exception that has its `InnerException` property set to the original exception. The outer `Catch` block reports its own exception and the inner exception.

```
Private Sub InnerExceptionExample()
    Try
        Try
            ' Set a reference to a StringBuilder.
            ' The exception below does not occur if the commented
            ' out statement is used instead.
            Dim sb As System.Text.StringBuilder
            'Dim sb As New System.Text.StringBuilder

            ' Cause a NullReferenceException.
            sb.Append("text")
        Catch ex As Exception
            ' Throw a new exception that has the inner exception
            ' set to the original exception.
            Throw New ApplicationException("Something happened :(", ex)
        End Try
    Catch ex2 As Exception
        ' Show the exception.
        Console.WriteLine("Exception: " & ex2.Message)
        Console.WriteLine(ex2.StackTrace)

        ' Show the inner exception, if one is present.
        If ex2.InnerException IsNot Nothing Then
            Console.WriteLine("Inner Exception: " & ex2.InnerException.Message)
            Console.WriteLine(ex2.StackTrace)
        End If
    End Try
End Sub
```

Exception handling for async methods

The following example illustrates exception handling for async methods. To catch an exception that applies to an async task, the `Await` expression is in a `Try` block of the caller, and the exception is caught in the `Catch` block.

Uncomment the `Throw New Exception` line in the example to demonstrate exception handling. The exception is caught in the `Catch` block, the task's `IsFaulted` property is set to `True`, and the task's `Exception.InnerException` property is set to the exception.

Uncomment the `Throw New OperationCancelledException` line to demonstrate what happens when you cancel an asynchronous process. The exception is caught in the `Catch` block, and the task's `IsCanceled` property is set to `True`. However, under some conditions that don't apply to this example, `IsFaulted` is set to `True` and `IsCanceled` is set to `False`.

```
Public Async Function DoSomethingAsync() As Task
    Dim theTask As Task(Of String) = DelayAsync()

    Try
        Dim result As String = Await theTask
        Debug.WriteLine("Result: " & result)
    Catch ex As Exception
        Debug.WriteLine("Exception Message: " & ex.Message)
    End Try

    Debug.WriteLine("Task IsCanceled: " & theTask.IsCanceled)
    Debug.WriteLine("Task IsFaulted:  " & theTask.IsFaulted)
    If theTask.Exception IsNot Nothing Then
        Debug.WriteLine("Task Exception Message: " &
            theTask.Exception.Message)
        Debug.WriteLine("Task Inner Exception Message: " &
            theTask.Exception.InnerException.Message)
    End If
End Function

Private Async Function DelayAsync() As Task(Of String)
    Await Task.Delay(100)

    ' Uncomment each of the following lines to
    ' demonstrate exception handling.

    'Throw New OperationCanceledException("canceled")
    'Throw New Exception("Something happened.")
    Return "Done"
End Function


' Output when no exception is thrown in the awaited method:
'   Result: Done
'   Task IsCanceled: False
'   Task IsFaulted:  False

' Output when an Exception is thrown in the awaited method:
'   Exception Message: Something happened.
'   Task IsCanceled: False
'   Task IsFaulted:  True
'   Task Exception Message: One or more errors occurred.
'   Task Inner Exception Message: Something happened.

' Output when an OperationCanceledException or TaskCanceledException
' is thrown in the awaited method:
'   Exception Message: canceled
'   Task IsCanceled: True
'   Task IsFaulted:  False
```

Handling multiple exceptions in async methods

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `Try` block has the `Await` expression for the task that Task.WhenAll returned. The task is complete when the three tasks to which Task.WhenAll is applied are complete.

Each of the three tasks causes an exception. The `Catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that `Task.WhenAll` returned.

```
Public Async Function DoMultipleAsync() As Task
    Dim theTask1 As Task = ExcAsync(info:="First Task")
    Dim theTask2 As Task = ExcAsync(info:="Second Task")
    Dim theTask3 As Task = ExcAsync(info:="Third Task")

    Dim allTasks As Task = Task.WhenAll(theTask1, theTask2, theTask3)

    Try
        Await allTasks
    Catch ex As Exception
        Debug.WriteLine("Exception: " & ex.Message)
        Debug.WriteLine("Task IsFaulted: " & allTasks.IsFaulted)
        For Each inEx In allTasks.Exception.InnerExceptions
            Debug.WriteLine("Task Inner Exception: " + inEx.Message)
        Next
    End Try
End Function

Private Async Function ExcAsync(info As String) As Task
    Await Task.Delay(100)

    Throw New Exception("Error-" & info)
End Function

' Output:
'   Exception: Error-First Task
'   Task IsFaulted: True
'   Task Inner Exception: Error-First Task
'   Task Inner Exception: Error-Second Task
'   Task Inner Exception: Error-Third Task
```

See also

- Err
- Exception
- Exit Statement
- On Error Statement
- Best Practices for Using Code Snippets
- Exception Handling
- Throw Statement

Using Statement (Visual Basic)

8/22/2019 • 4 minutes to read • Edit Online

Declares the beginning of a `Using` block and optionally acquires the system resources that the block controls.

Syntax

```
Using { resourcelist | resourceexpression }
    [ statements ]
End Using
```

Parts

TERM	DEFINITION
`resourcelist`	Required if you do not supply `resourceexpression`. List of one or more system resources that this `Using` block controls, separated by commas.
`resourceexpression`	Required if you do not supply `resourcelist`. Reference variable or expression referring to a system resource to be controlled by this `Using` block.
`statements`	Optional. Block of statements that the `Using` block runs.
`End Using`	Required. Terminates the definition of the `Using` block and disposes of all the resources that it controls.

Each resource in the `resourcelist` part has the following syntax and parts:

`resourcename As New resourcetype [ ( [ arglist ] ) ]`

-or-

`resourcename As resourcetype = resourceexpression`

resourcelist Parts

TERM	DEFINITION
`resourcename`	Required. Reference variable that refers to a system resource that the `Using` block controls.
`New`	Required if the `Using` statement acquires the resource. If you have already acquired the resource, use the second syntax alternative.
`resourcetype`	Required. The class of the resource. The class must implement the IDisposable interface.

TERM	DEFINITION
`arglist`	Optional. List of arguments you are passing to the constructor to create an instance of `resourcetype`. See Parameter List.
`resourceexpression`	Required. Variable or expression referring to a system resource satisfying the requirements of `resourcetype`. If you use the second syntax alternative, you must acquire the resource before passing control to the `Using` statement.

Remarks

Sometimes your code requires an unmanaged resource, such as a file handle, a COM wrapper, or a SQL connection. A `Using` block guarantees the disposal of one or more such resources when your code is finished with them. This makes them available for other code to use.

Managed resources are disposed of by the .NET Framework garbage collector (GC) without any extra coding on your part. You do not need a `Using` block for managed resources. However, you can still use a `Using` block to force the disposal of a managed resource instead of waiting for the garbage collector.

A `Using` block has three parts: acquisition, usage, and disposal.

- *Acquisition* means creating a variable and initializing it to refer to the system resource. The `Using` statement can acquire one or more resources, or you can acquire exactly one resource before entering the block and supply it to the `Using` statement. If you supply `resourceexpression`, you must acquire the resource before passing control to the `Using` statement.
- *Usage* means accessing the resources and performing actions with them. The statements between `Using` and `End Using` represent the usage of the resources.
- *Disposal* means calling the Dispose method on the object in `resourcename`. This allows the object to cleanly terminate its resources. The `End Using` statement disposes of the resources under the `Using` block's control.

Behavior

A `Using` block behaves like a `Try`...`Finally` construction in which the `Try` block uses the resources and the `Finally` block disposes of them. Because of this, the `Using` block guarantees disposal of the resources, no matter how you exit the block. This is true even in the case of an unhandled exception, except for a StackOverflowException.

The scope of every resource variable acquired by the `Using` statement is limited to the `Using` block.

If you specify more than one system resource in the `Using` statement, the effect is the same as if you nested `Using` blocks one within another.

If `resourcename` is `Nothing`, no call to Dispose is made, and no exception is thrown.

Structured Exception Handling Within a Using Block

If you need to handle an exception that might occur within the `Using` block, you can add a complete `Try`...`Finally` construction to it. If you need to handle the case where the `Using` statement is not successful in acquiring a resource, you can test to see if `resourcename` is `Nothing`.

Structured Exception Handling Instead of a Using Block

If you need finer control over the acquisition of the resources, or you need additional code in the `Finally` block, you can rewrite the `Using` block as a `Try`...`Finally` construction. The following example shows skeleton `Try` and `Using` constructions that are equivalent in the acquisition and disposal of `resource`.

```
Using resource As New resourceType
    ' Insert code to work with resource.
End Using

' For the acquisition and disposal of resource, the following
' Try construction is equivalent to the Using block.
Dim resource As New resourceType
Try
    ' Insert code to work with resource.
Finally
    If resource IsNot Nothing Then
        resource.Dispose()
    End If
End Try
```

NOTE

The code inside the `Using` block should not assign the object in `resourcename` to another variable. When you exit the `Using` block, the resource is disposed, and the other variable cannot access the resource to which it points.

Example

The following example creates a file that is named log.txt and writes two lines of text to the file. The example also reads that same file and displays the lines of text.

Because the TextWriter and TextReader classes implement the IDisposable interface, the code can use `Using` statements to ensure that the file is correctly closed after the write and read operations.

```
Private Sub WriteFile()
    Using writer As System.IO.TextWriter = System.IO.File.CreateText("log.txt")
        writer.WriteLine("This is line one.")
        writer.WriteLine("This is line two.")
    End Using
End Sub

Private Sub ReadFile()
    Using reader As System.IO.TextReader = System.IO.File.OpenText("log.txt")
        Dim line As String

        line = reader.ReadLine()
        Do Until line Is Nothing
            Console.WriteLine(line)
            line = reader.ReadLine()
        Loop
    End Using
End Sub
```

See also

- IDisposable
- Try...Catch...Finally Statement

- How to: Dispose of a System Resource

While...End While Statement (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Runs a series of statements as long as a given condition is `True`.

Syntax

```
While condition
    [ statements ]
    [ Continue While ]
    [ statements ]
    [ Exit While ]
    [ statements ]
End While
```

Parts

TERM	DEFINITION
`condition`	Required. `Boolean` expression. If `condition` is `Nothing`, Visual Basic treats it as `False`.
`statements`	Optional. One or more statements following `While`, which run every time `condition` is `True`.
`Continue While`	Optional. Transfers control to the next iteration of the `While` block.
`Exit While`	Optional. Transfers control out of the `While` block.
`End While`	Required. Terminates the definition of the `While` block.

Remarks

Use a `While...End While` structure when you want to repeat a set of statements an indefinite number of times, as long as a condition remains `True`. If you want more flexibility with where you test the condition or what result you test it for, you might prefer the Do...Loop Statement. If you want to repeat the statements a set number of times, the For...Next Statement is usually a better choice.

> **NOTE**
>
> The `While` keyword is also used in the Do...Loop Statement, the Skip While Clause and the Take While Clause.

If `condition` is `True`, all of the `statements` run until the `End While` statement is encountered. Control then returns to the `While` statement, and `condition` is again checked. If `condition` is still `True`, the process is repeated. If it's `False`, control passes to the statement that follows the `End While` statement.

The `While` statement always checks the condition before it starts the loop. Looping continues while the

condition remains `True`. If `condition` is `False` when you first enter the loop, it doesn't run even once.

The `condition` usually results from a comparison of two values, but it can be any expression that evaluates to a Boolean Data Type value (`True` or `False`). This expression can include a value of another data type, such as a numeric type, that has been converted to `Boolean`.

You can nest `While` loops by placing one loop within another. You can also nest different kinds of control structures within one another. For more information, see Nested Control Structures.

Exit While

The Exit While statement can provide another way to exit a `While` loop. `Exit While` immediately transfers control to the statement that follows the `End While` statement.

You typically use `Exit While` after some condition is evaluated (for example, in an `If...Then...Else` structure). You might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. You can use `Exit While` when you test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. You can then use `Exit While` to escape the loop.

You can place any number of `Exit While` statements anywhere in the `While` loop.

When used within nested `While` loops, `Exit While` transfers control out of the innermost loop and into the next higher level of nesting.

The `Continue While` statement immediately transfers control to the next iteration of the loop. For more information, see Continue Statement.

Example

In the following example, the statements in the loop continue to run until the `index` variable is greater than 10.

```
Dim index As Integer = 0
While index <= 10
    Debug.Write(index.ToString & " ")
    index += 1
End While

Debug.WriteLine("")
' Output: 0 1 2 3 4 5 6 7 8 9 10
```

Example

The following example illustrates the use of the `Continue While` and `Exit While` statements.

```
Dim index As Integer = 0
While index < 100000
    index += 1

    ' If index is between 5 and 7, continue
    ' with the next iteration.
    If index >= 5 And index <= 8 Then
        Continue While
    End If

    ' Display the index.
    Debug.Write(index.ToString & " ")

    ' If index is 10, exit the loop.
    If index = 10 Then
        Exit While
    End If
End While

Debug.WriteLine("")
' Output: 1 2 3 4 9 10
```

Example

The following example reads all lines in a text file. The OpenText method opens the file and returns a StreamReader that reads the characters. In the `While` condition, the Peek method of the `StreamReader` determines whether the file contains additional characters.

```
Private Sub ShowText(ByVal textFilePath As String)
    If System.IO.File.Exists(textFilePath) = False Then
        Debug.WriteLine("File Not Found: " & textFilePath)
    Else
        Dim sr As System.IO.StreamReader = System.IO.File.OpenText(textFilePath)

        While sr.Peek() >= 0
            Debug.WriteLine(sr.ReadLine())
        End While

        sr.Close()
    End If
End Sub
```

See also

- Loop Structures
- Do...Loop Statement
- For...Next Statement
- Boolean Data Type
- Nested Control Structures
- Exit Statement
- Continue Statement

With...End With Statement (Visual Basic)

8/27/2019 • 3 minutes to read • Edit Online

Executes a series of statements that repeatedly refer to a single object or structure so that the statements can use a simplified syntax when accessing members of the object or structure. When using a structure, you can only read the values of members or invoke methods, and you get an error if you try to assign values to members of a structure used in a `With...End With` statement.

Syntax

```
With objectExpression
    [ statements ]
End With
```

Parts

TERM	DEFINITION
`objectExpression`	Required. An expression that evaluates to an object. The expression may be arbitrarily complex and is evaluated only once. The expression can evaluate to any data type, including elementary types.
`statements`	Optional. One or more statements between `With` and `End With` that may refer to members of an object that's produced by the evaluation of `objectExpression`.
`End With`	Required. Terminates the definition of the `With` block.

Remarks

By using `With...End With`, you can perform a series of statements on a specified object without specifying the name of the object multiple times. Within a `With` statement block, you can specify a member of the object starting with a period, as if the `With` statement object preceded it.

For example, to change multiple properties on a single object, place the property assignment statements inside the `With...End With` block, referring to the object only once instead of once for each property assignment.

If your code accesses the same object in multiple statements, you gain the following benefits by using the `With` statement:

- You don't need to evaluate a complex expression multiple times or assign the result to a temporary variable to refer to its members multiple times.
- You make your code more readable by eliminating repetitive qualifying expressions.

The data type of `objectExpression` can be any class or structure type or even a Visual Basic elementary type such as `Integer`. If `objectExpression` results in anything other than an object, you can only read the values of its members or invoke methods, and you get an error if you try to assign values to members of a structure used in a `With...End With` statement. This is the same error you would get if you invoked a method that returned a

structure and immediately accessed and assigned a value to a member of the function's result, such as `GetAPoint().x = 1`. The problem in both cases is that the structure exists only on the call stack, and there is no way a modified structure member in these situations can write to a location such that any other code in the program can observe the change.

The `objectExpression` is evaluated once, upon entry into the block. You can't reassign the `objectExpression` from within the `With` block.

Within a `With` block, you can access the methods and properties of only the specified object without qualifying them. You can use methods and properties of other objects, but you must qualify them with their object names.

You can place one `With...End With` statement within another. Nested `With...End With` statements may be confusing if the objects that are being referred to aren't clear from context. You must provide a fully qualified reference to an object that's in an outer `With` block when the object is referenced from within an inner `With` block.

You can't branch into a `With` statement block from outside the block.

Unless the block contains a loop, the statements run only once. You can nest different kinds of control structures. For more information, see Nested Control Structures.

> **NOTE**
>
> You can use the `With` keyword in object initializers also. For more information and examples, see Object Initializers: Named and Anonymous Types and Anonymous Types.
>
> If you're using a `With` block only to initialize the properties or fields of an object that you've just instantiated, consider using an object initializer instead.

Example

In the following example, each `With` block executes a series of statements on a single object.

```
Private Sub AddCustomer()
    Dim theCustomer As New Customer

    With theCustomer
        .Name = "Coho Vineyard"
        .URL = "http://www.cohovineyard.com/"
        .City = "Redmond"
    End With

    With theCustomer.Comments
        .Add("First comment.")
        .Add("Second comment.")
    End With
End Sub

Public Class Customer
    Public Property Name As String
    Public Property City As String
    Public Property URL As String

    Public Property Comments As New List(Of String)
End Class
```

Example

The following example nests `With…End With` statements. Within the nested `With` statement, the syntax refers to the inner object.

```
Dim theWindow As New EntryWindow

With theWindow
    With .InfoLabel
        .Content = "This is a message."
        .Foreground = Brushes.DarkSeaGreen
        .Background = Brushes.LightYellow
    End With

    .Title = "The Form Title"
    .Show()
End With
```

See also

- List<T>
- Nested Control Structures
- Object Initializers: Named and Anonymous Types
- Anonymous Types

Yield Statement (Visual Basic)

4/28/2019 • 4 minutes to read • Edit Online

Sends the next element of a collection to a `For Each...Next` statement.

Syntax

```
Yield expression
```

Parameters

TERM	DEFINITION
`expression`	Required. An expression that is implicitly convertible to the type of the iterator function or `Get` accessor that contains the `Yield` statement.

Remarks

The `Yield` statement returns one element of a collection at a time. The `Yield` statement is included in an iterator function or `Get` accessor, which perform custom iterations over a collection.

You consume an iterator function by using a For Each...Next Statement or a LINQ query. Each iteration of the `For Each` loop calls the iterator function. When a `Yield` statement is reached in the iterator function, `expression` is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

An implicit conversion must exist from the type of `expression` in the `Yield` statement to the return type of the iterator.

You can use an `Exit Function` or `Return` statement to end the iteration.

"Yield" is not a reserved word and has special meaning only when it is used in an `Iterator` function or `Get` accessor.

For more information about iterator functions and `Get` accessors, see Iterators.

Iterator Functions and Get Accessors

The declaration of an iterator function or `Get` accessor must meet the following requirements:

- It must include an Iterator modifier.
- The return type must be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.
- It cannot have any `ByRef` parameters.

An iterator function cannot occur in an event, instance constructor, static constructor, or static destructor.

An iterator function can be an anonymous function. For more information, see Iterators.

Exception Handling

A `Yield` statement can be inside a `Try` block of a Try...Catch...Finally Statement. A `Try` block that has a `Yield` statement can have `Catch` blocks, and can have a `Finally` block.

A `Yield` statement cannot be inside a `Catch` block or a `Finally` block.

If the `For Each` body (outside of the iterator function) throws an exception, a `Catch` block in the iterator function is not executed, but a `Finally` block in the iterator function is executed. A `Catch` block inside an iterator function catches only exceptions that occur inside the iterator function.

Technical Implementation

The following code returns an `IEnumerable (Of String)` from an iterator function and then iterates through the elements of the `IEnumerable (Of String)`.

```
Dim elements As IEnumerable(Of String) = MyIteratorFunction()
    …
For Each element As String In elements
Next
```

The call to `MyIteratorFunction` doesn't execute the body of the function. Instead the call returns an `IEnumerable(Of String)` into the `elements` variable.

On an iteration of the `For Each` loop, the MoveNext method is called for `elements`. This call executes the body of `MyIteratorFunction` until the next `Yield` statement is reached. The `Yield` statement returns an expression that determines not only the value of the `element` variable for consumption by the loop body but also the Current property of elements, which is an `IEnumerable (Of String)`.

On each subsequent iteration of the `For Each` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `Yield` statement. The `For Each` loop completes when the end of the iterator function or a `Return` or `Exit Function` statement is reached.

Example

The following example has a `Yield` statement that is inside a For...Next loop. Each iteration of the For Each statement body in `Main` creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `Yield` statement, which occurs during the next iteration of the `For…Next` loop.

The return type of the iterator method is IEnumerable<T>, an iterator interface type. When the iterator method is called, it returns an enumerable object that contains the powers of a number.

```
Sub Main()
    For Each number In Power(2, 8)
        Console.Write(number & " ")
    Next
    ' Output: 2 4 8 16 32 64 128 256
    Console.ReadKey()
End Sub

Private Iterator Function Power(
ByVal base As Integer, ByVal highExponent As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    Dim result = 1

    For counter = 1 To highExponent
        result = result * base
        Yield result
    Next
End Function
```

Example

The following example demonstrates a `Get` accessor that is an iterator. The property declaration includes an `Iterator` modifier.

```
Sub Main()
    Dim theGalaxies As New Galaxies
    For Each theGalaxy In theGalaxies.NextGalaxy
        With theGalaxy
            Console.WriteLine(.Name & "  " & .MegaLightYears)
        End With
    Next
    Console.ReadKey()
End Sub

Public Class Galaxies
    Public ReadOnly Iterator Property NextGalaxy _
    As System.Collections.Generic.IEnumerable(Of Galaxy)
        Get
            Yield New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400}
            Yield New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25}
            Yield New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0}
            Yield New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        End Get
    End Property
End Class

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class
```

For additional examples, see Iterators.

See also

- Statements

Clauses (Visual Basic)

5/4/2018 • 2 minutes to read • Edit Online

The topics in this section document Visual Basic run-time clauses.

In This Section

Alias

As

Handles

Implements

In

Into

Of

Related Sections

Visual Basic Language Reference

Visual Basic

Alias Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that an external procedure has another name in its DLL.

Remarks

The `Alias` keyword can be used in this context:

Declare Statement

In the following example, the `Alias` keyword is used to provide the name of the function in advapi32.dll, `GetUserNameA`, that `getUserName` is used in place of in this example. Function `getUserName` is called in sub `getUser`, which displays the name of the current user.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Sub GetUser()
    Dim buffer As String = New String(CChar(" "), 25)
    Dim retVal As Integer = GetUserName(buffer, 25)
    Dim userName As String = Strings.Left(buffer, InStr(buffer, Chr(0)) - 1)
    MsgBox(userName)
End Sub
```

See also

- Keywords

As Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Introduces an `As` clause, which identifies a data type in a declaration statement or a constraint list on a generic type parameter.

Remarks

The `As` keyword can be used in these contexts:

Aggregate Clause

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

For...Next Statements

For Each...Next Statements

From Clause

Function Statement

Group Join Clause

Interface Statement

Operator Statement

Property Statement

Structure Statement

Sub Statement

Try...Catch...Finally Statements

See also

- How to: Create a New Variable
- Data Types
- Variable Declaration
- Type List
- Generic Types in Visual Basic
- Keywords

Handles Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Declares that a procedure handles a specified event.

Syntax

```
proceduredeclaration Handles eventlist
```

Parts

`proceduredeclaration`
The `Sub` procedure declaration for the procedure that will handle the event.

`eventlist`
List of the events for `proceduredeclaration` to handle, separated by commas. The events must be raised by either the base class for the current class, or by an object declared using the `WithEvents` keyword.

Remarks

Use the `Handles` keyword at the end of a procedure declaration to cause it to handle events raised by an object variable declared using the `WithEvents` keyword. The `Handles` keyword can also be used in a derived class to handle events from a base class.

The `Handles` keyword and the `AddHandler` statement both allow you to specify that particular procedures handle particular events, but there are differences. Use the `Handles` keyword when defining a procedure to specify that it handles a particular event. The `AddHandler` statement connects procedures to events at run time. For more information, see AddHandler Statement.

For custom events, the application invokes the event's `AddHandler` accessor when it adds the procedure as an event handler. For more information on custom events, see Event Statement.

Example

```
Public Class ContainerClass
    ' Module or class level declaration.
    WithEvents Obj As New Class1

    Public Class Class1
        ' Declare an event.
        Public Event Ev_Event()
        Sub CauseSomeEvent()
            ' Raise an event.
            RaiseEvent Ev_Event()
        End Sub
    End Class

    Sub EventHandler() Handles Obj.Ev_Event
        ' Handle the event.
        MsgBox("EventHandler caught event.")
    End Sub

    ' Call the TestEvents procedure from an instance of the ContainerClass
    ' class to test the Ev_Event event and the event handler.
    Public Sub TestEvents()
        Obj.CauseSomeEvent()
    End Sub
End Class
```

The following example demonstrates how a derived class can use the `Handles` statement to handle an event from a base class.

```
Public Class BaseClass
    ' Declare an event.
    Event Ev1()
End Class
Class DerivedClass
    Inherits BaseClass
    Sub TestEvents() Handles MyBase.Ev1
        ' Add code to handle this event.
    End Sub
End Class
```

Example

The following example contains two button event handlers for a **WPF Application** project.

```
Private Sub Button1_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles
Button1.Click
    MessageBox.Show(sender.Name & " clicked")
End Sub

Private Sub Button2_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles
Button2.Click
    MessageBox.Show(sender.Name & " clicked")
End Sub
```

Example

The following example is equivalent to the previous example. The `eventlist` in the `Handles` clause contains the events for both buttons.

```
Private Sub Button_Click(sender As System.Object, e As System.Windows.RoutedEventArgs) Handles
Button1.Click, Button2.Click
    MessageBox.Show(sender.Name & " clicked")
End Sub
```

See also

- WithEvents
- AddHandler Statement
- RemoveHandler Statement
- Event Statement
- RaiseEvent Statement
- Events

Implements Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that a class or structure member is providing the implementation for a member defined in an interface.

Remarks

The `Implements` keyword is not the same as the Implements Statement. You use the `Implements` statement to specify that a class or structure implements one or more interfaces, and then for each member you use the `Implements` keyword to specify which interface and which member it implements.

If a class or structure implements an interface, it must include the `Implements` statement immediately after the Class Statement or Structure Statement, and it must implement all the members defined by the interface.

Reimplementation

In a derived class, you can reimplement an interface member that the base class has already implemented. This is different from overriding the base class member in the following respects:

- The base class member does not need to be Overridable to be reimplemented.
- You can reimplement the member with a different name.

The `Implements` keyword can be used in the following contexts:

- Event Statement
- Function Statement
- Property Statement
- Sub Statement

See also

- Implements Statement
- Interface Statement
- Class Statement
- Structure Statement

In Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the group that the loop variable is to traverse in a `For Each` loop, or specifies the collection to query in a `From`, `Join`, or `Group Join` clause.

Remarks

The `In` keyword can be used in the following contexts:

For Each...Next Statement

From Clause

Join Clause

Group Join Clause

See also

- Keywords

Into Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Identifies aggregate functions or groupings to apply to a collection.

Remarks

The `Each` keyword is used in the following contexts:

Aggregate Clause

Group By Clause

Group Join Clause

See also

- Keywords

Of Clause (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Introduces an `Of` clause, which identifies a *type parameter* on a *generic* class, structure, interface, delegate, or procedure. For information on generic types, see Generic Types in Visual Basic.

Using the Of Keyword

The following code example uses the `Of` keyword to define the outline of a class that takes two type parameters. It *constrains* the `keyType` parameter by the IComparable interface, which means the consuming code must supply a type argument that implements IComparable. This is necessary so that the `add` procedure can call the IComparable.CompareTo method. For more information on constraints, see Type List.

```
Public Class Dictionary(Of entryType, keyType As IComparable)
    Public Sub add(ByVal e As entryType, ByVal k As keyType)
        Dim dk As keyType
        If k.CompareTo(dk) = 0 Then
        End If
    End Sub
    Public Function find(ByVal k As keyType) As entryType
    End Function
End Class
```

If you complete the preceding class definition, you can construct a variety of `dictionary` classes from it. The types you supply to `entryType` and `keyType` determine what type of entry the class holds and what type of key it associates with each entry. Because of the constraint, you must supply to `keyType` a type that implements IComparable.

The following code example creates an object that holds `String` entries and associates an `Integer` key with each one. `Integer` implements IComparable and therefore satisfies the constraint on `keyType`.

```
Dim d As New dictionary(Of String, Integer)
```

The `Of` keyword can be used in these contexts:

Class Statement

Delegate Statement

Function Statement

Interface Statement

Structure Statement

Sub Statement

See also

- IComparable
- Type List
- Generic Types in Visual Basic

- In
- Out

Declaration Contexts and Default Access Levels (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

This topic describes which Visual Basic types can be declared within which other types, and what their access levels default to if not specified.

Declaration Context Levels

The *declaration context* of a programming element is the region of code in which it is declared. This is often another programming element, which is then called the *containing element*.

The levels for declaration contexts are the following:

- *Namespace level* — within a source file or namespace but not within a class, structure, module, or interface
- *Module level* — within a class, structure, module, or interface but not within a procedure or block
- *Procedure level* — within a procedure or block (such as `If` or `For`)

The following table shows the default access levels for various declared programming elements, depending on their declaration contexts.

DECLARED ELEMENT	NAMESPACE LEVEL	MODULE LEVEL	PROCEDURE LEVEL
Variable (Dim Statement)	Not allowed	`Private` (`Public` in `Structure`, not allowed in `Interface`)	`Public`
Constant (Const Statement)	Not allowed	`Private` (`Public` in `Structure`, not allowed in `Interface`)	`Public`
Enumeration (Enum Statement)	`Friend`	`Public`	Not allowed
Class (Class Statement)	`Friend`	`Public`	Not allowed
Structure (Structure Statement)	`Friend`	`Public`	Not allowed
Module (Module Statement)	`Friend`	Not allowed	Not allowed
Interface (Interface Statement)	`Friend`	`Public`	Not allowed
Procedure (Function Statement, Sub Statement)	Not allowed	`Public`	Not allowed

DECLARED ELEMENT	NAMESPACE LEVEL	MODULE LEVEL	PROCEDURE LEVEL
External reference (Declare Statement)	Not allowed	`Public` (not allowed in `Interface`)	Not allowed
Operator (Operator Statement)	Not allowed	`Public` (not allowed in `Interface` or `Module`)	Not allowed
Property (Property Statement)	Not allowed	`Public`	Not allowed
Default property (Default)	Not allowed	`Public` (not allowed in `Module`)	Not allowed
Event (Event Statement)	Not allowed	`Public`	Not allowed
Delegate (Delegate Statement)	`Friend`	`Public`	Not allowed

For more information, see Access levels in Visual Basic.

See also

- Friend
- Private
- Public

Attribute List (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Specifies the attributes to be applied to a declared programming element. Multiple attributes are separated by commas. Following is the syntax for one attribute.

Syntax

```
[ attributemodifier ] attributename [ ( attributearguments | attributeinitializer ) ]
```

Parts

`attributemodifier`	Required for attributes applied at the beginning of a source file. Can be Assembly or Module.
`attributename`	Required. Name of the attribute.
`attributearguments`	Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
`attributeinitializer`	Optional. List of variable or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

You can apply one or more attributes to nearly any programming element (types, procedures, properties, and so forth). Attributes appear in your assembly's metadata, and they can help you annotate your code or specify how to use a particular programming element. You can apply attributes defined by Visual Basic and the .NET Framework, and you can define your own attributes.

For more information on when to use attributes, see Attributes overview. For information on attribute names, see Declared Element Names.

Rules

- **Placement.** You can apply attributes to most declared programming elements. To apply one or more attributes, you place an *attribute block* at the beginning of the element declaration. Each entry in the attribute list specifies an attribute you wish to apply, and the modifier and arguments you are using for this invocation of the attribute.
- **Angle Brackets.** If you supply an attribute list, you must enclose it in angle brackets ("`<`" and "`>`").
- **Part of the Declaration.** The attribute must be part of the element declaration, not a separate statement. You can use the line-continuation sequence ("` _`") to extend the declaration statement onto multiple source-code lines.
- **Modifiers.** An attribute modifier (`Assembly` or `Module`) is required on every attribute applied to a programming element at the beginning of a source file. Attribute modifiers are not allowed on

attributes applied to elements that are not at the beginning of a source file.

- **Arguments.** All positional arguments for an attribute must precede any variable or property initializers.

Example

The following example applies the DllImportAttribute attribute to a skeleton definition of a `Function` procedure.

```
<DllImportAttribute("kernel32.dll", EntryPoint:="MoveFileW",
    SetLastError:=True, CharSet:=CharSet.Unicode,
    ExactSpelling:=True,
    CallingConvention:=CallingConvention.StdCall)>
Public Shared Function MoveFile(ByVal src As String,
  ByVal dst As String) As Boolean
    ' This function copies a file from the path src to the path dst.
    ' Leave this function empty. The DLLImport attribute forces calls
    ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

DllImportAttribute indicates that the attributed procedure represents an entry point in an unmanaged dynamic-link library (DLL). The attribute supplies the DLL name as a positional argument and the other information as variable initializers.

See also

- Assembly
- Module <keyword>
- Attributes overview
- How to: Break and Combine Statements in Code

Parameter List (Visual Basic)

4/28/2019 • 3 minutes to read • Edit Online

Specifies the parameters a procedure expects when it is called. Multiple parameters are separated by commas. The following is the syntax for one parameter.

Syntax

```
[ <attributelist> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ]
parametername[( )] [ As parametertype ] [ = defaultvalue ]
```

Parts

`attributelist`
Optional. List of attributes that apply to this parameter. You must enclose the Attribute List in angle brackets ("`<`" and "`>`").

`Optional`
Optional. Specifies that this parameter is not required when the procedure is called.

`ByVal`
Optional. Specifies that the procedure cannot replace or reassign the variable element underlying the corresponding argument in the calling code.

`ByRef`
Optional. Specifies that the procedure can modify the underlying variable element in the calling code the same way the calling code itself can.

`ParamArray`
Optional. Specifies that the last parameter in the parameter list is an optional array of elements of the specified data type. This lets the calling code pass an arbitrary number of arguments to the procedure.

`parametername`
Required. Name of the local variable representing the parameter.

`parametertype`
Required if `Option Strict` is `On`. Data type of the local variable representing the parameter.

`defaultvalue`
Required for `Optional` parameters. Any constant or constant expression that evaluates to the data type of the parameter. If the type is `Object`, or a class, interface, array, or structure, the default value can only be `Nothing`.

Remarks

Parameters are surrounded by parentheses and separated by commas. A parameter can be declared with any data type. If you do not specify `parametertype`, it defaults to `Object`.

When the calling code calls the procedure, it passes an *argument* to each required parameter. For more information, see Differences Between Parameters and Arguments.

The argument the calling code passes to each parameter is a pointer to an underlying element in the calling

code. If this element is *nonvariable* (a constant, literal, enumeration, or expression), it is impossible for any code to change it. If it is a *variable* element (a declared variable, field, property, array element, or structure element), the calling code can change it. For more information, see Differences Between Modifiable and Nonmodifiable Arguments.

If a variable element is passed `ByRef`, the procedure can change it as well. For more information, see Differences Between Passing an Argument By Value and By Reference.

Rules

- **Parentheses.** If you specify a parameter list, you must enclose it in parentheses. If there are no parameters, you can still use parentheses enclosing an empty list. This improves the readability of your code by clarifying that the element is a procedure.
- **Optional Parameters.** If you use the `Optional` modifier on a parameter, all subsequent parameters in the list must also be optional and be declared by using the `Optional` modifier.

  Every optional parameter declaration must supply the `defaultvalue` clause.

  For more information, see Optional Parameters.
- **Parameter Arrays.** You must specify `ByVal` for a `ParamArray` parameter.

  You cannot use both `Optional` and `ParamArray` in the same parameter list.

  For more information, see Parameter Arrays.
- **Passing Mechanism.** The default mechanism for every argument is `ByVal`, which means the procedure cannot change the underlying variable element. However, if the element is a reference type, the procedure can modify the contents or members of the underlying object, even though it cannot replace or reassign the object itself.
- **Parameter Names.** If the parameter's data type is an array, follow `parametername` immediately by parentheses. For more information on parameter names, see Declared Element Names.

Example

The following example shows a `Function` procedure that defines two parameters.

```
Public Function HowMany(ByVal ch As Char, ByVal st As String) As Integer
End Function
Dim howManyA As Integer = HowMany("a"c, "How many a's in this string?")
```

See also

- DllImportAttribute
- Function Statement
- Sub Statement
- Declare Statement
- Structure Statement
- Option Strict Statement
- Attributes overview
- How to: Break and Combine Statements in Code

Type List (Visual Basic)

4/26/2019 • 3 minutes to read • Edit Online

Specifies the *type parameters* for a *generic* programming element. Multiple parameters are separated by commas. Following is the syntax for one type parameter.

Syntax

```
[genericmodifier] typename [ As constraintlist ]
```

Parts

TERM	DEFINITION
`genericmodifier`	Optional. Can be used only in generic interfaces and delegates. You can declare a type covariant by using the Out keyword or contravariant by using the In keyword. See Covariance and Contravariance.
`typename`	Required. Name of the type parameter. This is a placeholder, to be replaced by a defined type supplied by the corresponding type argument.
`constraintlist`	Optional. List of requirements that constrain the data type that can be supplied for `typename`. If you have multiple constraints, enclose them in curly braces (`{ }`) and separate them with commas. You must introduce the constraint list with the As keyword. You use `As` only once, at the beginning of the list.

Remarks

Every generic programming element must take at least one type parameter. A type parameter is a placeholder for a specific type (a *constructed element*) that client code specifies when it creates an instance of the generic type. You can define a generic class, structure, interface, procedure, or delegate.

For more information on when to define a generic type, see Generic Types in Visual Basic. For more information on type parameter names, see Declared Element Names.

Rules

- **Parentheses.** If you supply a type parameter list, you must enclose it in parentheses, and you must introduce the list with the Of keyword. You use `Of` only once, at the beginning of the list.
- **Constraints.** A list of *constraints* on a type parameter can include the following items in any combination:
  - Any number of interfaces. The supplied type must implement every interface in this list.
  - At most one class. The supplied type must inherit from that class.

- The `New` keyword. The supplied type must expose a parameterless constructor that your generic type can access. This is useful if you constrain a type parameter by one or more interfaces. A type that implements interfaces does not necessarily expose a constructor, and depending on the access level of a constructor, the code within the generic type might not be able to access it.
- Either the `Class` keyword or the `Structure` keyword. The `Class` keyword constrains a generic type parameter to require that any type argument passed to it be a reference type, for example a string, array, or delegate, or an object created from a class. The `Structure` keyword constrains a generic type parameter to require that any type argument passed to it be a value type, for example a structure, enumeration, or elementary data type. You cannot include both `Class` and `Structure` in the same `constraintlist`.

The supplied type must satisfy every requirement you include in `constraintlist`.

Constraints on each type parameter are independent of constraints on other type parameters.

Behavior

- **Compile-Time Substitution.** When you create a constructed type from a generic programming element, you supply a defined type for each type parameter. The Visual Basic compiler substitutes that supplied type for every occurrence of `typename` within the generic element.
- **Absence of Constraints.** If you do not specify any constraints on a type parameter, your code is limited to the operations and members supported by the Object Data Type for that type parameter.

Example

The following example shows a skeleton definition of a generic dictionary class, including a skeleton function to add a new entry to the dictionary.

```
Public Class dictionary(Of entryType, keyType As {IComparable, IFormattable, New})
    Public Sub add(ByVal et As entryType, ByVal kt As keyType)
        Dim dk As keyType
        If kt.CompareTo(dk) = 0 Then
        End If
    End Sub
End Class
```

Example

Because `dictionary` is generic, the code that uses it can create a variety of objects from it, each having the same functionality but acting on a different data type. The following example shows a line of code that creates a `dictionary` object with `String` entries and `Integer` keys.

```
Dim dictInt As New dictionary(Of String, Integer)
```

Example

The following example shows the equivalent skeleton definition generated by the preceding example.

```
Public Class dictionary
    Public Sub Add(ByVal et As String, ByVal kt As Integer)
        Dim dk As Integer
        If kt.CompareTo(dk) = 0 Then
        End If
    End Sub
End Class
```

See also

- Of
- New Operator
- Access levels in Visual Basic
- Object Data Type
- Function Statement
- Structure Statement
- Sub Statement
- How to: Use a Generic Class
- Covariance and Contravariance
- In
- Out

Recommended XML Tags for Documentation Comments (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler can process documentation comments in your code to an XML file. You can use additional tools to process the XML file into documentation.

XML comments are allowed on code constructs such as types and type members. For partial types, only one part of the type can have XML comments, although there is no restriction on commenting its members.

> **NOTE**
> Documentation comments cannot be applied to namespaces. The reason is that one namespace can span several assemblies, and not all assemblies have to be loaded at the same time.

The compiler processes any tag that is valid XML. The following tags provide commonly used functionality in user documentation.

<c>	<code>	<example>
<exception> [1]	<include> [1]	<list>
<para>	<param> [1]	<paramref>
<permission> [1]	<remarks>	<returns>
<see> [1]	<seealso> [1]	<summary>
<typeparam> [1]	<value>	

([1] The compiler verifies syntax.)

> **NOTE**
> If you want angle brackets to appear in the text of a documentation comment, use `<` and `>`. For example, the string `"<text in angle brackets>"` will appear as `<text in angle brackets>`.

See also

- Documenting Your Code with XML
- /doc
- How to: Create XML Documentation

<c> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that text within a description is code.

Syntax

```
<c>text</c>
```

Parameters

PARAMETER	DESCRIPTION
`text`	The text you would like to indicate as code.

Remarks

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<c>` tag in the summary section to indicate that `Counter` is code.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- XML Comment Tags

<code> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates that the text is multiple lines of code.

Syntax

```
<code>content</code>
```

Parameters

`content`
The text to mark as code.

Remarks

Use the `<code>` tag to indicate multiple lines as code. Use `<c>` to indicate that text within a description should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

This example uses the <code> tag to include example code for using the `ID` field.

```
Public Class Employee
    ''' <remarks>
    ''' <example> This sample shows how to set the <c>ID</c> field.
    ''' <code>
    ''' Dim alice As New Employee
    ''' alice.ID = 1234
    ''' </code>
    ''' </example>
    ''' </remarks>
    Public ID As Integer
End Class
```

See also

- XML Comment Tags

<example> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies an example for the member.

Syntax

```
<example>description</example>
```

Parameters

`description`
A description of the code sample.

Remarks

The `<example>` tag lets you specify an example of how to use a method or other library member. This commonly involves using the <code> tag.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<example>` tag to include an example for using the `ID` field.

```
Public Class Employee
    ''' <remarks>
    ''' <example> This sample shows how to set the <c>ID</c> field.
    ''' <code>
    ''' Dim alice As New Employee
    ''' alice.ID = 1234
    ''' </code>
    ''' </example>
    ''' </remarks>
    Public ID As Integer
End Class
```

See also

- XML Comment Tags

<exception> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies which exceptions can be thrown.

Syntax

```
<exception cref="member">description</exception>
```

Parameters

`member`
A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks (" ").

`description`
A description.

Remarks

Use the `<exception>` tag to specify which exceptions can be thrown. This tag is applied to a method definition.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<exception>` tag to describe an exception that the `IntDivide` function can throw.

```
''' <exception cref="System.OverflowException">
''' Thrown when <paramref name="denominator"/><c> = 0</c>.
''' </exception>
Public Function IntDivide(
        ByVal numerator As Integer,
        ByVal denominator As Integer
) As Integer
    Return numerator \ denominator
End Function
```

See also

- XML Comment Tags

<include> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Refers to another file that describes the types and members in your source code.

Syntax

```
<include file="filename" path="tagpath[@name='id']" />
```

Parameters

`filename`
Required. The name of the file containing the documentation. The file name can be qualified with a path. Enclose `filename` in double quotation marks (" ").

`tagpath`
Required. The path of the tags in `filename` that leads to the tag `name`. Enclose the path in double quotation marks (" ").

`name`
Required. The name specifier in the tag that precedes the comments. `Name` will have an `id`.

`id`
Required. The ID for the tag that precedes the comments. Enclose the ID in single quotation marks (' ').

Remarks

Use the `<include>` tag to refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file.

The `<include>` tag uses the W3C XML Path Language (XPath) Version 1.0 Recommendation. For more information about ways to customize your `<include>` use, see https://www.w3.org/TR/xpath.

Example

This example uses the `<include>` tag to import member documentation comments from a file called `commentFile.xml`.

```
''' <include file="commentFile.xml"
''' path="Docs/Members[@name='Open']/*" />
Public Sub Open(ByVal filename As String)
    ' Code goes here.
End Sub
''' <include file="commentFile.xml"
''' path="Docs/Members[@name='Close']/*" />
Public Sub Close(ByVal filename As String)
    ' Code goes here.
End Sub
```

The format of the `commentFile.xml` is as follows.

```
<Docs>
<Members name="Open">
<summary>Opens a file.</summary>
<param name="filename">File name to open.</param>
</Members>
<Members name="Close">
<summary>Closes a file.</summary>
<param name="filename">File name to close.</param>
</Members>
</Docs>
```

See also

- XML Comment Tags

<list> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Defines a list or table.

Syntax

```
<list type="type">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

Parameters

`type`
The type of the list. Must be a "bullet" for a bulleted list, "number" for a numbered list, or "table" for a two-column table.

`term`
Only used when `type` is "table." A term to define, which is defined in the description tag.

`description`
When `type` is "bullet" or "number," `description` is an item in the list When `type` is "table," `description` is the definition of `term`.

Remarks

The `<listheader>` block defines the heading of either a table or definition list. When defining a table, you only have to supply an entry for `term` in the heading.

Each item in the list is specified with an `<item>` block. When creating a definition list, you must specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only have to supply an entry for `description`.

A list or table can have as many `<item>` blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<list>` tag to define a bulleted list in the remarks section.

```
''' <remarks>Before calling the <c>Reset</c> method, be sure to:
''' <list type="bullet">
''' <item><description>Close all connections.</description></item>
''' <item><description>Save the object state.</description></item>
''' </list>
''' </remarks>
Public Sub Reset()
    ' Code goes here.
End Sub
```

See also

- XML Comment Tags

<para> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies that the content is formatted as a paragraph.

Syntax

```
<para>content</para>
```

Parameters

`content`
The text of the paragraph.

Remarks

The `<para>` tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<para>` tag to split the remarks section for the `UpdateRecord` method into two paragraphs.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<param> (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Defines a parameter name and description.

Syntax

```
<param name="name">description</param>
```

Parameters

`name`
The name of a method parameter. Enclose the name in double quotation marks (" ").

`description`
A description for the parameter.

Remarks

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method.

The text for the `<param>` tag will appear in the following locations:

- Parameter Info of IntelliSense. For more information, see Using IntelliSense.
- Object Browser. For more information, see Viewing the Structure of Code.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<param>` tag to describe the `id` parameter.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<paramref> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Formats a word as a parameter.

Syntax

```
<paramref name="name"/>
```

Parameters

`name`

The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks

The `<paramref>` tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<paramref>` tag to refer to the `id` parameter.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<permission> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies a required permission for the member.

Syntax

```
<permission cref="member">description</permission>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates `member` to the canonical element name in the output XML. Enclose `member` in quotation marks (" ").

`description`
A description of the access to the member.

Remarks

Use the `<permission>` tag to document the access of a member. Use the PermissionSet class to specify access to a member.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<permission>` tag to describe that the FileIOPermission is required by the `ReadFile` method.

```
''' <permission cref="System.Security.Permissions.FileIOPermission">
''' Needs full access to the specified file.
''' </permission>
Public Sub ReadFile(ByVal filename As String)
    ' Code goes here.
End Sub
```

See also

- XML Comment Tags

<remarks> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies a remarks section for the member.

Syntax

```
<remarks>description</remarks>
```

Parameters

`description`
A description of the member.

Remarks

Use the `<remarks>` tag to add information about a type, supplementing the information specified with <summary>.

This information appears in the Object Browser. For information about the Object Browser, see Viewing the Structure of Code.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<remarks>` tag to explain what the `UpdateRecord` method does.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<returns> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the return value of the property or function.

Syntax

```
<returns>description</returns>
```

Parameters

`description`
A description of the return value.

Remarks

Use the `<returns>` tag in the comment for a method declaration to describe the return value.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<returns>` tag to explain what the `DoesRecordExist` function returns.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<see> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies a link to another member.

Syntax

```
<see cref="member"/>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks (" ").

Remarks

Use the `<see>` tag to specify a link from within text. Use <seealso> to indicate text that you might want to appear in a "See Also" section.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<see>` tag in the `UpdateRecord` remarks section to refer to the `DoesRecordExist` method.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<seealso> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies a link that appears in the See Also section.

Syntax

```
<seealso cref="member"/>
```

Parameters

`member`
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks (" ").

Remarks

Use the `<seealso>` tag to specify the text that you want to appear in a See Also section. Use <see> to specify a link from within text.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<seealso>` tag in the `DoesRecordExist` remarks section to refer to the `UpdateRecord` method.

```
''' <param name="id">The ID of the record to update.</param>
''' <remarks>Updates the record <paramref name="id"/>.
''' <para>Use <see cref="DoesRecordExist"/> to verify that
''' the record exists before calling this method.</para>
''' </remarks>
Public Sub UpdateRecord(ByVal id As Integer)
    ' Code goes here.
End Sub
''' <param name="id">The ID of the record to check.</param>
''' <returns><c>True</c> if <paramref name="id"/> exists,
''' <c>False</c> otherwise.</returns>
''' <remarks><seealso cref="UpdateRecord"/></remarks>
Public Function DoesRecordExist(ByVal id As Integer) As Boolean
    ' Code goes here.
End Function
```

See also

- XML Comment Tags

<summary> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the summary of the member.

Syntax

```
<summary>description</summary>
```

Parameters

`description`
A summary of the object.

Remarks

Use the `<summary>` tag to describe a type or a type member. Use <remarks> to add supplemental information to a type description.

The text for the `<summary>` tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser. For information about the Object Browser, see Viewing the Structure of Code.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<summary>` tag to describe the `ResetCounter` method and `Counter` property.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- XML Comment Tags

<typeparam> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Defines a type parameter name and description.

Syntax

```
<typeparam name="name">description</typeparam>
```

Parameters

`name`
The name of the type parameter. Enclose the name in double quotation marks (" ").

`description`
A description of the type parameter.

Remarks

Use the `<typeparam>` tag in the comment for a generic type or generic member declaration to describe one of the type parameters.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<typeparam>` tag to describe the `id` parameter.

```
''' <typeparam name="T">
''' The base item type. Must implement IComparable.
''' </typeparam>
Public Class itemManager(Of T As IComparable)
    ' Insert code that defines class members.
End Class
```

See also

- XML Comment Tags

<value> (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the description of a property.

Syntax

```
<value>property-description</value>
```

Parameters

`property-description`
A description for the property.

Remarks

Use the `<value>` tag to describe a property. Note that when you add a property using the code wizard in the Visual Studio development environment, it will add a <summary> tag for the new property. You should then manually add a `<value>` tag to describe the value that the property represents.

Compile with /doc to process documentation comments to a file.

Example

This example uses the `<value>` tag to describe what value the `Counter` property holds.

```
''' <summary>
''' Resets the value the <c>Counter</c> field.
''' </summary>
Public Sub ResetCounter()
    counterValue = 0
End Sub
Private counterValue As Integer = 0
''' <summary>
''' Returns the number of times Counter was called.
''' </summary>
''' <value>Number of times Counter was called.</value>
Public ReadOnly Property Counter() As Integer
    Get
        counterValue += 1
        Return counterValue
    End Get
End Property
```

See also

- XML Comment Tags

XML Axis Properties (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The topics in this section document the syntax of XML axis properties in Visual Basic. The XML axis properties make it easy to access XML directly in your code.

In This Section

TOPIC	DESCRIPTION
XML Attribute Axis Property	Describes how to access the attributes of an XElement object.
XML Child Axis Property	Describes how to access the children of an XElement object.
XML Descendant Axis Property	Describes how to access the descendants of an XElement object.
Extension Indexer Property	Describes how to access individual elements in a collection of XElement or XAttribute objects.
XML Value Property	Describes how to access the value of the first element of a collection of XElement or XAttribute objects.

See also

- XML

XML Attribute Axis Property (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Provides access to the value of an attribute for an XElement object or to the first element in a collection of XElement objects.

Syntax

```
      object.@attribute
-or-
object.@<attribute>
```

Parts

`object`
Required. An XElement object or a collection of XElement objects.

.@
Required. Denotes the start of an attribute axis property.

<
Optional. Denotes the beginning of the name of the attribute when `attribute` is not a valid identifier in Visual Basic.

`attribute`
Required. Name of the attribute to access, of the form [`prefix` :] `name` .

PART	DESCRIPTION
`prefix`	Optional. XML namespace prefix for the attribute. Must be a global XML namespace defined with an `Imports` statement.
`name`	Required. Local attribute name. See Names of Declared XML Elements and Attributes.

>
Optional. Denotes the end of the name of the attribute when `attribute` is not a valid identifier in Visual Basic.

Return Value

A string that contains the value of `attribute` . If the attribute name does not exist, `Nothing` is returned.

Remarks

You can use an XML attribute axis property to access the value of an attribute by name from an XElement object or from the first element in a collection of XElement objects. You can retrieve an attribute value by name, or add a new attribute to an element by specifying a new name preceded by the @ identifier.

When you refer to an XML attribute using the @ identifier, the attribute value is returned as a string and you do not need to explicitly specify the Value property.

The naming rules for XML attributes differ from the naming rules for Visual Basic identifiers. To access an XML attribute that has a name that is not a valid Visual Basic identifier, enclose the name in angle brackets (< and >).

XML Namespaces

The name in an attribute axis property can use only XML namespace prefixes declared globally by using the `Imports` statement. It cannot use XML namespace prefixes declared locally within XML element literals. For more information, see Imports Statement (XML Namespace).

Example

The following example shows how to get the values of the XML attributes named `type` from a collection of XML elements that are named `phone`.

```
' Topic: XML Attribute Axis Property
Dim phones As XElement =
    <phones>
        <phone type="home">206-555-0144</phone>
        <phone type="work">425-555-0145</phone>
    </phones>

Dim phoneTypes As XElement =
  <phoneTypes>
      <%= From phone In phones.<phone>
          Select <type><%= phone.@type %></type>
      %>
  </phoneTypes>

Console.WriteLine(phoneTypes)
```

This code displays the following text:

```
<phoneTypes>
<type>home</type>
<type>work</type>
</phoneTypes>
```

Example

The following example shows how to create attributes for an XML element both declaratively, as part of the XML, and dynamically by adding an attribute to an instance of an XElement object. The `type` attribute is created declaratively and the `owner` attribute is created dynamically.

```
Dim phone2 As XElement = <phone type="home">206-555-0144</phone>
phone2.@owner = "Harris, Phyllis"

Console.WriteLine(phone2)
```

This code displays the following text:

```
<phone type="home" owner="Harris, Phyllis">206-555-0144</phone>
```

Example

The following example uses the angle bracket syntax to get the value of the XML attribute named `number-type`, which is not a valid identifier in Visual Basic.

```
Dim phone As XElement =
     <phone number-type=" work">425-555-0145</phone>

 Console.WriteLine("Phone type: " & phone.@<number-type>)
```

This code displays the following text:

```
Phone type: work
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name "`ns:name`".

```
Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass3

    Shared Sub TestPrefix()
        Dim phone =
            <ns:phone ns:type="home">206-555-0144</ns:phone>

        Console.WriteLine("Phone type: " & phone.@ns:type)
    End Sub

End Class
```

This code displays the following text:

```
Phone type: home
```

See also

- XElement
- XML Axis Properties
- XML Literals
- Creating XML in Visual Basic
- Names of Declared XML Elements and Attributes

XML Child Axis Property (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Provides access to the children of one of the following: an XElement object, an XDocument object, a collection of XElement objects, or a collection of XDocument objects.

Syntax

```
object.<child>
```

Parts

TERM	DEFINITION
`object`	Required. An XElement object, an XDocument object, a collection of XElement objects, or a collection of XDocument objects.
.<	Required. Denotes the start of a child axis property.
`child`	Required. Name of the child nodes to access, of the form [`prefix:]name` . - `Prefix` - Optional. XML namespace prefix for the child node. Must be a global XML namespace defined with an `Imports` statement. - `Name` - Required. Local child node name. See Names of Declared XML Elements and Attributes.
>	Required. Denotes the end of a child axis property.

Return Value

A collection of XElement objects.

Remarks

You can use an XML child axis property to access child nodes by name from an XElement or XDocument object, or from a collection of XElement or XDocument objects. Use the XML `Value` property to access the value of the first child node in the returned collection. For more information, see XML Value Property.

The Visual Basic compiler converts child axis properties to calls to the Elements method.

XML Namespaces

The name in a child axis property can use only XML namespace prefixes declared globally with the `Imports` statement. It cannot use XML namespace prefixes declared locally within XML element literals. For more information, see Imports Statement (XML Namespace).

Example

The following example shows how to access the child nodes named `phone` from the `contact` object.

```
Dim contact As XElement =
    <contact>
        <name>Patrick Hines</name>
        <phone type="home">206-555-0144</phone>
        <phone type="work">425-555-0145</phone>
    </contact>

Dim homePhone = From hp In contact.<phone>
                Where contact.<phone>.@type = "home"
                Select hp

Console.WriteLine("Home Phone = {0}", homePhone(0).Value)
```

This code displays the following text:

```
Home Phone = 206-555-0144
```

Example

The following example shows how to access the child nodes named `phone` from the collection returned by the `contact` child axis property of the `contacts` object.

```
Dim contacts As XElement =
    <contacts>
        <contact>
            <name>Patrick Hines</name>
            <phone type="home">206-555-0144</phone>
        </contact>
        <contact>
            <name>Lance Tucker</name>
            <phone type="work">425-555-0145</phone>
        </contact>
    </contacts>

Dim homePhone = From contact In contacts.<contact>
                Where contact.<phone>.@type = "home"
                Select contact.<phone>

Console.WriteLine("Home Phone = {0}", homePhone(0).Value)
```

This code displays the following text:

```
Home Phone = 206-555-0144
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the first child node with the qualified name `ns:name`.

```
Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass4

    Shared Sub TestPrefix()
        Dim contact = <ns:contact>
                        <ns:name>Patrick Hines</ns:name>
                      </ns:contact>
        Console.WriteLine(contact.<ns:name>.Value)
    End Sub

End Class
```

This code displays the following text:

```
Patrick Hines
```

See also

- XElement
- XML Axis Properties
- XML Literals
- Creating XML in Visual Basic
- Names of Declared XML Elements and Attributes

XML Descendant Axis Property (Visual Basic)

3/5/2019 • 2 minutes to read • Edit Online

Provides access to the descendants of the following: an XElement object, an XDocument object, a collection of XElement objects, or a collection of XDocument objects.

Syntax

```
object...<descendant>
```

Parts

`object` Required. An XElement object, an XDocument object, a collection of XElement objects, or a collection of XDocument objects.

`...<` Required. Denotes the start of a descendant axis property.

`descendant` Required. Name of the descendant nodes to access, of the form [`prefix:`]`name` .

PART	DESCRIPTION
`prefix`	Optional. XML namespace prefix for the descendant node. Must be a global XML namespace that is defined by using an `Imports` statement.
`name`	Required. Local name of the descendant node. See Names of Declared XML Elements and Attributes.

`>` Required. Denotes the end of a descendant axis property.

Return Value

A collection of XElement objects.

Remarks

You can use an XML descendant axis property to access descendant nodes by name from an XElement or XDocument object, or from a collection of XElement or XDocument objects. Use the XML `Value` property to access the value of the first descendant node in the returned collection. For more information, see XML Value Property.

The Visual Basic compiler converts descendant axis properties into calls to the Descendants method.

XML Namespaces

The name in a descendant axis property can use only XML namespaces declared globally with the `Imports` statement. It cannot use XML namespaces declared locally within XML element literals. For more information, see Imports Statement (XML Namespace).

Example

The following example shows how to access the value of the first descendant node named `name` and the values of all descendant nodes named `phone` from the `contacts` object.

```
Dim contacts As XElement =
    <contacts>
        <contact>
            <name>Patrick Hines</name>
            <phone type="home">206-555-0144</phone>
            <phone type="work">425-555-0145</phone>
        </contact>
    </contacts>

Console.WriteLine("Name: " & contacts...<name>.Value)

Dim homePhone = From phone In contacts...<phone>
                Select phone.Value

Console.WriteLine("Home Phone = {0}", homePhone(0))
```

This code displays the following text:

```
Name: Patrick Hines
```

```
Home Phone = 206-555-0144
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and access the value of the first child node with the qualified name `ns:name`.

```
Imports <xmlns:ns = "http://SomeNamespace">

Class TestClass2

    Shared Sub TestPrefix()
        Dim contacts =
            <ns:contacts>
                <ns:contact>
                    <ns:name>Patrick Hines</ns:name>
                </ns:contact>
            </ns:contacts>

        Console.WriteLine("Name: " & contacts...<ns:name>.Value)
    End Sub

End Class
```

This code displays the following text:

```
Name: Patrick Hines
```

See also

- XElement
- XML Axis Properties
- XML Literals
- Creating XML in Visual Basic

- Names of Declared XML Elements and Attributes

Extension Indexer Property (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Provides access to individual elements in a collection.

Syntax

```
object(index)
```

Parts

TERM	DEFINITION
`object`	Required. A queryable collection. That is, a collection that implements IEnumerable<T> or IQueryable<T>.
(	Required. Denotes the start of the indexer property.
`index`	Required. An integer expression that specifies the zero-based position of an element of the collection.
)	Required. Denotes the end of the indexer property.

Return Value

The object from the specified location in the collection, or `Nothing` if the index is out of range.

Remarks

You can use the extension indexer property to access individual elements in a collection. This indexer property is typically used on the output of XML axis properties. The XML child and XML descendent axis properties return collections of XElement objects or an attribute value.

The Visual Basic compiler converts extension indexer properties to calls to the `ElementAtOrDefault` method. Unlike an array indexer, the `ElementAtOrDefault` method returns `Nothing` if the index is out of range. This behavior is useful when you cannot easily determine the number of elements in a collection.

This indexer property is like an extension property for collections that implement IEnumerable<T> or IQueryable<T>: it is used only if the collection does not have an indexer or a default property.

To access the value of the first element in a collection of XElement or XAttribute objects, you can use the XML `Value` property. For more information, see XML Value Property.

Example

The following example shows how to use the extension indexer to access the second child node in a collection of XElement objects. The collection is accessed by using the child axis property, which gets all child elements named `phone` in the `contact` object.

```
Dim contact As XElement =
    <contact>
        <name>Patrick Hines</name>
        <phone type="home">206-555-0144</phone>
        <phone type="work">425-555-0145</phone>
    </contact>

Console.WriteLine("Second phone number: " & contact.<phone>(1).Value)
```

This code displays the following text:

```
Second phone number: 425-555-0145
```

See also

- XElement
- XML Axis Properties
- XML Literals
- Creating XML in Visual Basic
- XML Value Property

XML Value Property (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Provides access to the value of the first element of a collection of XElement objects.

Syntax

```
object.Value
```

Parts

TERM	DEFINITION
`object`	Required. Collection of XElement objects.

Return Value

A `String` that contains the value of the first element of the collection, or `Nothing` if the collection is empty.

Remarks

The Value property makes it easy to access the value of the first element in a collection of XElement objects. This property first checks whether the collection contains at least one object. If the collection is empty, this property returns `Nothing`. Otherwise, this property returns the value of the Value property of the first element in the collection.

> **NOTE**
> When you access the value of an XML attribute using the '@' identifier, the attribute value is returned as a `String` and you do not need to explicitly specify the Value property.

To access other elements in a collection, you can use the XML extension indexer property. For more information, see Extension Indexer Property.

Inheritance

Most users will not have to implement IEnumerable<T>, and can therefore ignore this section.

The Value property is an extension property for types that implement `IEnumerable(Of XElement)`. The binding of this extension property is like the binding of extension methods: if a type implements one of the interfaces and defines a property that has the name "Value", that property has precedence over the extension property. In other words, this Value property can be overridden by defining a new property in a class that implements `IEnumerable(Of XElement)`.

Example

The following example shows how to use the Value property to access the first node in a collection of XElement

objects. The example uses the child axis property to get the collection of all child nodes named `phone` that are in the `contact` object.

```
Dim contact As XElement =
    <contact>
        <name>Patrick Hines</name>
        <phone type="home">206-555-0144</phone>
        <phone type="work">425-555-0145</phone>
    </contact>

Console.WriteLine("Phone number: " & contact.<phone>.Value)
```

This code displays the following text:

```
Phone number: 206-555-0144
```

Example

The following example shows how to get the value of an XML attribute from a collection of XAttribute objects. The example uses the attribute axis property to display the value of the `type` attribute for all of the `phone` elements.

```
Dim contact As XElement =
    <contact>
      <name>Patrick Hines</name>
      <phone type="home">206-555-0144</phone>
      <phone type="work">425-555-0145</phone>
    </contact>


Dim types = contact.<phone>.Attributes("type")

For Each attr In types
  Console.WriteLine(attr.Value)
Next
```

This code displays the following text:

```
home
```

```
work
```

See also

- XElement
- IEnumerable<T>
- XML Axis Properties
- XML Literals
- Creating XML in Visual Basic
- Extension Methods
- Extension Indexer Property
- XML Child Axis Property
- XML Attribute Axis Property

XML Literals (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The topics in this section document the syntax of XML literals in Visual Basic. The XML literal syntax enables you to incorporate XML directly in your code.

In This Section

TOPIC	DESCRIPTION
XML Element Literal	Describes the syntax for literals that represent XElement objects.
XML Document Literal	Describes the syntax for literals that represent XDocument objects.
XML CDATA Literal	Describes the syntax for literals that represent XCData objects.
XML Comment Literal	Describes the syntax for literals that represent XComment objects.
XML Processing Instruction Literal	Describes the syntax for literals that represent XProcessingInstruction objects.

See also

- XML

XML Element Literal (Visual Basic)

4/26/2019 • 4 minutes to read • Edit Online

A literal that represents an XElement object.

Syntax

```
<name [ attributeList ] />
-or-
<name [ attributeList ] > [ elementContents ] </[ name ]>
```

Parts

- `<`

  Required. Opens the starting element tag.

- `name`

  Required. Name of the element. The format is one of the following:

  - Literal text for the element name, of the form `[ePrefix:]eName`, where:

PART	DESCRIPTION
`ePrefix`	Optional. XML namespace prefix for the element. Must be a global XML namespace that is defined with an `Imports` statement in the file or at the project level, or a local XML namespace that is defined in this element or a parent element.
`eName`	Required. Name of the element. The format is one of the following: - Literal text. See Names of Declared XML Elements and Attributes. - Embedded expression of the form `<%= eNameExp %>`. The type of `eNameExp` must be `String` or a type that is implicitly convertible to XName.

  - Embedded expression of the form `<%= nameExp %>`. The type of `nameExp` must be `String` or a type implicitly convertible to XName. An embedded expression is not allowed in a closing tag of an element.

- `attributeList`

  Optional. List of attributes declared in the literal.

  `attribute [ attribute ... ]`

  Each `attribute` has one of the following syntaxes:

  - Attribute assignment, of the form `[aPrefix:]aName=aValue`, where:

PART	DESCRIPTION
`aPrefix`	Optional. XML namespace prefix for the attribute. Must be a global XML namespace that is defined with an `Imports` statement, or a local XML namespace that is defined in this element or a parent element.
`aName`	Required. Name of the attribute. The format is one of the following: - Literal text. See Names of Declared XML Elements and Attributes. - Embedded expression of the form `<%= aNameExp %>`. The type of `aNameExp` must be `String` or a type that is implicitly convertible to XName.
`aValue`	Optional. Value of the attribute. The format is one of the following: - Literal text, enclosed in quotation marks. - Embedded expression of the form `<%= aValueExp %>`. Any type is allowed.

  - Embedded expression of the form `<%= aExp %>`.

- `/>`

  Optional. Indicates that the element is an empty element, without content.

- `>`

  Required. Ends the beginning or empty element tag.

- `elementContents`

  Optional. Content of the element.

  `content [ content ... ]`

  Each `content` can be one of the following:

  - Literal text. All the white space in `elementContents` becomes significant if there is any literal text.
  - Embedded expression of the form `<%= contentExp %>`.
  - XML element literal.
  - XML comment literal. See XML Comment Literal.
  - XML processing instruction literal. See XML Processing Instruction Literal.
  - XML CDATA literal. See XML CDATA Literal.

- `</[name]>`

  Optional. Represents the closing tag for the element. The optional `name` parameter is not allowed when it is the result of an embedded expression.

Return Value

An XElement object.

Remarks

You can use the XML element literal syntax to create XElement objects in your code.

> **NOTE**
> An XML literal can span multiple lines without using line continuation characters. This feature enables you to copy content from an XML document and paste it directly into a Visual Basic program.

Embedded expressions of the form `<%= exp %>` enable you to add dynamic information to an XML element literal. For more information, see Embedded Expressions in XML.

The Visual Basic compiler converts the XML element literal into calls to the XElement constructor and, if it is required, the XAttribute constructor.

XML Namespaces

XML namespace prefixes are useful when you have to create XML literals with elements from the same namespace many times in code. You can use global XML namespace prefixes, which you define by using the `Imports` statement, or local prefixes, which you define by using the `xmlns:xmlPrefix="xmlNamespace"` attribute syntax. For more information, see Imports Statement (XML Namespace).

In accordance with the scoping rules for XML namespaces, local prefixes take precedence over global prefixes. However, if an XML literal defines an XML namespace, that namespace is not available to expressions that appear in an embedded expression. The embedded expression can access only the global XML namespace.

The Visual Basic compiler converts each global XML namespace that is used by an XML literal into a one local namespace definition in the generated code. Global XML namespaces that are not used do not appear in the generated code.

Example

The following example shows how to create a simple XML element that has two nested empty elements.

```
Dim test1 As XElement =
<outer>
    <inner1></inner1>
    <inner2/>
</outer>

Console.WriteLine(test1)
```

The example displays the following text. Notice that the literal preserves the structure of the empty elements.

```
<outer>
  <inner1></inner1>
  <inner2 />
</outer>
```

Example

The following example shows how to use embedded expressions to name an element and create attributes.

```
Dim elementType = "book"
Dim authorName = "My Author"
Dim attributeName1 = "year"
Dim attributeValue1 = 1999
Dim attributeName2 = "title"
Dim attributeValue2 = "My Book"

Dim book As XElement =
<<%= elementType %>
    isbn="1234"
    author=<%= authorName %>
    <%= attributeName1 %>=<%= attributeValue1 %>
    <%= New XAttribute(attributeName2, attributeValue2) %>
/>

Console.WriteLine(book)
```

This code displays the following text:

```
<book isbn="1234" author="My Author" year="1999" title="My Book" />
```

Example

The following example declares `ns` as an XML namespace prefix. It then uses the prefix of the namespace to create an XML literal and displays the element's final form.

```
' Place Imports statements at the top of your program.
Imports <xmlns:ns="http://SomeNamespace">

Class TestClass1

    Shared Sub TestPrefix()
        ' Create test using a global XML namespace prefix.
        Dim inner2 = <ns:inner2/>

        Dim test =
        <ns:outer>
            <ns:middle xmlns:ns="http://NewNamespace">
                <ns:inner1/>
                <%= inner2 %>
            </ns:middle>
        </ns:outer>

        ' Display test to see its final form.
        Console.WriteLine(test)
    End Sub

End Class
```

This code displays the following text:

```
<ns:outer xmlns:ns="http://SomeNamespace">
  <ns:middle xmlns:ns="http://NewNamespace">
    <ns:inner1 />
    <inner2 xmlns="http://SomeNamespace" />
  </ns:middle>
</ns:outer>
```

Notice that the compiler converted the prefix of the global XML namespace into a prefix definition for the XML

namespace. The <ns:middle> element redefines the XML namespace prefix for the <ns:inner1> element. However, the <ns:inner2> element uses the namespace defined by the `Imports` statement.

See also

- XElement
- Names of Declared XML Elements and Attributes
- XML Comment Literal
- XML CDATA Literal
- XML Literals
- Creating XML in Visual Basic
- Embedded Expressions in XML
- Imports Statement (XML Namespace)

XML Document Literal (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

A literal representing an XDocument object.

Syntax

```
<?xml version="1.0" [encoding="encoding"] [standalone="standalone"] ?>
[ piCommentList ]
rootElement
[ piCommentList ]
```

Parts

TERM	DEFINITION
`encoding`	Optional. Literal text declaring which encoding the document uses.
`standalone`	Optional. Literal text. Must be "yes" or "no".
`piCommentList`	Optional. List of XML processing instructions and XML comments. Takes the following format: `piComment [ piComment ... ]` Each `piComment` can be one of the following: - XML Processing Instruction Literal. - XML Comment Literal.

`rootElement`	Required. Root element of the document. The format is one of the following:
	
	• XML Element Literal.
	• Embedded expression of the form `<%=` `elementExp` `%>`. The `elementExp` returns one of the following:
	
	  ◦ An XElement object.
	  ◦ A collection that contains one XElement object and any number of XProcessingInstruction and XComment objects.
	
	For more information, see Embedded Expressions in XML.

Return Value

An XDocument object.

Remarks

An XML document literal is identified by the XML declaration at the start of the literal. Although each XML document literal must have exactly one root XML element, it can have any number of XML processing instructions and XML comments.

An XML document literal cannot appear in an XML element.

> **NOTE**
> An XML literal can span multiple lines without using line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML document literal into calls to the XDocument and XDeclaration constructors.

Example

The following example creates an XML document that has an XML declaration, a processing instruction, a comment, and an element that contains another element.

```
Dim libraryRequest As XDocument =
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <?xml-stylesheet type="text/xsl" href="show_book.xsl"?>
    <!-- Tests that the application works. -->
    <books>
        <book/>
    </books>
Console.WriteLine(libraryRequest)
```

See also

- XElement
- XProcessingInstruction
- XComment
- XDocument
- XML Processing Instruction Literal
- XML Comment Literal
- XML Element Literal
- XML Literals
- Creating XML in Visual Basic
- Embedded Expressions in XML

XML CDATA Literal (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

A literal representing an XCData object.

Syntax

```
<![CDATA[content]]>
```

Parts

`<![CDATA[`
Required. Denotes the start of the XML CDATA section.

`content`
Required. Text content to appear in the XML CDATA section.

`]]>`
Required. Denotes the end of the section.

Return Value

An XCData object.

Remarks

XML CDATA sections contain raw text that should be included, but not parsed, with the XML that contains it. A XML CDATA section can contain any text. This includes reserved XML characters. The XML CDATA section ends with the sequence "]]>". This implies the following points:

- You cannot use an embedded expression in an XML CDATA literal because the embedded expression delimiters are valid XML CDATA content.
- XML CDATA sections cannot be nested, because `content` cannot contain the value "]]>".

You can assign an XML CDATA literal to a variable, or include it in an XML element literal.

> **NOTE**
> An XML literal can span multiple lines but does not use line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML CDATA literal to a call to the XCData constructor.

Example

The following example creates a CDATA section that contains the text "Can contain literal <XML> tags".

```
Dim cdata As XCData = <![CDATA[Can contain literal <XML> tags]]>
```

See also

- XCData
- XML Element Literal
- XML Literals
- Creating XML in Visual Basic

XML Comment Literal (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

A literal representing an XComment object.

Syntax

```
<!-- content -->
```

Parts

TERM	DEFINITION
`<!--`	Required. Denotes the start of the XML comment.
`content`	Required. Text to appear in the XML comment. Cannot contain a series of two hyphens (--) or end with a hyphen adjacent to the closing tag.
`-->`	Required. Denotes the end of the XML comment.

Return Value

An XComment object.

Remarks

XML comment literals do not contain document content; they contain information about the document. The XML comment section ends with the sequence "-->". This implies the following points:

- You cannot use an embedded expression in an XML comment literal because the embedded expression delimiters are valid XML comment content.
- XML comment sections cannot be nested, because `content` cannot contain the value "-->".

You can assign an XML comment literal to a variable, or you can include it in an XML element literal.

> **NOTE**
> An XML literal can span multiple lines without using line continuation characters. This feature enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML comment literal to a call to the XComment constructor.

Example

The following example creates an XML comment that contains the text "This is a comment".

```
Dim com As XComment = <!-- This is a comment -->
```

See also

- XComment
- XML Element Literal
- XML Literals
- Creating XML in Visual Basic

XML Processing Instruction Literal (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

A literal representing an XProcessingInstruction object.

Syntax

```
<?piName [ = piData ] ?>
```

Parts

`<?`
Required. Denotes the start of the XML processing instruction literal.

`piName`
Required. Name indicating which application the processing instruction targets. Cannot begin with "xml" or "XML".

`piData`
Optional. String indicating how the application targeted by `piName` should process the XML document.

`?>`
Required. Denotes the end of the processing instruction.

Return Value

An XProcessingInstruction object.

Remarks

XML processing instruction literals indicate how applications should process an XML document. When an application loads an XML document, the application can check the XML processing instructions to determine how to process the document. The application interprets the meaning of `piName` and `piData`.

The XML document literal uses syntax that is similar to that of the XML processing instruction. For more information, see XML Document Literal.

> **NOTE**
> The `piName` element cannot begin with the strings "xml" or "XML", because the XML 1.0 specification reserves those identifiers.

You can assign an XML processing instruction literal to a variable or include it in an XML document literal.

> **NOTE**
> An XML literal can span multiple lines without needing line continuation characters. This enables you to copy content from an XML document and paste it directly into a Visual Basic program.

The Visual Basic compiler converts the XML processing instruction literal to a call to the XProcessingInstruction constructor.

Example

The following example creates a processing instruction identifying a style-sheet for an XML document.

```
Dim pi As XProcessingInstruction =
  <?xml-stylesheet type="text/xsl" href="show_book.xsl"?>
```

See also

- XProcessingInstruction
- XML Document Literal
- XML Literals
- Creating XML in Visual Basic

Error Messages (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

When you write, compile, or run a Visual Basic application, the following types of errors can occur:

1. Design-time errors, which occur when you write an application in Visual Studio.
2. Compile-time errors, which occur when you compile an application in Visual Studio or at a command prompt.
3. Run-time errors, which occur when you run an application in Visual Studio or as a stand-alone executable file.

For information about how to troubleshoot a specific error, see Additional Resources for Visual Basic Programmers.

Run Time Errors

If a Visual Basic application tries to perform an action that the system can't execute, a run-time error occurs, and Visual Basic throws an `Exception` object. Visual Basic can generate custom errors of any data type, including `Exception` objects, by using the `Throw` statement. An application can identify the error by displaying the error number and message of a caught exception. If an error isn't caught, the application ends.

The code can trap and examine run-time errors. If you enclose the code that produces the error in a `Try` block, you can catch any thrown error within a matching `Catch` block. For information about how to trap errors at run time and respond to them in your code, see Try...Catch...Finally Statement.

Compile Time Errors

If the Visual Basic compiler encounters a problem in the code, a compile-time error occurs. In the Code Editor, you can easily identify which line of code caused the error because a wavy line appears under that line of code. The error message appears if you either point to the wavy underline or open the **Error List**, which also shows other messages.

If an identifier has a wavy underline and a short underline appears under the rightmost character, you can generate a stub for the class, constructor, method, property, field or enum. For more information, see Generate From Usage.

By resolving warnings from the Visual Basic compiler, you might be able to write code that runs faster and has fewer bugs. These warnings identify code that may cause errors when the application is run. For example, the compiler warns you if you try to invoke a member of an unassigned object variable, return from a function without setting the return value, or execute a `Try` block with errors in the logic to catch exceptions. For more information about warnings, including how to turn them on and off, see Configuring Warnings in Visual Basic.

'#ElseIf' must be preceded by a matching '#If' or '#ElseIf'

4/9/2019 • 2 minutes to read • Edit Online

`#ElseIf` is a conditional compilation directive. An `#ElseIf` clause must be preceded by a matching `#If` or `#ElseIf` clause.

Error ID: BC30014

To correct this error

1. Check that a preceding `#If` or `#ElseIf` has not been separated from this `#ElseIf` by an intervening conditional compilation block or an incorrectly placed `#End If`.
2. If the `#ElseIf` is preceded by a `#Else` directive, either remove the `#Else` or change it to an `#ElseIf`.
3. If everything else is in order, add an `#If` directive to the beginning of the conditional compilation block.

See also

- #If...Then...#Else Directives

'#Region' and '#End Region' statements are not valid within method bodies/multiline lambdas

4/9/2019 • 2 minutes to read • Edit Online

The `#Region` block must be declared at a class, module, or namespace level. A collapsible region can include one or more procedures, but it cannot begin or end inside of a procedure.

Error ID: BC32025

To correct this error

1. Ensure that the preceding procedure is properly terminated with an `End Function` or `End Sub` statement.
2. Ensure that the `#Region` and `#End Region` directives are in the same code block.

See also

- #Region Directive

'<attribute>' cannot be applied because the format of the GUID '<number>' is not correct

4/9/2019 • 2 minutes to read • Edit Online

A `COMClassAttribute` attribute block specifies a globally unique identifier (GUID) that does not conform to the proper format for a GUID. `COMClassAttribute` uses GUIDs to uniquely identify the class, the interface, and the creation event.

A GUID consists of 16 bytes, of which the first eight are numeric and the last eight are binary. It is generated by Microsoft utilities such as uuidgen.exe and is guaranteed to be unique in space and time.

Error ID: BC32500

To correct this error

1. Determine the correct GUID or GUIDs necessary to identify the COM object.
2. Ensure that the GUID strings presented to the `COMClassAttribute` attribute block are copied correctly.

See also

- Guid
- Attributes overview

'<classname>' is not CLS-compliant because the interface '<interfacename>' it implements is not CLS-compliant

4/28/2019 • 2 minutes to read • Edit Online

A class or interface is marked as `<CLSCompliant(True)>` when it derives from or implements a type that is marked as `<CLSCompliant(False)>` or is not marked.

For a class or interface to be compliant with the Language Independence and Language-Independent Components (CLS), its entire inheritance hierarchy must be compliant. That means every type from which it inherits, directly or indirectly, must be compliant. Similarly, if a class implements one or more interfaces, they must all be compliant throughout their inheritance hierarchies.

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40029

To correct this error

- If you require CLS compliance, define this type within a different inheritance hierarchy or implementation scheme.
- If you require that this type remain within its current inheritance hierarchy or implementation scheme, remove the CLSCompliantAttribute from its definition or mark it as `<CLSCompliant(False)>`.

'<elementname>' is obsolete (Visual Basic Warning)

4/28/2019 • 2 minutes to read • Edit Online

A statement attempts to access a programming element which has been marked with the ObsoleteAttribute attribute and the directive to treat it as a warning.

You can mark any programming element as being no longer in use by applying ObsoleteAttribute to it. If you do this, you can set the attribute's IsError property to either `True` or `False`. If you set it to `True`, the compiler treats an attempt to use the element as an error. If you set it to `False`, or let it default to `False`, the compiler issues a warning if there is an attempt to use the element.

By default, this message is a warning, because the IsError property of ObsoleteAttribute is `False`. For more information about hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40008

To correct this error

- Ensure that the source-code reference is spelling the element name correctly.

See also

- Attributes overview

'<eventname>' is an event, and cannot be called directly

4/9/2019 • 2 minutes to read • Edit Online

'< `eventname` >' is an event, and so cannot be called directly. Use a `RaiseEvent` statement to raise an event.

A procedure call specifies an event for the procedure name. An event handler is a procedure, but the event itself is a signaling device, which must be raised and handled.

Error ID: BC32022

To correct this error

1. Use a `RaiseEvent` statement to signal an event and invoke the procedure or procedures that handle it.

See also

- RaiseEvent Statement

'<expression>' cannot be used as a type constraint

4/28/2019 • 2 minutes to read • Edit Online

A constraint list includes an expression that does not represent a valid constraint on a type parameter.

A constraint list imposes requirements on the type argument passed to the type parameter. You can specify the following requirements in any combination:

- The type argument must implement one or more interfaces
- The type argument must inherit from at most one class
- The type argument must expose a parameterless constructor that the creating code can access (include the `New` constraint)

If you do not include any specific class or interface in the constraint list, you can impose a more general requirement by specifying one of the following:

- The type argument must be a value type (include the `Structure` constraint)
- The type argument must be a reference type (include the `Class` constraint)

You cannot specify both `Structure` and `Class` for the same type parameter, and you cannot specify either one more than once.

Error ID: BC32061

To correct this error

- Verify that the expression and its elements are spelled correctly.
- If the expression does not qualify for the preceding list of requirements, remove it from the constraint list.
- If the expression refers to an interface or class, verify that the compiler has access to that interface or class. You might need to qualify its name, and you might need to add a reference to your project. For more information, see "References to Projects" in References to Declared Elements.

See also

- Generic Types in Visual Basic
- Value Types and Reference Types
- References to Declared Elements

'<functionname>' is not declared (Smart Device/Visual Basic Compiler Error)

4/28/2019 • 2 minutes to read • Edit Online

< `functionname` > is not declared. File I/O functionality is normally available in the `Microsoft.VisualBasic` namespace, but the targeted version of the .NET Compact Framework does not support it.

Error ID: BC30766

To correct this error

- Perform file operations with functions defined in the `System.IO` namespace.

See also

- System.IO
- File Access with Visual Basic

'<interfacename>.<membername>' is already implemented by the base class '<baseclassname>'. Re-implementation of <type> assumed

4/28/2019 • 2 minutes to read • Edit Online

A property, procedure, or event in a derived class uses an `Implements` clause specifying an interface member that is already implemented in the base class.

A derived class can reimplement an interface member that is implemented by its base class. This is not the same as overriding the base class implementation. For more information, see Implements.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42015

To correct this error

- If you intend to reimplement the interface member, you do not need to take any action. Code in your derived class accesses the reimplemented member unless you use the `MyBase` keyword to access the base class implementation.
- If you do not intend to reimplement the interface member, remove the `Implements` clause from the property, procedure, or event declaration.

See also

- Interfaces

'<keyword>' is valid only within an instance method

4/28/2019 • 2 minutes to read • Edit Online

The `Me`, `MyClass`, and `MyBase` keywords refer to specific class instances. You cannot use them inside a shared `Function` or `Sub` procedure.

Error ID: BC30043

To correct this error

- Remove the keyword from the procedure, or remove the `Shared` keyword from the procedure declaration.

See also

- Object Variable Assignment
- Me, My, MyBase, and MyClass
- Inheritance Basics

'<membername>' cannot expose type '<typename>' outside the project through <containertype> '<containertypename>'

4/28/2019 • 2 minutes to read • Edit Online

A variable, procedure parameter, or function return is exposed outside its container, but it is declared as a type that must not be exposed outside the container.

The following skeleton code shows a situation that generates this error.

```
Private Class privateClass
End Class
Public Class mainClass
    Public exposedVar As New privateClass
End Class
```

A type that is declared `Protected`, `Friend`, `Protected Friend`, or `Private` is intended to have limited access outside its declaration context. Using it as the data type of a variable with less restricted access would defeat this purpose. In the preceding skeleton code, `exposedVar` is `Public` and would expose `privateClass` to code that should not have access to it.

Error ID: BC30909

To correct this error

- Change the access level of the variable, procedure parameter, or function return to be at least as restrictive as the access level of its data type.

See also

- Access levels in Visual Basic

'<membername>' is ambiguous across the inherited interfaces '<interfacename1>' and '<interfacename2>'

4/28/2019 • 2 minutes to read • Edit Online

The interface inherits two or more members with the same name from multiple interfaces.

Error ID: BC30685

To correct this error

- Cast the value to the base interface that you want to use; for example:

```
Interface Left
    Sub MySub()
End Interface

Interface Right
    Sub MySub()
End Interface

Interface LeftRight
    Inherits Left, Right
End Interface

Module test
    Sub Main()
        Dim x As LeftRight
        ' x.MySub()  'x is ambiguous.
        CType(x, Left).MySub() ' Cast to base type.
        CType(x, Right).MySub() ' Call the other base type.
    End Sub
End Module
```

See also

- Interfaces

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>'

4/9/2019 • 2 minutes to read • Edit Online

<message> This error could also be due to mixing a file reference with a project reference to assembly '<assemblyname>. In this case, try replacing the file reference to '<assemblyfilename>' in project '<projectname1>' with a project reference to '<projectname2>'.

Code in your project accesses a member of another project, but the configuration of your solution does not allow the Visual Basic compiler to resolve the reference.

To access a type defined in another assembly, the Visual Basic compiler must have a reference to that assembly. This must be a single, unambiguous reference that does not cause circular references among projects.

Error ID: BC30971

To correct this error

1. Determine which project produces the best assembly for your project to reference. For this decision, you might use criteria such as ease of file access and frequency of updates.
2. In your project properties, add a reference to the project that contains the assembly that defines the type you are using.

See also

- Managing references in a project
- References to Declared Elements
- Managing Project and Solution Properties
- Troubleshooting Broken References

'<methodname>' has multiple definitions with identical signatures

4/28/2019 • 2 minutes to read • Edit Online

A `Function` or `Sub` procedure declaration uses the identical procedure name and argument list as a previous declaration. One possible cause is an attempt to overload the original procedure. Overloaded procedures must have different argument lists.

Error ID: BC30269

To correct this error

- Change the procedure name or the argument list, or remove the duplicate declaration.

See also

- References to Declared Elements
- Considerations in Overloading Procedures

'<name>' is ambiguous in the namespace '<namespacename>'

4/28/2019 • 2 minutes to read • Edit Online

You have provided a name that is ambiguous and therefore conflicts with another name. The Visual Basic compiler does not have any conflict resolution rules; you must disambiguate names yourself.

Error ID: BC30560

To correct this error

- Fully qualify the name.

See also

- Namespaces in Visual Basic
- Namespace Statement

'<name1>' is ambiguous, imported from the namespaces or types '<name2>'

4/9/2019 • 2 minutes to read • Edit Online

You have provided a name that is ambiguous and therefore conflicts with another name. The Visual Basic compiler does not have any conflict resolution rules; you must disambiguate names yourself.

Error ID: BC30561

To correct this error

1. Disambiguate the name by removing namespace imports.
2. Fully qualify the name.

See also

- Imports Statement (.NET Namespace and Type)
- Namespaces in Visual Basic
- Namespace Statement

<proceduresignature1> is not CLS-compliant because it overloads <proceduresignature2> which differs from it only by array of array parameter types or by the rank of the array parameter types

4/28/2019 • 2 minutes to read • Edit Online

A procedure or property is marked as `<CLSCompliant(True)>` when it overrides another procedure or property and the only difference between their parameter lists is the nesting level of a jagged array or the rank of an array.

In the following declarations, the second and third declarations generate this error.

```
Overloads Sub processArray(ByVal arrayParam() As Integer)
```

```
Overloads Sub processArray(ByVal arrayParam()() As Integer)
```

```
Overloads Sub processArray(ByVal arrayParam(,) As Integer)
```

The second declaration changes the original one-dimensional parameter `arrayParam` to an array of arrays. The third declaration changes `arrayParam` to a two-dimensional array (rank 2). While Visual Basic allows overloads to differ only by one of these changes, such overloading is not compliant with the Language Independence and Language-Independent Components (CLS).

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40035

To correct this error

- If you require CLS compliance, define your overloads to differ from each other in more ways than only the changes cited on this Help page.
- If you require that the overloads differ only by the changes cited on this Help page, remove the CLSCompliantAttribute from their definitions or mark them as `<CLSCompliant(False)>`.

See also

- Procedure Overloading
- Overloads

<type1>'<typename>' must implement '<membername>' for interface '<interfacename>'

4/9/2019 • 2 minutes to read • Edit Online

'<typename>' must implement '<membername>' for interface '<interfacename>'. Implementing property must have matching 'ReadOnly'/'WriteOnly' specifiers.

A class or structure claims to implement an interface but does not implement a procedure, property, or event defined by the interface. Every member of the interface must be implemented.

Error ID: BC30154

To correct this error

1. Declare a member with the same name and signature as defined in the interface. Be sure to include at least the `End Function`, `End Sub`, or `End Property` statement.
2. Add an `Implements` clause to the end of the `Function`, `Sub`, `Property`, or `Event` statement. For example:

   ```
   Public Event ItHappened() Implements IBaseInterface.ItHappened
   ```

3. When implementing a property, make sure that `ReadOnly` or `WriteOnly` is used in the same way as in the interface definition.
4. When implementing a property, declare `Get` and `Set` procedures, as appropriate.

See also

- Implements Statement
- Interfaces

<type1>'<typename>' must implement '<methodname>' for interface '<interfacename>'

4/9/2019 • 2 minutes to read • Edit Online

A class or structure claims to implement an interface but does not implement a procedure defined by the interface. Every member of the interface must be implemented.

Error ID: BC30149

To correct this error

1. Declare a procedure with the same name and signature as defined in the interface. Be sure to include at least the `End Function` or `End Sub` statement.
2. Add an `Implements` clause to the end of the `Function` or `Sub` statement. For example:

```
Public Sub DoSomething() Implements IBaseInterface.DoSomething
```

See also

- Implements Statement
- Interfaces

'<typename>' cannot inherit from <type> '<basetypename>' because it expands the access of the base <type> outside the assembly

4/28/2019 • 2 minutes to read • Edit Online

A class or interface inherits from a base class or interface but has a less restrictive access level.

For example, a `Public` interface inherits from a `Friend` interface, or a `Protected` class inherits from a `Private` class. This exposes the base class or interface to access beyond the intended level.

Error ID: BC30910

To correct this error

- Change the access level of the derived class or interface to be at least as restrictive as that of the base class or interface.

  -or-

- If you require the less restrictive access level, remove the `Inherits` statement. You cannot inherit from a more restricted base class or interface.

See also

- Class Statement
- Interface Statement
- Inherits Statement
- Access levels in Visual Basic

'<typename>' is a delegate type

4/28/2019 • 2 minutes to read • Edit Online

'<typename>' is a delegate type. Delegate construction permits only a single AddressOf expression as an argument list. Often an AddressOf expression can be used instead of a delegate construction.

A `New` clause creating an instance of a delegate class supplies an invalid argument list to the delegate constructor.

You can supply only a single `AddressOf` expression when creating a new delegate instance.

This error can result if you do not pass any arguments to the delegate constructor, if you pass more than one argument, or if you pass a single argument that is not a valid `AddressOf` expression.

Error ID: BC32008

To correct this error

- Use a single `AddressOf` expression in the argument list for the delegate class in the `New` clause.

See also

- New Operator
- AddressOf Operator
- Delegates
- How to: Invoke a Delegate Method

'<typename>' is a type and cannot be used as an expression

4/28/2019 • 2 minutes to read • Edit Online

A type name occurs where an expression is required. An expression must consist of some combination of variables, constants, literals, properties, and `Function` procedure calls.

Error ID: BC30108

To correct this error

- Remove the type name and construct the expression using valid elements.

See also

- Operators and Expressions

A double quote is not a valid comment token for delimited fields where EscapeQuote is set to True

4/28/2019 • 2 minutes to read • Edit Online

A quotation mark has been supplied as the delimiter for the `TextFieldParser`, but `EscapeQuotes` is set to `True`.

To correct this error

- Set `EscapeQuotes` to `False`.

See also

- SetDelimiters
- Delimiters
- TextFieldParser
- How to: Read From Comma-Delimited Text Files

A property or method call cannot include a reference to a private object, either as an argument or as a return value

4/28/2019 • 2 minutes to read • Edit Online

Among the possible causes of this error are:

- A client invoked a property or method of an out-of-process component and attempted to pass a reference to a private object as one of the arguments.
- An out-of-process component invoked a call-back method on its client and attempted to pass a reference to a private object.
- An out-of-process component attempted to pass a reference to a private object as an argument of an event it was raising.
- A client attempted to assign a private object reference to a `ByRef` argument of an event it was handling.

To correct this error

1. Remove the reference.

See also

- Private

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'

8/22/2019 • 2 minutes to read • Edit Online

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'. Consider changing the 'Embed Interop Types' property on either assembly.

You have added a reference to an assembly (assembly1) that has the `Embed Interop Types` property set to `True`. This instructs the compiler to embed interop type information from that assembly. However, the compiler cannot embed interop type information from that assembly because another assembly that you have referenced (assembly2) also references that assembly (assembly1) and has the `Embed Interop Types` property set to `False`.

> **NOTE**
> Setting the `Embed Interop Types` property on an assembly reference to `True` is equivalent to referencing the assembly by using the `/link` option for the command-line compiler.

Error ID: BC40059

To address this warning

- To embed interop type information for both assemblies, set the `Embed Interop Types` property on all references to assembly1 to `True`.
- To remove the warning, you can set the `Embed Interop Types` property of assembly1 to `False`. In this case, interop type information is provided by a primary interop assembly (PIA).

See also

- /link (Visual Basic)
- Interoperating with Unmanaged Code

A startup form has not been specified

4/9/2019 • 2 minutes to read • Edit Online

The application uses the WindowsFormsApplicationBase class but does not specify the startup form.

This can occur if the **Enable application framework** check box is selected in the project designer but the **Startup form** is not specified. For more information, see Application Page, Project Designer (Visual Basic).

To correct this error

1. Specify a startup object for the application.

   For more information, see Application Page, Project Designer (Visual Basic).

2. Override the OnCreateMainForm method to set the MainForm property to the startup form.

See also

- WindowsFormsApplicationBase
- OnCreateMainForm
- MainForm
- Overview of the Visual Basic Application Model

Access of shared member through an instance; qualifying expression will not be evaluated

8/22/2019 • 2 minutes to read • Edit Online

An instance variable of a class or structure is used to access a `Shared` variable, property, procedure, or event defined in that class or structure. This warning can also occur if an instance variable is used to access an implicitly shared member of a class or structure, such as a constant or enumeration, or a nested class or structure.

The purpose of sharing a member is to create only a single copy of that member and make that single copy available to every instance of the class or structure in which it is declared. It is consistent with this purpose to access a `Shared` member through the name of its class or structure, rather than through a variable that holds an individual instance of that class or structure.

Accessing a `Shared` member through an instance variable can make your code more difficult to understand by obscuring the fact that the member is `Shared`. Furthermore, if such access is part of an expression that performs other actions, such as a `Function` procedure that returns an instance of the shared member, Visual Basic bypasses the expression and any other actions it would otherwise perform.

For more information and an example, see Shared.

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42025

To correct this error

- Use the name of the class or structure that defines the `Shared` member to access it, as shown in the following example.

```
Public Class testClass
    Public Shared Sub sayHello()
        MsgBox("Hello")
    End Sub
End Class

Module testModule
    Public Sub Main()
        ' Access a shared method through an instance variable.
        ' This generates a warning.
        Dim tc As New testClass
        tc.sayHello()

        ' Access a shared method by using the class name.
        ' This does not generate a warning.
        testClass.sayHello()
    End Sub
End Module
```

NOTE

Be alert for the effects of scope when two programming elements have the same name. In the previous example, if you declare an instance by using `Dim testClass as testClass = Nothing`, the compiler treats a call to `testClass.sayHello()` as an access of the method through the class name, and no warning occurs.

See also

- Shared
- Scope in Visual Basic

'AddressOf' operand must be the name of a method (without parentheses)

4/9/2019 • 2 minutes to read • Edit Online

The `AddressOf` operator creates a procedure delegate instance that references a specific procedure. The syntax is as follows.

```
AddressOf procedurename
```

You inserted parentheses around the argument following `AddressOf`, where none are needed.

Error ID: BC30577

To correct this error

1. Remove the parentheses around the argument following `AddressOf`.
2. Make sure the argument is a method name.

See also

- AddressOf Operator
- Delegates

An unexpected error has occurred because an operating system resource required for single instance startup cannot be acquired

4/28/2019 • 2 minutes to read • Edit Online

The application could not acquire a necessary operating system resource. Some of the possible causes for this problem are:

- The application does not have permissions to create named operating-system objects.
- The common language runtime does not have permissions to create memory-mapped files.
- The application needs to access an operating-system object, but another process is using it.

To correct this error

1. Check that the application has sufficient permissions to create named operating-system objects.
2. Check that the common language runtime has sufficient permissions to create memory-mapped files.
3. Restart the computer to clear any process that may be using the resource needed to connect to the original instance application.
4. Note the circumstances under which the error occurred, and call Microsoft Product Support Services

See also

- Application Page, Project Designer (Visual Basic)
- Debugger Basics
- Talk to Us

Anonymous type member name can be inferred only from a simple or qualified name with no arguments

4/28/2019 • 2 minutes to read • Edit Online

You cannot infer an anonymous type member name from a complex expression.

```
Dim numbers() As Integer = {1, 2, 3, 4, 5}
' Not valid.
' Dim instanceName1 = New With {numbers(3)}
```

For more information about sources from which anonymous types can and cannot infer member names and types, see How to: Infer Property Names and Types in Anonymous Type Declarations.

Error ID: BC36556

To correct this error

- Assign the expression to a member name, as shown in the following code:

```
Dim instanceName2 = New With {.number = numbers(3)}
```

See also

- Anonymous Types
- How to: Infer Property Names and Types in Anonymous Type Declarations

Argument not optional (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

The number and types of arguments must match those expected. Either there is an incorrect number of arguments, or an omitted argument is not optional. An argument can only be omitted from a call to a user-defined procedure if it was declared `Optional` in the procedure definition.

To correct this error

1. Supply all necessary arguments.
2. Make sure omitted arguments are optional. If they are not, either supply the argument in the call, or declare the parameter `Optional` in the definition.

See also

- Error Types

Array bounds cannot appear in type specifiers

7/26/2019 • 2 minutes to read • Edit Online

Array sizes cannot be declared as part of a data type specifier.

Error ID: BC30638

To correct this error

- Specify the size of the array immediately following the variable name instead of placing the array size after the type, as shown in the following example.

```
Dim Array(8) As Integer
```

- Define an array and initialize it with the desired number of elements, as shown in the following example.

```
Dim Array2() As Integer = New Integer(8) {}
```

See also

- Arrays

Array declared as for loop control variable cannot be declared with an initial size

4/28/2019 • 2 minutes to read • Edit Online

A `For Each` loop uses an array as its *element* iteration variable but initializes that array.

The following statements show how this error can be generated.

```
Dim arrayList As New List(Of Integer())
For Each listElement() As Integer In arrayList
For Each listElement(1) As Integer In arrayList
```

The first `For Each` statement is the correct way to access elements of `arrayList`. The second `For Each` statement generates this error.

Error ID: BC32039

To correct this error

- Remove the initialization from the declaration of the *element* iteration variable.

See also

- For...Next Statement
- Arrays
- Collections

Array subscript expression missing

4/28/2019 • 2 minutes to read • Edit Online

An array initialization leaves out one or more of the subscripts that define the array bounds. For example, the statement might contain the expression `myArray (5,5,,10)`, which leaves out the third subscript.

Error ID: BC30306

To correct this error

- Supply the missing subscript.

See also

- Arrays

Arrays declared as structure members cannot be declared with an initial size

4/9/2019 • 2 minutes to read • Edit Online

An array in a structure is declared with an initial size. You cannot initialize any structure element, and declaring an array size is one form of initialization.

Error ID: BC31043

To correct this error

1. Define the array in your structure as dynamic (no initial size).
2. If you require a certain size of array, you can redimension a dynamic array with a ReDim Statement when your code is running. The following example illustrates this.

```
Structure demoStruct
    Public demoArray() As Integer
End Structure
Sub useStruct()
    Dim struct As demoStruct
    ReDim struct.demoArray(9)
    Struct.demoArray(2) = 777
End Sub
```

See also

- Arrays
- How to: Declare a Structure

'As Any' is not supported in 'Declare' statements

4/9/2019 • 2 minutes to read • Edit Online

The `Any` data type was used with `Declare` statements in Visual Basic 6.0 and earlier versions to permit the use of arguments that could contain any type of data. Visual Basic supports overloading, however, and so makes the `Any` data type obsolete.

Error ID: BC30828

To correct this error

1. Declare parameters of the specific type you want to use; for example.

```
Declare Function GetUserName Lib "advapi32.dll" Alias "GetUserNameA" (
    ByVal lpBuffer As String,
    ByRef nSize As Integer) As Integer
```

2. Use the MarshalAsAttribute attribute to specify `As Any` when `Void*` is expected by the procedure being called.

```
Declare Sub SetData Lib "..\LIB\UnmgdLib.dll" (
    ByVal x As Short,
    <System.Runtime.InteropServices.MarshalAsAttribute(
        System.Runtime.InteropServices.UnmanagedType.AsAny)>
        ByVal o As Object)
```

See also

- MarshalAsAttribute
- Walkthrough: Calling Windows APIs
- Declare Statement
- Creating Prototypes in Managed Code

Attribute '<attributename>' cannot be applied multiple times

4/9/2019 • 2 minutes to read • Edit Online

The attribute can only be applied once. The `AttributeUsage` attribute determines whether an attribute can be applied more than once.

Error ID: BC30663

To correct this error

1. Make sure the attribute is only applied once.
2. If you are using custom attributes you developed, consider changing their `AttributeUsage` attribute to allow multiple attribute usage, as with the following example.

```
<AttributeUsage(AllowMultiple := True)>
```

See also

- AttributeUsageAttribute
- Creating Custom Attributes
- AttributeUsage

Automation error

4/9/2019 • 2 minutes to read • Edit Online

An error occurred while executing a method or getting or setting a property of an object variable. The error was reported by the application that created the object.

To correct this error

1. Check the properties of the `Err` object to determine the source and nature of the error.
2. Use the `On Error Resume Next` statement immediately before the accessing statement, and then check for errors immediately after the accessing statement.

See also

- Error Types
- Talk to Us

Bad checksum value, non hex digits or odd number of hex digits

4/9/2019 • 2 minutes to read • Edit Online

A checksum value contains invalid hexadecimal digits or has an odd number of digits.

When ASP.NET generates a Visual Basic source file (extension .vb), it calculates a checksum and places it in a hidden source file identified by `#externalchecksum`. It is possible for a user generating a .vb file to do this also, but this process is best left to internal use.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42033

To correct this error

1. If ASP.NET is generating the Visual Basic source file, restart the project build.
2. If this warning persists after restarting, reinstall ASP.NET and try the build again.
3. If the warning still persists, or if you are not using ASP.NET, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- ASP.NET Overview
- Talk to Us

Bad DLL calling convention

4/9/2019 • 2 minutes to read • Edit Online

Arguments passed to a dynamic-link library (DLL) must exactly match those expected by the routine. Calling conventions deal with number, type, and order of arguments. Your program may be calling a routine in a DLL that is being passed the wrong type or number of arguments.

To correct this error

1. Make sure all argument types agree with those specified in the declaration of the routine that you are calling.
2. Make sure you are passing the same number of arguments indicated in the declaration of the routine that you are calling.
3. If the DLL routine expects arguments by value, make sure `ByVal` is specified for those arguments in the declaration for the routine.

See also

- Error Types
- Call Statement
- Declare Statement

Bad file mode

4/28/2019 • 2 minutes to read • Edit Online

Statements used in manipulating file contents must be appropriate to the mode in which the file was opened. Possible causes include:

- A `FilePutObject` or `FileGetObject` statement specifies a sequential file.
- A `Print` statement specifies a file opened for an access mode other than `Output` or `Append`.
- An `Input` statement specifies a file opened for an access mode other than `Input`
- An attempt to write to a read-only file.

To correct this error

- Make sure `FilePutObject` and `FileGetObject` are only referring to files open for `Random` or `Binary` access.
- Make sure `Print` specifies a file opened for either `Output` or `Append` access mode. If not, use a different statement to place data in the file, or reopen the file in an appropriate mode.
- Make sure `Input` specifies a file opened for `Input`. If not, use a different statement to place data in the file or reopen the file in an appropriate mode.
- If you are writing to a read-only file, change the read/write status of the file or do not try to write to it.
- Use the functionality available in the `My.Computer.FileSystem` object.

See also

- FileSystem
- Troubleshooting: Reading from and Writing to Text Files

Bad file name or number

4/28/2019 • 2 minutes to read • Edit Online

An error occurred while trying to access the specified file. Among the possible causes for this error are:

- A statement refers to a file with a file name or number that was not specified in the `FileOpen` statement or that was specified in a `FileOpen` statement but was subsequently closed.
- A statement refers to a file with a number that is out of the range of file numbers.
- A statement refers to a file name or number that is not valid.

To correct this error

1. Make sure the file name is specified in a `FileOpen` statement. Note that if you invoked the `FileClose` statement without arguments, you may have inadvertently closed all open files.
2. If your code is generating file numbers algorithmically, make sure the numbers are valid.
3. Check the file names to make sure they conform to operating system conventions.

See also

- FileOpen
- Visual Basic Naming Conventions

Bad record length

4/28/2019 • 2 minutes to read • Edit Online

Among the possible causes of this error are:

- The length of a record variable specified in a `FileGet`, `FileGetObject`, `FilePut` or `FilePutObject` statement differs from the length specified in the corresponding `FileOpen` statement.
- The variable in a `FilePut` or `FilePutObject` statement is or includes a variable-length string.
- The variable in a `FilePut` or `FilePutObject` is or includes a `Variant` type.

To correct this error

1. Make sure the sum of the sizes of fixed-length variables in the user-defined type defining the record variable's type is the same as the value stated in the `FileOpen` statement's `Len` clause.
2. If the variable in a `FilePut` or `FilePutObject` statement is or includes a variable-length string, make sure the variable-length string is at least 2 characters shorter than the record length specified in the `Len` clause of the `FileOpen` statement.
3. If the variable in a `FilePut` or `FilePutObject` is or includes a `Variant` make sure the variable-length string is at least 4 bytes shorter than the record length specified in the `Len` clause of the `FileOpen` statement.

See also

- FileGet
- FileGetObject
- FilePut
- FilePutObject

Because this call is not awaited, the current method continues to run before the call is completed

4/28/2019 • 6 minutes to read • Edit Online

Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'Await' operator to the result of the call.

The current method calls an async method that returns a Task or a Task<TResult> and doesn't apply the Await operator to the result. The call to the async method starts an asynchronous task. However, because no `Await` operator is applied, the program continues without waiting for the task to complete. In most cases, that behavior isn't expected. Usually other aspects of the calling method depend on the results of the call or, minimally, the called method is expected to complete before you return from the method that contains the call.

An equally important issue is what happens with exceptions that are raised in the called async method. An exception that's raised in a method that returns a Task or Task<TResult> is stored in the returned task. If you don't await the task or explicitly check for exceptions, the exception is lost. If you await the task, its exception is rethrown.

As a best practice, you should always await the call.

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42358

To address this warning

- You should consider suppressing the warning only if you're sure that you don't want to wait for the asynchronous call to complete and that the called method won't raise any exceptions. In that case, you can suppress the warning by assigning the task result of the call to a variable.

  The following example shows how to cause the warning, how to suppress it, and how to await the call.

```
Async Function CallingMethodAsync() As Task

    ResultsTextBox.Text &= vbCrLf & "  Entering calling method."

    ' Variable delay is used to slow down the called method so that you
    ' can distinguish between awaiting and not awaiting in the program's output.
    ' You can adjust the value to produce the output that this topic shows
    ' after the code.
    Dim delay = 5000

    ' Call #1.
    ' Call an async method. Because you don't await it, its completion isn't
    ' coordinated with the current method, CallingMethodAsync.
    ' The following line causes the warning.
    CalledMethodAsync(delay)

    ' Call #2.
    ' To suppress the warning without awaiting, you can assign the
    ' returned task to a variable. The assignment doesn't change how
    ' the program runs. However, the recommended practice is always to
    ' await a call to an async method.
    ' Replace Call #1 with the following line.
    'Task delayTask = CalledMethodAsync(delay)

    ' Call #3
    ' To contrast with an awaited call, replace the unawaited call
    ' (Call #1 or Call #2) with the following awaited call. The best
    ' practice is to await the call.

    'Await CalledMethodAsync(delay)

    ' If the call to CalledMethodAsync isn't awaited, CallingMethodAsync
    ' continues to run and, in this example, finishes its work and returns
    ' to its caller.
    ResultsTextBox.Text &= vbCrLf & "  Returning from calling method."
End Function

Async Function CalledMethodAsync(howLong As Integer) As Task

    ResultsTextBox.Text &= vbCrLf & "    Entering called method, starting and awaiting Task.Delay."
    ' Slow the process down a little so you can distinguish between awaiting
    ' and not awaiting. Adjust the value for howLong if necessary.
    Await Task.Delay(howLong)
    ResultsTextBox.Text &= vbCrLf & "    Task.Delay is finished--returning from called method."
End Function
```

In the example, if you choose Call #1 or Call #2, the unawaited async method (`CalledMethodAsync`) finishes after both its caller (`CallingMethodAsync`) and the caller's caller (`StartButton_Click`) are complete. The last line in the following output shows you when the called method finishes. Entry to and exit from the event handler that calls `CallingMethodAsync` in the full example are marked in the output.

```
Entering the Click event handler.
  Entering calling method.
    Entering called method, starting and awaiting Task.Delay.
  Returning from calling method.
Exiting the Click event handler.
    Task.Delay is finished--returning from called method.
```

Example

The following Windows Presentation Foundation (WPF) application contains the methods from the previous example. The following steps set up the application.

1. Create a WPF application, and name it `AsyncWarning`.

2. In the Visual Studio Code Editor, choose the **MainWindow.xaml** tab.

   If the tab isn't visible, open the shortcut menu for MainWindow.xaml in **Solution Explorer**, and then choose **View Code**.

3. Replace the code in the **XAML** view of MainWindow.xaml with the following code.

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton" Content="Start" HorizontalAlignment="Left" Margin="214,28,0,0"
VerticalAlignment="Top" Width="75" HorizontalContentAlignment="Center" FontWeight="Bold"
FontFamily="Aharoni" Click="StartButton_Click" />
        <TextBox x:Name="ResultsTextBox" Margin="0,80,0,0" TextWrapping="Wrap" FontFamily="Lucida
Console"/>
    </Grid>
</Window>
```

   A simple window that contains a button and a text box appears in the **Design** view of MainWindow.xaml.

   For more information about the XAML Designer, see Creating a UI by using XAML Designer. For information about how to build your own simple UI, see the "To create a WPF application" and "To design a simple WPF MainWindow" sections of Walkthrough: Accessing the Web by Using Async and Await.

4. Replace the code in MainWindow.xaml.vb with the following code.

```
Class MainWindow

    Private Async Sub StartButton_Click(sender As Object, e As RoutedEventArgs)

        ResultsTextBox.Text &= vbCrLf & "Entering the Click event handler."
        Await CallingMethodAsync()
        ResultsTextBox.Text &= vbCrLf & "Exiting the Click event handler."
    End Sub

    Async Function CallingMethodAsync() As Task

        ResultsTextBox.Text &= vbCrLf & "  Entering calling method."

        ' Variable delay is used to slow down the called method so that you
        ' can distinguish between awaiting and not awaiting in the program's output.
        ' You can adjust the value to produce the output that this topic shows
        ' after the code.
        Dim delay = 5000

        ' Call #1.
        ' Call an async method. Because you don't await it, its completion isn't
        ' coordinated with the current method, CallingMethodAsync.
        ' The following line causes the warning.
        CalledMethodAsync(delay)

        ' Call #2.
        ' To suppress the warning without awaiting, you can assign the
        ' returned task to a variable. The assignment doesn't change how
        ' the program runs. However, the recommended practice is always to
        ' await a call to an async method.

        ' Replace Call #1 with the following line.
        'Task delayTask = CalledMethodAsync(delay)

        ' Call #3
```

```
        ' To contrast with an awaited call, replace the unawaited call
        ' (Call #1 or Call #2) with the following awaited call. The best
        ' practice is to await the call.

        'Await CalledMethodAsync(delay)

        ' If the call to CalledMethodAsync isn't awaited, CallingMethodAsync
        ' continues to run and, in this example, finishes its work and returns
        ' to its caller.
        ResultsTextBox.Text &= vbCrLf & "  Returning from calling method."
    End Function

    Async Function CalledMethodAsync(howLong As Integer) As Task

        ResultsTextBox.Text &= vbCrLf & "    Entering called method, starting and awaiting Task.Delay."
        ' Slow the process down a little so you can distinguish between awaiting
        ' and not awaiting. Adjust the value for howLong if necessary.
        Await Task.Delay(howLong)
        ResultsTextBox.Text &= vbCrLf & "    Task.Delay is finished--returning from called method."
    End Function

End Class

' Output

' Entering the Click event handler.
'   Entering calling method.
'     Entering called method, starting and awaiting Task.Delay.
'   Returning from calling method.
' Exiting the Click event handler.
'     Task.Delay is finished--returning from called method.

' Output

' Entering the Click event handler.
'   Entering calling method.
'     Entering called method, starting and awaiting Task.Delay.
'     Task.Delay is finished--returning from called method.
'   Returning from calling method.
' Exiting the Click event handler.
```

5. Choose the F5 key to run the program, and then choose the **Start** button.

   The expected output appears at the end of the code.

See also

- Await Operator
- Asynchronous Programming with Async and Await

Cannot convert anonymous type to expression tree because it contains a field that is used in the initialization of another field

4/28/2019 • 2 minutes to read • Edit Online

The compiler does not accept conversion of an anonymous to an expression tree when one property of the anonymous type is used to initialize another property of the anonymous type. For example, in the following code, `Prop1` is declared in the initialization list and then used as the initial value for `Prop2`.

```
Module M2

    Sub ExpressionExample(Of T)(ByVal x As Expressions.Expression(Of Func(Of T)))
    End Sub

    Sub Main()
        ' The following line causes the error.
        ' ExpressionExample(Function() New With {.Prop1 = 2, .Prop2 = .Prop1})

    End Sub
End Module
```

Error ID: BC36548

To correct this error

- Assign the initial value for `Prop1` to a local variable. Assign that variable to both `Prop1` and `Prop2`, as shown in the following code.

```
Sub Main()

    Dim temp = 2
    ExpressionExample(Function() New With {.Prop1 = temp, .Prop2 = temp})

End Sub
```

See also

- Anonymous Types (Visual Basic)
- Expression Trees (Visual Basic)
- How to: Use Expression Trees to Build Dynamic Queries (Visual Basic)

Cannot create ActiveX Component

4/28/2019 • 2 minutes to read • Edit Online

You tried to place an ActiveX control on a form at design time or add a form to a project with an ActiveX control on it, but the associated information in the registry could not be found.

To correct this error

- The information in the registry may have been deleted or corrupted. Reinstall the ActiveX control or contact the control vendor.

See also

- Error Types
- Talk to Us

Cannot refer to '<name>' because it is a member of the value-typed field '<name>' of class '<classname>' which has 'System.MarshalByRefObject' as a base class

4/9/2019 • 2 minutes to read • Edit Online

The `System.MarshalByRefObject` class enables applications that support remote access to objects across application domain boundaries. Types must inherit from the `MarshalByRejectObject` class when the type is used across application domain boundaries. The state of the object must not be copied because the members of the object are not usable outside the application domain in which they were created.

Error ID: BC30310

To correct this error

1. Check the reference to make sure the member being referred to is valid.
2. Explicitly qualify the member with the `Me` keyword.

See also

- MarshalByRefObject
- Dim Statement

Cannot refer to an instance member of a class from within a shared method or shared member initializer without an explicit instance of the class

4/9/2019 • 2 minutes to read • Edit Online

You have tried to refer to a non-shared member of a class from within a shared procedure. The following example demonstrates such a situation.

```
Class sample
    Public x as Integer
    Public Shared Sub setX()
        x = 10
    End Sub
End Class
```

In the preceding example, the assignment statement `x = 10` generates this error message. This is because a shared procedure is attempting to access an instance variable.

The variable `x` is an instance member because it is not declared as Shared. Each instance of class `sample` contains its own individual variable `x`. When one instance sets or changes the value of `x`, it does not affect the value of `x` in any other instance.

However, the procedure `setX` is `Shared` among all instances of class `sample`. This means it is not associated with any one instance of the class, but rather operates independently of individual instances. Because it has no connection with a particular instance, `setX` cannot access an instance variable. It must operate only on `Shared` variables. When `setX` sets or changes the value of a shared variable, that new value is available to all instances of the class.

Error ID: BC30369

To correct this error

1. Decide whether you want the member to be shared among all instances of the class, or kept individual for each instance.
2. If you want a single copy of the member to be shared among all instances, add the `Shared` keyword to the member declaration. Retain the `Shared` keyword in the procedure declaration.
3. If you want each instance to have its own individual copy of the member, do not specify `Shared` for the member declaration. Remove the `Shared` keyword from the procedure declaration.

See also

- Shared

Can't create necessary temporary file

4/9/2019 • 2 minutes to read • Edit Online

Either the drive is full that contains the directory specified by the TEMP environment variable, or the TEMP environment variable specifies an invalid or read-only drive or directory.

To correct this error

1. Delete files from the drive, if full.
2. Specify a different drive in the TEMP environment variable.
3. Specify a valid drive for the TEMP environment variable.
4. Remove the read-only restriction from the currently specified drive or directory.

See also

- Error Types

Can't open '<filename>' for writing

4/9/2019 • 2 minutes to read • Edit Online

The specified file cannot be opened for writing, perhaps because it has already been opened.

Error ID: BC2012

To correct this error

1. Close the file and reopen it.
2. Check the file's permissions.

See also

- WriteAllText
- WriteAllBytes
- Writing to Files

Class '<classname>' cannot be found

4/9/2019 • 2 minutes to read • Edit Online

Class '<classname>' cannot be found. This condition is usually the result of a mismatched 'Microsoft.VisualBasic.dll'.

A defined member could not be located.

Error ID: BC31098

To correct this error

1. Compile the program again to see if the error recurs.
2. If the error recurs, save your work and restart Visual Studio.
3. If the error persists, reinstall Visual Basic.
4. If the error persists after reinstallation, notify Microsoft Product Support Services.

See also

- Talk to Us

Class does not support Automation or does not support expected interface

4/9/2019 • 2 minutes to read • Edit Online

Either the class you specified in the `GetObject` or `CreateObject` function call has not exposed a programmability interface, or you changed a project from .dll to .exe, or vice versa.

To correct this error

1. Check the documentation of the application that created the object for limitations on the use of automation with this class of object.
2. If you changed a project from .dll to .exe or vice versa, you must manually unregister the old .dll or .exe.

See also

- Error Types
- Talk to Us

'Class' statement must end with a matching 'End Class'

4/28/2019 • 2 minutes to read • Edit Online

`Class` is used to initiate a `Class` block; hence it can only appear at the beginning of the block, with a matching `End Class` statement ending the block. Either you have a redundant `Class` statement, or you have not ended your `Class` block with `End Class`.

Error ID: BC30481

To correct this error

- Locate and remove the unnecessary `Class` statement.
- Conclude the `Class` block with a matching `End Class`.

See also

- End <keyword> Statement
- Class Statement

Clipboard format is not valid

4/28/2019 • 2 minutes to read • Edit Online

The specified Clipboard format is incompatible with the method being executed. Among the possible causes for this error are:

- Using the Clipboard's `GetText` or `SetText` method with a Clipboard format other than `vbCFText` or `vbCFLink`.
- Using the Clipboard's `GetData` or `SetData` method with a Clipboard format other than `vbCFBitmap`, `vbCFDIB`, or `vbCFMetafile`.
- Using the `GetData` or `SetData` methods of a `DataObject` with a Clipboard format in the range reserved by Microsoft Windows for registered formats (&HC000-&HFFFF), when that Clipboard format has not been registered with Microsoft Windows.

To correct this error

- Remove the invalid format and specify a valid one.

See also

- Clipboard: Adding Other Formats

Constant expression not representable in type '<typename>'

4/9/2019 • 2 minutes to read • Edit Online

You are trying to evaluate a constant that will not fit into the target type, usually because it is overflowing the range.

Error ID: BC30439

To correct this error

1. Change the target type to one that can handle the constant.

See also

- Constants Overview
- Constants and Enumerations

Constants must be of an intrinsic or enumerated type, not a class, structure, type parameter, or array type

4/9/2019 • 2 minutes to read • Edit Online

You have attempted to declare a constant as a class, structure, or array type, or as a type parameter defined by a containing generic type.

Constants must be of an intrinsic type (`Boolean` , `Byte` , `Date` , `Decimal` , `Double` , `Integer` , `Long` , `Object` , `SByte` , `Short` , `Single` , `String` , `UInteger` , `ULong` , or `UShort`), or an `Enum` type based on one of the integral types.

Error ID: BC30424

To correct this error

1. Declare the constant as an intrinsic or `Enum` type.
2. A constant can also be a special value such as `True` , `False` , or `Nothing` . The compiler considers these predefined values to be of the appropriate intrinsic type.

See also

- Constants and Enumerations
- Data Types
- Data Types

Constructor '<name>' cannot call itself

4/9/2019 • 2 minutes to read • Edit Online

A `Sub New` procedure in a class or structure calls itself.

The purpose of a constructor is to initialize an instance of a class or structure when it is first created. A class or structure can have several constructors, provided they all have different parameter lists. A constructor is permitted to call another constructor to perform its functionality in addition to its own. But it is meaningless for a constructor to call itself, and in fact it would result in infinite recursion if permitted.

Error ID: BC30298

To correct this error

1. Check the parameter list of the constructor being called. It should be different from that of the constructor making the call.
2. If you do not intend to call a different constructor, remove the `Sub New` call entirely.

See also

- Object Lifetime: How Objects Are Created and Destroyed

Copying the value of 'ByRef' parameter '<parametername>' back to the matching argument narrows from type '<typename1>' to type '<typename2>'

4/28/2019 • 2 minutes to read • Edit Online

A procedure is called with an argument that widens to the corresponding parameter type, and the conversion from the parameter to the argument is narrowing.

When you define a class or structure, you can define one or more conversion operators to convert that class or structure type to other types. You can also define reverse conversion operators to convert those other types back to your class or structure type. When you use your class or structure type in a procedure call, Visual Basic can use these conversion operators to convert the type of an argument to the type of its corresponding parameter.

If you pass the argument ByRef, Visual Basic sometimes copies the argument value into a local variable in the procedure instead of passing a reference. In such a case, when the procedure returns, Visual Basic must then copy the local variable value back into the argument in the calling code.

If a `ByRef` argument value is copied into the procedure and the argument and parameter are of the same type, no conversion is necessary. But if the types are different, Visual Basic must convert in both directions. If one of the types is your class or structure type, Visual Basic must convert it both to and from the other type. If one of these conversions is widening, the reverse conversion might be narrowing.

Error ID: BC32053

To correct this error

- If possible, use a calling argument of the same type as the procedure parameter, so Visual Basic does not need to do any conversion.
- If you need to call the procedure with an argument type different from the parameter type but do not need to return a value into the calling argument, define the parameter to be ByVal instead of `ByRef`.
- If you need to return a value into the calling argument, define the reverse conversion operator as Widening, if possible.

See also

- Procedures
- Procedure Parameters and Arguments
- Passing Arguments by Value and by Reference
- Operator Procedures
- Operator Statement
- How to: Define an Operator
- How to: Define a Conversion Operator
- Type Conversions in Visual Basic
- Widening and Narrowing Conversions

'Custom' modifier is not valid on events declared without explicit delegate types

4/9/2019 • 2 minutes to read • Edit Online

Unlike a non-custom event, a `Custom Event` declaration requires an `As` clause following the event name that explicitly specifies the delegate type for the event.

Non-custom events can be defined either with an `As` clause and an explicit delegate type, or with a parameter list immediately following the event name.

Error ID: BC31122

To correct this error

1. Define a delegate with the same parameter list as the custom event.

   For example, if the `Custom Event` was defined by `Custom Event Test(ByVal sender As Object, ByVal i As Integer)`, then the corresponding delegate would be the following.

   ```
   Delegate Sub TestDelegate(ByVal sender As Object, ByVal i As Integer)
   ```

2. Replace the parameter list of the custom event with an `As` clause specifying the delegate type.

   Continuing with the example, `Custom Event` declaration would be rewritten as follows.

   ```
   Custom Event Test As TestDelegate
   ```

Example

This example declares a `Custom Event` and specifies the required `As` clause with a delegate type.

```
Delegate Sub TestDelegate(ByVal sender As Object, ByVal i As Integer)
Custom Event Test As TestDelegate
    AddHandler(ByVal value As TestDelegate)
        ' Code for adding an event handler goes here.
    End AddHandler

    RemoveHandler(ByVal value As TestDelegate)
        ' Code for removing an event handler goes here.
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal i As Integer)
        ' Code for raising an event goes here.
    End RaiseEvent
End Event
```

See also

- Event Statement

- Delegate Statement
- Events

Data type(s) of the type parameter(s) cannot be inferred from these arguments

8/22/2019 • 2 minutes to read • Edit Online

Data type(s) of the type parameter(s) cannot be inferred from these arguments. Specifying the data type(s) explicitly might correct this error.

This error occurs when overload resolution has failed. It occurs as a subordinate message that states why a particular overload candidate has been eliminated. The error message explains that the compiler cannot use type inference to find data types for the type parameters.

> **NOTE**
> When specifying arguments is not an option (for example, for query operators in query expressions), the error message appears without the second sentence.

The following code demonstrates the error.

```
Module Module1

    Sub Main()

        '' Not Valid.
        'OverloadedGenericMethod("Hello", "World")

    End Sub

    Sub OverloadedGenericMethod(Of T)(ByVal x As String,
                                      ByVal y As InterfaceExample(Of T))
    End Sub

    Sub OverloadedGenericMethod(Of T, R)(ByVal x As T,
                                         ByVal y As InterfaceExample(Of R))
    End Sub

End Module

Interface InterfaceExample(Of T)
End Interface
```

Error ID: BC36647 and BC36644

To correct this error

- You may be able to specify a data type for the type parameter or parameters instead of relying on type inference.

See also

- Relaxed Delegate Conversion
- Generic Procedures in Visual Basic
- Type Conversions in Visual Basic

Declaration expected

4/28/2019 • 2 minutes to read • Edit Online

A nondeclarative statement, such as an assignment or loop statement, occurs outside any procedure. Only declarations are allowed outside procedures.

Alternatively, a programming element is declared without a declaration keyword such as `Dim` or `Const`.

Error ID: BC30188

To correct this error

- Move the nondeclarative statement to the body of a procedure.
- Begin the declaration with an appropriate declaration keyword.
- Ensure that a declaration keyword is not misspelled.

See also

- Procedures
- Dim Statement

Default property '<propertyname1>' conflicts with default property '<propertyname2>' in '<classname>' and so should be declared 'Shadows'

4/28/2019 • 2 minutes to read • Edit Online

A property is declared with the same name as a property defined in the base class. In this situation, the property in this class should shadow the base class property.

This message is a warning. `Shadows` is assumed by default. For more information about hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40007

To correct this error

- Add the `Shadows` keyword to the declaration, or change the name of the property being declared.

See also

- Shadows
- Shadowing in Visual Basic

Default property access is ambiguous between the inherited interface members '<defaultpropertyname>' of interface '<interfacename1>' and '<defaultpropertyname>' of interface '<interfacename2>'

7/26/2019 • 2 minutes to read • Edit Online

An interface inherits from two interfaces, each of which declares a default property with the same name. The compiler cannot resolve an access to this default property without qualification. The following example illustrates this.

```
Public Interface Iface1
    Default Property prop(ByVal arg As Integer) As Integer
End Interface
Public Interface Iface2
    Default Property prop(ByVal arg As Integer) As Integer
End Interface
Public Interface Iface3
    Inherits Iface1, Iface2
End Interface
Public Class testClass
    Public Sub accessDefaultProperty()
        Dim testObj As Iface3
        Dim testInt As Integer = testObj(1)
    End Sub
End Class
```

When you specify `testObj(1)`, the compiler tries to resolve it to the default property. However, there are two possible default properties because of the inherited interfaces, so the compiler signals this error.

Error ID: BC30686

To correct this error

- Avoid inheriting any members with the same name. In the preceding example, if `testObj` does not need any of the members of, say, `Iface2`, then declare it as follows:

  ```
  Dim testObj As Iface1
  ```

  -or-

- Implement the inheriting interface in a class. Then you can implement each of the inherited properties with different names. However, only one of them can be the default property of the implementing class. The following example illustrates this.

```
Public Class useIface3
    Implements Iface3
    Default Public Property prop1(ByVal arg As Integer) As Integer Implements Iface1.prop
        ' Insert code to define Get and Set procedures for prop1.
    End Property
    Public Property prop2(ByVal arg As Integer) As Integer Implements Iface2.prop
        ' Insert code to define Get and Set procedures for prop2.
    End Property
End Class
```

See also

- Interfaces

Delegate class '<classname>' has no Invoke method, so an expression of this type cannot be the target of a method call

4/9/2019 • 2 minutes to read • Edit Online

A call to `Invoke` through a delegate has failed because `Invoke` is not implemented on the delegate class.

Error ID: BC30220

To correct this error

1. Ensure that an instance of the delegate class has been created with a `Dim` statement and that a procedure has been assigned to the delegate instance with the `AddressOf` operator.
2. Locate the code that implements the delegate class and make sure it implements the `Invoke` procedure.

See also

- Delegates
- Delegate Statement
- AddressOf Operator
- Dim Statement

Derived classes cannot raise base class events

4/28/2019 • 2 minutes to read • Edit Online

An event can be raised only from the declaration space in which it is declared. Therefore, a class cannot raise events from any other class, even one from which it is derived.

Error ID: BC30029

To correct this error

- Move the `Event` statement or the `RaiseEvent` statement so they are in the same class.

See also

- Event Statement
- RaiseEvent Statement

Device I/O error

4/28/2019 • 2 minutes to read • Edit Online

An input or output error occurred while your program was using a device such as a printer or disk drive.

To correct this error

- Make sure the device is operating properly, and then retry the operation.

See also

- Error Types

'Dir' function must first be called with a 'PathName' argument

4/9/2019 • 2 minutes to read • Edit Online

An initial call to the `Dir` function does not include the `PathName` argument. The first call to `Dir` must include a `PathName`, but subsequent calls to `Dir` do not need to include parameters to retrieve the next item.

To correct this error

1. Supply a `PathName` argument in the function call.

See also

- Dir

End of statement expected

4/9/2019 • 2 minutes to read • Edit Online

The statement is syntactically complete, but an additional programming element follows the element that completes the statement. A line terminator is required at the end of every statement.

A line terminator divides the characters of a Visual Basic source file into lines. Examples of line terminators are the Unicode carriage return character (&HD), the Unicode linefeed character (&HA), and the Unicode carriage return character followed by the Unicode linefeed character. For more information about line terminators, see the Visual Basic Language Specification.

Error ID: BC30205

To correct this error

1. Check to see if two different statements have inadvertently been put on the same line.
2. Insert a line terminator after the element that completes the statement.

See also

- How to: Break and Combine Statements in Code
- Statements

Error creating assembly manifest: <error message>

4/9/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error in the pre-emission stage of creating the assembly.

This can occur if there are problems with the key file or the key container specified. To fully sign an assembly, you must provide a valid key file that contains information about the public and private keys. To delay sign an assembly, you must select the **Delay sign only** check box and provide a valid key file that contains information about the public key information. The private key is not necessary when an assembly is delay-signed. For more information, see How to: Sign an Assembly with a Strong Name.

Error ID: BC30140

To correct this error

1. Examine the quoted error message and consult the topic Al.exe. for error AL1019 further explanation and advice
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- How to: Sign an Assembly with a Strong Name
- Signing Page, Project Designer
- Al.exe
- Talk to Us

Error creating Win32 resources: <error message>

4/9/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error creating an in-memory resource. This might be a problem with the environment, or your computer might be low on memory.

Error ID: BC30136

To correct this error

1. Examine the quoted error message and consult the topic Al.exe. for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- Al.exe
- Talk to Us

Error in loading DLL (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

A dynamic-link library (DLL) is a library specified in the `Lib` clause of a `Declare` statement. Possible causes for this error include:

- The file is not DLL executable.
- The file is not a Microsoft Windows DLL.
- The DLL references another DLL that is not present.
- The DLL or referenced DLL is not in a directory specified in the path.

To correct this error

- If the file is a source-text file and therefore not DLL executable, it must be compiled and linked to a DLL-executable form.
- If the file is not a Microsoft Windows DLL, obtain the Microsoft Windows equivalent.
- If the DLL references another DLL that is not present, obtain the referenced DLL and make it available.
- If the DLL or referenced DLL is not in a directory specified by the path, move the DLL to a referenced directory.

See also

- Declare Statement

Error saving temporary Win32 resource file '<filename>': <error message>

4/9/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker reported an error obtaining a file name for use in writing an in-memory resource.

Error ID: BC30137

To correct this error

1. Examine the quoted error message and consult the topic Al.exe. for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- Al.exe
- Talk to Us

Errors occurred while compiling the XML schemas in the project

4/28/2019 • 2 minutes to read • Edit Online

Errors occurred while compiling the XML schemas in the project. Because of this, XML IntelliSense is not available.

There is an error in an XML Schema Definition (XSD) schema included in the project. This error occurs when you add an XSD schema (.xsd) file that conflicts with the existing XSD schema set for the project.

Error ID: BC36810

To correct this error

- Double-click the warning in the **Errors List** window. Visual Basic will take you to the location in the XSD file that is the source of the warning. Correct the error in the XSD schema.
- Ensure that all required XSD schema (.xsd) files are included in the project. You may need to click **Show All Files** on the **Project** menu to see your .xsd files in **Solution Explorer**. Right-click an .xsd file and then click **Include In Project** to include the file in your project.
- If you are using the XML to Schema Wizard, this error can occur if you infer schemas more than one time from the same source. In this case, you can remove the existing XSD schema files from the project, add a new XML to Schema item template, and then provide the XML to Schema Wizard with all the applicable XML sources for your project.
- If no error is identified in your XSD schema, the XML compiler may not have enough information to provide a detailed error message. You may be able to get more detailed error information if you ensure that the XML namespaces for the .xsd files included in your project match the XML namespaces identified for the XML Schema set in Visual Studio.

See also

- Error List Window
- XML

Evaluation of expression or statement timed out

4/9/2019 • 2 minutes to read • Edit Online

The evaluation of an expression did not complete in a timely manner.

Error ID: BC30722

To correct this error

1. Verify that the entered code is correct.
2. Simplify your expression so that it takes less time to execute.

See also

- Debugging in Visual Studio

Event '<eventname1>' cannot implement event '<eventname2>' on interface '<interface>' because their delegate types '<delegate1>' and '<delegate2>' do not match

4/28/2019 • 2 minutes to read • Edit Online

Visual Basic cannot implement an event because the delegate type of the event does not match the delegate type of the event in the interface. This error can occur when you define multiple events in an interface and then attempt to implement them together with the same event. An event can implement two or more events only if all implemented events are declared using the `As` syntax and specify the same delegate type.

Error ID: BC31423

To correct this error

- Implement the events separately.

  —or—

- Define the events in the interface using the `As` syntax and specify the same delegate type.

See also

- Event Statement
- Delegate Statement
- Events

Events cannot be declared with a delegate type that has a return type

4/28/2019 • 2 minutes to read • Edit Online

A delegate was specified for a function procedure.

Error ID: BC31084

To correct this error

- Specify a delegate for a `Sub` procedure.

See also

- Events

Events of shared WithEvents variables cannot be handled by non-shared methods

4/28/2019 • 2 minutes to read • Edit Online

A variable declared with the `Shared` modifier is a shared variable. A shared variable identifies exactly one storage location. A variable declared with the `WithEvents` modifier asserts that the type to which the variable belongs handles the set of events the variable raises. When a value is assigned to the variable, the property created by the `WithEvents` declaration unhooks any existing event handler and hooks up the new event handler via the `Add` method.

Error ID: BC30594

To correct this error

- Declare your event handler `Shared`.

See also

- Shared
- WithEvents

Expression does not produce a value

4/28/2019 • 2 minutes to read • Edit Online

You have tried to use an expression that does not produce a value in a value-producing context, such as calling a `Sub` in a context where a `Function` is expected.

Error ID: BC30491

To correct this error

- Change the expression to one that produces a value.

See also

- Error Types

Expression has the type '<typename>' which is a restricted type and cannot be used to access members inherited from 'Object' or 'ValueType'

4/9/2019 • 2 minutes to read • Edit Online

An expression evaluates to a type that cannot be boxed by the common language runtime (CLR) but accesses a member that requires boxing.

Boxing refers to the processing necessary to convert a type to `Object` or, on occasion, to ValueType. The common language runtime cannot box certain structure types, for example ArgIterator, RuntimeArgumentHandle, and TypedReference.

This expression attempts to use the restricted type to call a method inherited from Object or ValueType, such as GetHashCode or ToString. To access this method, Visual Basic has attempted an implicit boxing conversion that causes this error.

Error ID: BC31393

To correct this error

1. Locate the expression that evaluates to the cited type.
2. Locate the part of your statement that attempts to call the method inherited from Object or ValueType.
3. Rewrite the statement to avoid the method call.

See also

- Implicit and Explicit Conversions

Expression is a value and therefore cannot be the target of an assignment

7/26/2019 • 2 minutes to read • Edit Online

A statement attempts to assign a value to an expression. You can assign a value only to a writable variable, property, or array element at run time. The following example illustrates how this error can occur.

```
Dim yesterday As Integer
ReadOnly maximum As Integer = 45
yesterday + 1 = DatePart(DateInterval.Day, Now)
' The preceding line is an ERROR because of an expression on the left.
maximum = 50
' The preceding line is an ERROR because maximum is declared ReadOnly.
```

Similar examples could apply to properties and array elements.

Indirect Access. Indirect access through a value type can also generate this error. Consider the following code example, which attempts to set the value of Point by accessing it indirectly through Location.

```
' Assume this code runs inside Form1.
Dim exitButton As New System.Windows.Forms.Button()
exitButton.Text = "Exit this form"
exitButton.Location.X = 140
' The preceding line is an ERROR because of no storage for Location.
```

The last statement of the preceding example fails because it creates only a temporary allocation for the Point structure returned by the Location property. A structure is a value type, and the temporary structure is not retained after the statement runs. The problem is resolved by declaring and using a variable for Location, which creates a more permanent allocation for the Point structure. The following example shows code that can replace the last statement of the preceding example.

```
Dim exitLocation as New System.Drawing.Point(140, exitButton.Location.Y)
exitButton.Location = exitLocation
```

Error ID: BC30068

To correct this error

- If the statement assigns a value to an expression, replace the expression with a single writable variable, property, or array element.
- If the statement makes indirect access through a value type (usually a structure), create a variable to hold the value type.
- Assign the appropriate structure (or other value type) to the variable.
- Use the variable to access the property to assign it a value.

See also

- Operators and Expressions

- Statements
- Troubleshooting Procedures

Expression of type <type> is not queryable

4/9/2019 • 2 minutes to read • Edit Online

Expression of type <type> is not queryable. Make sure you are not missing an assembly reference and/or namespace import for the LINQ provider.

Queryable types are defined in the System.Linq, System.Data.Linq, and System.Xml.Linq namespaces. You must import one or more of these namespaces to perform LINQ queries.

The System.Linq namespace enables you to query objects such as collections and arrays by using LINQ.

The System.Data.Linq namespace enables you to query ADO.NET Datasets and SQL Server databases by using LINQ.

The System.Xml.Linq namespace enables you to query XML by using LINQ and to use XML features in Visual Basic.

Error ID: BC36593

To correct this error

1. Add an `Import` statement for the System.Linq, System.Data.Linq, or System.Xml.Linq namespace to your code file. You can also import namespaces for your project by using the **References** page of the Project Designer (**My Project**).

2. Ensure that the type that you have identified as the source of your query is a queryable type. That is, a type that implements IEnumerable<T> or IQueryable<T>.

See also

- System.Linq
- System.Data.Linq
- System.Xml.Linq
- Introduction to LINQ in Visual Basic
- LINQ
- XML
- References and the Imports Statement
- Imports Statement (.NET Namespace and Type)
- References Page, Project Designer (Visual Basic)

Expression recursively calls the containing property '<propertyname>'

4/28/2019 • 2 minutes to read • Edit Online

A statement in the `Set` procedure of a property definition stores a value into the name of the property.

The recommended approach to holding the value of a property is to define a `Private` variable in the property's container and use it in both the `Get` and `Set` procedures. The `Set` procedure should then store the incoming value in this `Private` variable.

The `Get` procedure behaves like a `Function` procedure, so it can assign a value to the property name and return control by encountering the `End Get` statement. The recommended approach, however, is to include the `Private` variable as the value in a Return Statement.

The `Set` procedure behaves like a `Sub` procedure, which does not return a value. Therefore, the procedure or property name has no special meaning within a `Set` procedure, and you cannot store a value into it.

The following example illustrates the approach that can cause this error, followed by the recommended approach.

```
Public Class illustrateProperties
' The code in the following property causes this error.
    Public Property badProp() As Char
        Get
            Dim charValue As Char
            ' Insert code to update charValue.
            badProp = charValue
        End Get
        Set(ByVal Value As Char)
            ' The following statement causes this error.
            badProp = Value
            ' The value stored in the local variable badProp
            ' is not used by the Get procedure in this property.
        End Set
    End Property
' The following code uses the recommended approach.
    Private propValue As Char
    Public Property goodProp() As Char
        Get
            ' Insert code to update propValue.
            Return propValue
        End Get
        Set(ByVal Value As Char)
            propValue = Value
        End Set
    End Property
End Class
```

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, please see Configuring Warnings in Visual Basic.

Error ID: BC42026

To correct this error

- Rewrite the property definition to use the recommended approach as illustrated in the preceding example.

See also

- Property Procedures
- Property Statement
- Set Statement

Expression too complex

4/28/2019 • 2 minutes to read • Edit Online

A floating-point expression contains too many nested subexpressions.

To correct this error

- Break the expression into as many separate expressions as necessary to prevent the error from occurring.

See also

- Operators and Expressions

'Extension' attribute can be applied only to 'Module', 'Sub', or 'Function' declarations

4/28/2019 • 2 minutes to read • Edit Online

The only way to extend a data type in Visual Basic is to define an extension method inside a standard module. The extension method can be a `Sub` procedure or a `Function` procedure. All extension methods must be marked with the extension attribute, `<Extension()>`, from the System.Runtime.CompilerServices namespace. Optionally, a module that contains an extension method may be marked in the same way. No other use of the extension attribute is valid.

Error ID: BC36550

To correct this error

- Remove the extension attribute.
- Redesign your extension as a method, defined in an enclosing module.

Example

The following example defines a `Print` method for the `String` data type.

```
Imports StringUtility
Imports System.Runtime.CompilerServices
Namespace StringUtility
    <Extension()>
    Module StringExtensions
        <Extension()>
        Public Sub Print (ByVal str As String)
            Console.WriteLine(str)
        End Sub
    End Module
End Namespace
```

See also

- Attributes overview
- Extension Methods
- Module Statement

File already open

4/28/2019 • 2 minutes to read • Edit Online

Sometimes a file must be closed before another `FileOpen` or other operation can occur. Among the possible causes of this error are:

- A sequential output mode `FileOpen` operation was executed for a file that is already open
- A statement refers to an open file.

To correct this error

1. Close the file before executing the statement.

See also

- FileOpen

File is too large to read into a byte array

4/28/2019 • 2 minutes to read • Edit Online

The size of the file you are attempting to read into a byte array exceeds 4 GB. The `My.Computer.FileSystem.ReadAllBytes` method cannot read a file that exceeds this size.

To correct this error

- Use a StreamReader to read the file. For more information, see Basics of .NET Framework File I/O and the File System (Visual Basic).

See also

- ReadAllBytes
- StreamReader
- File Access with Visual Basic
- How to: Read Text from Files with a StreamReader

File name or class name not found during Automation operation (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

The name specified for file name or class in a call to the `GetObject` function could not be found.

To correct this error

- Check the names and try again. Make sure the name used for the `class` parameter matches that registered with the system.

See also

- Error Types

File not found (Visual Basic Run-Time Error)

4/28/2019 • 2 minutes to read • Edit Online

The file was not found where specified. The error has the following possible causes:

- A statement refers to a file that does not exist.
- An attempt was made to call a procedure in a dynamic-link library (DLL), but the library specified in the `Lib` clause of the `Declare` statement cannot be found.
- You attempted to open a project or load a text file that does not exist.

To correct this error

1. Check the spelling of the file name and the path specification.

See also

- Declare Statement

First operand in a binary 'If' expression must be nullable or a reference type

4/28/2019 • 2 minutes to read • Edit Online

An `If` expression can take either two or three arguments. When you send only two arguments, the first argument must be a reference type or a nullable type. If the first argument evaluates to anything other than `Nothing`, its value is returned. If the first argument evaluates to `Nothing`, the second argument is evaluated and returned.

For example, the following code contains two `If` expressions, one with three arguments and one with two arguments. The expressions calculate and return the same value.

```
' firstChoice is a nullable value type.
Dim firstChoice? As Integer = Nothing
Dim secondChoice As Integer = 1128
' If expression with three arguments.
Console.WriteLine(If(firstChoice IsNot Nothing, firstChoice, secondChoice))
' If expression with two arguments.
Console.WriteLine(If(firstChoice, secondChoice))
```

The following expressions cause this error:

```
Dim choice1 = 4
Dim choice2 = 5
Dim booleanVar = True

' Not valid.
'Console.WriteLine(If(choice1 < choice2, 1))
' Not valid.
'Console.WriteLine(If(booleanVar, "Test returns True."))
```

Error ID: BC33107

To correct this error

- If you cannot change the code so that the first argument is a nullable type or reference type, consider converting to a three-argument `If` expression, or to an `If...Then...Else` statement.

```
Console.WriteLine(If(choice1 < choice2, 1, 2))
Console.WriteLine(If(booleanVar, "Test returns True.", "Test returns False."))
```

See also

- If Operator
- If...Then...Else Statement
- Nullable Value Types

First statement of this 'Sub New' must be a call to 'MyBase.New' or 'MyClass.New' (No Accessible Constructor Without Parameters)

4/28/2019 • 2 minutes to read • Edit Online

First statement of this 'Sub New' must be a call to 'MyBase.New' or 'MyClass.New' because base class '<basename>' of '<derivedname>' does not have an accessible 'Sub New' that can be called with no arguments.

In a derived class, every constructor must call a base class constructor (`MyBase.New`). If the base class has a constructor with no parameters that is accessible to derived classes, `MyBase.New` can be called automatically. If not, a base class constructor must be called with parameters, and this cannot be done automatically. In this case, the first statement of every derived class constructor must call a parameterized constructor on the base class, or call another constructor in the derived class that makes a base class constructor call.

Error ID: BC30148

To correct this error

- Either call `MyBase.New` supplying the required parameters, or call a peer constructor that makes such a call.

  For example, if the base class has a constructor that's declared as `Public Sub New(ByVal index as Integer)` , the first statement in the derived class constructor might be `MyBase.New(100)` .

See also

- Inheritance Basics

First statement of this 'Sub New' must be an explicit call to 'MyBase.New' or 'MyClass.New' because the '<constructorname>' in the base class '<baseclassname>' of '<derivedclassname>' is marked obsolete: '<errormessage>'

4/9/2019 • 2 minutes to read • Edit Online

A class constructor does not explicitly call a base class constructor, and the implicit base class constructor is marked with the ObsoleteAttribute attribute and the directive to treat it as an error.

When a derived class constructor does not call a base class constructor, Visual Basic attempts to generate an implicit call to a parameterless base class constructor. If there is no accessible constructor in the base class that can be called without arguments, Visual Basic cannot generate an implicit call. In this case, the required constructor is marked with the ObsoleteAttribute attribute, so Visual Basic cannot call it.

You can mark any programming element as being no longer in use by applying ObsoleteAttribute to it. If you do this, you can set the attribute's IsError property to either `True` or `False`. If you set it to `True`, the compiler treats an attempt to use the element as an error. If you set it to `False`, or let it default to `False`, the compiler issues a warning if there is an attempt to use the element.

Error ID: BC30920

To correct this error

1. Examine the quoted error message and take appropriate action.
2. Include a call to `MyBase.New()` or `MyClass.New()` as the first statement of the `Sub New` in the derived class.

See also

- Attributes overview

'For Each' on type '<typename>' is ambiguous because the type implements multiple instantiations of 'System.Collections.Generic.IEnumerable(Of T)'

5/14/2019 • 2 minutes to read • Edit Online

A `For Each` statement specifies an iterator variable that has more than one GetEnumerator method.

The iterator variable must be of a type that implements the System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> interface in one of the `Collections` namespaces of the .NET Framework. It is possible for a class to implement more than one constructed generic interface, using a different type argument for each construction. If a class that does this is used for the iterator variable, that variable has more than one GetEnumerator method. In such a case, Visual Basic cannot choose which method to call.

Error ID: BC32096

To correct this error

- Use DirectCast Operator or TryCast Operator to cast the iterator variable type to the interface defining the GetEnumerator method you want to use.

See also

- For Each...Next Statement
- Interfaces

Friend assembly reference <reference> is invalid

4/9/2019 • 2 minutes to read • Edit Online

Friend assembly reference <reference> is invalid. Strong-name signed assemblies must specify a public key in their InternalsVisibleTo declarations.

The assembly name passed to the InternalsVisibleToAttribute attribute constructor identifies a strong-named assembly, but it does not include a `PublicKey` attribute.

Error ID: BC31535

To correct this error

1. Determine the public key for the strong-named friend assembly. Include the public key as part of the assembly name passed to the InternalsVisibleToAttribute attribute constructor by using the `PublicKey` attribute.

See also

- AssemblyName
- Friend Assemblies

Function '<procedurename>' doesn't return a value on all code paths

4/28/2019 • 2 minutes to read • Edit Online

Function '<procedurename>' doesn't return a value on all code paths. Are you missing a 'Return' statement?

A `Function` procedure has at least one possible path through its code that does not return a value.

You can return a value from a `Function` procedure in any of the following ways:

- Include the value in a Return Statement.
- Assign the value to the `Function` procedure name and then perform an `Exit Function` statement.
- Assign the value to the `Function` procedure name and then perform the `End Function` statement.

If control passes to `Exit Function` or `End Function` and you have not assigned any value to the procedure name, the procedure returns the default value of the return data type. For more information, see "Behavior" in Function Statement.

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42105

To correct this error

- Check your control flow logic and make sure you assign a value before every statement that causes a return.

  It is easier to guarantee that every return from the procedure returns a value if you always use the `Return` statement. If you do this, the last statement before `End Function` should be a `Return` statement.

See also

- Function Procedures
- Function Statement
- Compile Page, Project Designer (Visual Basic)

Function evaluation is disabled because a previous function evaluation timed out

4/9/2019 • 2 minutes to read • Edit Online

Function evaluation is disabled because a previous function evaluation timed out. To re-enable function evaluation, step again or restart debugging.

In the Visual Studio debugger, an expression specifies a procedure call, but another evaluation has timed out.

Possible causes for a procedure call to time out include an infinite loop or *endless loop*. For more information, see For...Next Statement.

A special case of an infinite loop is *recursion*. For more information, see Recursive Procedures.

Error ID: BC30957

To correct this error

1. If possible, determine what the previous function evaluation was and what caused it to time out. Otherwise, you might encounter this error again.
2. Either step the debugger again, or terminate and restart debugging.

See also

- Debugging in Visual Studio
- Navigating through Code with the Debugger

Generic parameters used as optional parameter types must be class constrained

4/28/2019 • 2 minutes to read • Edit Online

A procedure is declared with an optional parameter that uses a type parameter that is not constrained to be a reference type.

You must always supply a default value for each optional parameter. If the parameter is of a reference type, the optional value must be `Nothing`, which is a valid value for any reference type. However, if the parameter is of a value type, that type must be an elementary data type predefined by Visual Basic. This is because a composite value type, such as a user-defined structure, has no valid default value.

When you use a type parameter for an optional parameter, you must guarantee that it is of a reference type to avoid the possibility of a value type with no valid default value. This means you must constrain the type parameter either with the `Class` keyword or with the name of a specific class.

Error ID: BC32124

To correct this error

- Constrain the type parameter to accept only a reference type, or do not use it for the optional parameter.

See also

- Generic Types in Visual Basic
- Type List
- Class Statement
- Optional Parameters
- Structures
- Nothing

'Get' accessor of property '<propertyname>' is not accessible

4/28/2019 • 2 minutes to read • Edit Online

A statement attempts to retrieve the value of a property when it does not have access to the property's `Get` procedure.

If the Get Statement is marked with a more restrictive access level than its Property Statement, an attempt to read the property value could fail in the following cases:

- The `Get` statement is marked Private and the calling code is outside the class or structure in which the property is defined.
- The `Get` statement is marked Protected and the calling code is not in the class or structure in which the property is defined, nor in a derived class.
- The `Get` statement is marked Friend and the calling code is not in the same assembly in which the property is defined.

Error ID: BC31103

To correct this error

- If you have control of the source code defining the property, consider declaring the `Get` procedure with the same access level as the property itself.
- If you do not have control of the source code defining the property, or you must restrict the `Get` procedure access level more than the property itself, try to move the statement that reads the property value to a region of code that has better access to the property.

See also

- Property Procedures
- How to: Declare a Property with Mixed Access Levels

Handles clause requires a WithEvents variable defined in the containing type or one of its base types

7/26/2019 • 2 minutes to read • Edit Online

You did not supply a `WithEvents` variable in your `Handles` clause. The `Handles` keyword at the end of a procedure declaration causes it to handle events raised by an object variable declared using the `WithEvents` keyword.

Error ID: BC30506

To correct this error

- Supply the necessary `WithEvents` variable.

Example

In the following example, Visual Basic generates compiler error `BC30506` because the WithEvents keyword is not used in the definition of the System.Timers.Timer instance.

```
Imports System.Timers

Module Module1
    Private _timer1 As New Timer() With {.Interval = 1000, .Enabled = True}

    Sub Main()
        Console.WriteLine("Press any key to start the timer...")
        Console.ReadKey()
        _timer1.Start()
        Console.ReadKey()
    End Sub

    Private Sub Timer1_Tick(sender As Object, args As EventArgs) Handles _timer1.Elapsed
        Console.WriteLine("Press any key to terminate...")
    End Sub
End Module
```

The following example compiles successfully because the `_timer1` variable is defined with the `WithEvents` keyword:

```
Imports System.Timers

Module Module1
    Private WithEvents _timer1 As New Timer() With {.Interval = 1000}

    Sub Main()
        Console.WriteLine("Press any key to start the timer...")
        Console.ReadKey()
        _timer1.Start()
        Console.ReadKey()
    End Sub

    Private Sub Timer1_Tick(sender As Object, args As EventArgs) Handles _timer1.Elapsed
        Console.WriteLine("Press any key to terminate...")
    End Sub

End Module
```

See also

- Handles

Identifier expected

4/28/2019 • 2 minutes to read • Edit Online

A programming element that is not a recognizable declared element name occurs where the context requires an element name. One possible cause is that an attribute has been specified somewhere other than at the beginning of the statement.

Error ID: BC30203

To correct this error

- Verify that any attributes in the statement are all placed at the beginning.
- Verify that all element names in the statement are spelled correctly.

See also

- Declared Element Names
- Attributes overview

Identifier is too long

4/28/2019 • 2 minutes to read • Edit Online

The name, or identifier, of every programming element is limited to 1023 characters. In addition, a fully qualified name cannot exceed 1023 characters. This means that the entire identifier string (`<namespace>.<...>.<namespace>.<class>.<element>`) cannot be more than 1023 characters long, including the member-access operator (`.`) characters.

Error ID: BC30033

To correct this error

- Reduce the length of the identifier.

See also

- Declared Element Names

Initializer expected

4/9/2019 • 2 minutes to read • Edit Online

You have tried to declare an instance of a class by using an object initializer in which the initialization list is empty, as shown in the following example.

```
' Not valid.

' Dim aStudent As New Student With {}
```

At least one field or property must be initialized in the initializer list, as shown in the following example.

```
Dim aStudent As New Student With {.year = "Senior"}
```

Error ID: BC30996

To correct this error

1. Initialize at least one field or property in the initializer, or do not use an object initializer.

See also

- Object Initializers: Named and Anonymous Types
- How to: Declare an Object by Using an Object Initializer

Input past end of file

4/9/2019 • 2 minutes to read • Edit Online

Either an `Input` statement is reading from a file that is empty or one in which all the data is used, or you used the `EOF` function with a file opened for binary access.

To correct this error

1. Use the `EOF` function immediately before the `Input` statement to detect the end of the file.
2. If the file is opened for binary access, use `Seek` and `Loc`.

See also

- Input
- EOF
- Seek
- Loc

Internal error happened at <location>

4/28/2019 • 2 minutes to read • Edit Online

An internal error has occurred. The line at which it occurred is contained in the error message.

To correct this error

- Make sure this error was not generated by the `Error` statement or `Raise` method; if it was not, contact Microsoft Product Support Services to report the conditions under which the message appeared.

See also

- Debugger Basics

Implicit conversion from '<typename1>' to '<typename2>' in copying the value of 'ByRef' parameter '<parametername>' back to the matching argument.

4/28/2019 • 2 minutes to read • Edit Online

A procedure is called with a ByRef argument of a different type than that of its corresponding parameter.

If you pass an argument `ByRef`, Visual Basic sometimes copies the argument value into a local variable in the procedure instead of passing a reference. In such a case, when the procedure returns, Visual Basic must then copy the local variable value back into the argument in the calling code.

If a `ByRef` argument value is copied into the procedure and the argument and parameter are of the same type, no conversion is necessary. But if the types are different, Visual Basic must convert in both directions. Because you cannot use `CType` or any of the other conversion keywords on a procedure argument or parameter, such a conversion is always implicit.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC41999

To correct this error

- If possible, use a calling argument of the same type as the procedure parameter, so Visual Basic does not need to do any conversion.
- If you need to call the procedure with an argument type different from the parameter type but do not need to return a value into the calling argument, define the parameter to be ByVal instead of `ByRef`.

See also

- Procedures
- Procedure Parameters and Arguments
- Passing Arguments by Value and by Reference
- Implicit and Explicit Conversions

'Is' requires operands that have reference types, but this operand has the value type '<typename>'

4/28/2019 • 2 minutes to read • Edit Online

The `Is` comparison operator determines whether two object variables refer to the same instance. This comparison is not defined for value types.

Error ID: BC30020

To correct this error

- Use the appropriate arithmetic comparison operator or the `Like` operator to compare two value types.

See also

- Is Operator
- Like Operator
- Comparison Operators

'IsNot' operand of type 'typename' can only be compared to 'Nothing', because 'typename' is a nullable type

4/9/2019 • 2 minutes to read • Edit Online

A variable declared as nullable has been compared to an expression other than `Nothing` using the `IsNot` operator.

Error ID: BC32128

To correct this error

1. To compare a nullable type to an expression other than `Nothing` by using the `IsNot` operator, call the `GetType` method on the nullable type and compare the result to the expression, as shown in the following example.

```
Dim number? As Integer = 5

If number IsNot Nothing Then
  If number.GetType() IsNot Type.GetType("System.Int32") Then

  End If
End If
```

See also

- Nullable Value Types
- IsNot Operator

Labels that are numbers must be followed by colons

4/28/2019 • 2 minutes to read • Edit Online

Line numbers follow the same rules as other kinds of labels, and must contain a colon.

Error ID: BC30801

To correct this error

- Place the number followed by a colon at the start of a line of code; for example:

```
400:    X += 1
```

See also

- GoTo Statement

Lambda expression will not be removed from this event handler

4/28/2019 • 2 minutes to read • Edit Online

Lambda expression will not be removed from this event handler. Assign the lambda expression to a variable and use the variable to add and remove the event.

When lambda expressions are used with event handlers, you may not see the behavior you expect. The compiler generates a new method for each lambda expression definition, even if they are identical. Therefore, the following code displays `False`.

```
Module Module1

    Sub Main()
        Dim fun1 As ChangeInteger = Function(p As Integer) p + 1
        Dim fun2 As ChangeInteger = Function(p As Integer) p + 1
        Console.WriteLine(fun1 = fun2)
    End Sub

    Delegate Function ChangeInteger(ByVal x As Integer) As Integer

End Module
```

When lambda expressions are used with event handlers, this may cause unexpected results. In the following example, the lambda expression added by `AddHandler` is not removed by the `RemoveHandler` statement.

```
Module Module1

    Event ProcessInteger(ByVal x As Integer)

    Sub Main()

        ' The following line adds one listener to the event.
        AddHandler ProcessInteger, Function(m As Integer) m

        ' The following statement searches the current listeners
        ' for a match to remove. However, this lambda is not the same
        ' as the previous one, so nothing is removed.
        RemoveHandler ProcessInteger, Function(m As Integer) m

    End Sub
End Module
```

By default, this message is a warning. For more information about how to hide warnings or treat warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42326

To correct this error

- To avoid the warning and remove the lambda expression, assign the lambda expression to a variable and use the variable in both the `AddHandler` and `RemoveHandler` statements, as shown in the following example.

```
Module Module1

    Event ProcessInteger(ByVal x As Integer)

    Dim PrintHandler As ProcessIntegerEventHandler

    Sub Main()

        ' Assign the lambda expression to a variable.
        PrintHandler = Function(m As Integer) m

        ' Use the variable to add the listener.
        AddHandler ProcessInteger, PrintHandler

        ' Use the variable again when you want to remove the listener.
        RemoveHandler ProcessInteger, PrintHandler

    End Sub
End Module
```

See also

- Lambda Expressions
- Relaxed Delegate Conversion
- Events

Lambda expressions are not valid in the first expression of a 'Select Case' statement

4/28/2019 • 2 minutes to read • Edit Online

You cannot use a lambda expression for the test expression in a `Select Case` statement. Lambda expression definitions return functions, and the test expression of a `Select Case` statement must be an elementary data type.

The following code causes this error:

```
' Select Case (Function(arg) arg Is Nothing)
    ' List of the cases.
' End Select
```

Error ID: BC36635

To correct this error

- Examine your code to determine whether a different conditional construction, such as an `If...Then...Else` statement, would work for you.
- You may have intended to call the function, as shown in the following code:

```
Dim num? As Integer
Select Case ((Function(arg? As Integer) arg Is Nothing)(num))
    ' List of the cases
End Select
```

See also

- Lambda Expressions
- If...Then...Else Statement
- Select...Case Statement

Late bound resolution; runtime errors could occur

4/28/2019 • 2 minutes to read • Edit Online

An object is assigned to a variable declared to be of the Object Data Type.

When you declare a variable as `Object`, the compiler must perform *late binding*, which causes extra operations at run time. It also exposes your application to potential run-time errors. For example, if you assign a Form to the `Object` variable and then try to access the XmlDocument.NameTable property, the runtime throws a MemberAccessException because the Form class does not expose a `NameTable` property.

If you declare the variable to be of a specific type, the compiler can perform *early binding* at compile time. This results in improved performance, controlled access to the members of the specific type, and better readability of your code.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42017

To correct this error

- If possible, declare the variable to be of a specific type.

See also

- Early and Late Binding
- Object Variable Declaration

Latebound overload resolution cannot be applied to '<procedurename>' because the accessing instance is an interface type

7/26/2019 • 2 minutes to read • Edit Online

The compiler is attempting to resolve a reference to an overloaded property or procedure, but the reference fails because an argument is of type `Object` and the referring object has the data type of an interface. The `Object` argument forces the compiler to resolve the reference as late-bound.

In these circumstances, the compiler resolves the overload through the implementing class instead of through the underlying interface. If the class renames one of the overloaded versions, the compiler does not consider that version to be an overload because its name is different. This in turn causes the compiler to ignore the renamed version when it might have been the correct choice to resolve the reference.

Error ID: BC30933

To correct this error

- Use `CType` to cast the argument from `Object` to the type specified by the signature of the overload you want to call.

  Note that it does not help to cast the referring object to the underlying interface. You must cast the argument to avoid this error.

Example

The following example shows a call to an overloaded `Sub` procedure that causes this error at compile time.

```
Module m1
    Interface i1
        Sub s1(ByVal p1 As Integer)
        Sub s1(ByVal p1 As Double)
    End Interface
    Class c1
        Implements i1
        Public Overloads Sub s1(ByVal p1 As Integer) Implements i1.s1
        End Sub
        Public Overloads Sub s2(ByVal p1 As Double) Implements i1.s1
        End Sub
    End Class
    Sub Main()
        Dim refer As i1 = New c1
        Dim o1 As Object = 3.1415
        ' The following reference is INVALID and causes a compiler error.
        refer.s1(o1)
    End Sub
End Module
```

In the preceding example, if the compiler allowed the call to `s1` as written, the resolution would take place through the class `c1` instead of the interface `i1`. This would mean that the compiler would not consider `s2` because its name is different in `c1`, even though it is the correct choice as defined by `i1`.

You can correct the error by changing the call to either of the following lines of code:

```
refer.s1(CType(o1, Integer))
refer.s1(CType(o1, Double))
```

Each of the preceding lines of code explicitly casts the `Object` variable `o1` to one of the parameter types defined for the overloads.

See also

- Procedure Overloading
- Overload Resolution
- CType Function

Leading '.' or '!' can only appear inside a 'With' statement

4/9/2019 • 2 minutes to read • Edit Online

A period (.) or exclamation point (!) that is not inside a `With` block occurs without an expression on the left. Member access (`.`) and dictionary member access (`!`) require an expression specifying the element that contains the member. This must appear immediately to the left of the accessor or as the target of a `With` block containing the member access.

Error ID: BC30157

To correct this error

1. Ensure that the `With` block is correctly formatted.
2. If there is no `With` block, add an expression to the left of the accessor that evaluates to a defined element containing the member.

See also

- Special Characters in Code
- With...End With Statement

Line is too long

4/28/2019 • 2 minutes to read • Edit Online

Source text lines cannot exceed 65535 characters.

Error ID: BC30494

To correct this error

- Shorten the length of the line to 65535 characters or fewer.

See also

- Error Types

'Line' statements are no longer supported (Visual Basic Compiler Error)

4/9/2019 • 2 minutes to read • Edit Online

Line statements are no longer supported. File I/O functionality is available as `Microsoft.VisualBasic.FileSystem.LineInput` and graphics functionality is available as `System.Drawing.Graphics.DrawLine`.

Error ID: BC30830

To correct this error

1. If performing file access, use `Microsoft.VisualBasic.FileSystem.LineInput`.
2. If performing graphics, use `System.Drawing.Graphics.Drawline`.

See also

- System.IO
- System.Drawing
- File Access with Visual Basic

Method does not have a signature compatible with the delegate

4/2/2019 • 2 minutes to read • Edit Online

There is an incompatibility between the signatures of the method and the delegate you are trying to use. The `Delegate` statement defines the parameter types and return types of a delegate class. Any procedure that has matching parameters of compatible types and return types can be used to create an instance of this delegate type.

Error ID: BC36563

See also

- AddressOf Operator
- Delegate Statement
- Overload Resolution
- Generic Types in Visual Basic

Methods of 'System.Nullable(Of T)' cannot be used as operands of the 'AddressOf' operator

4/28/2019 • 2 minutes to read • Edit Online

A statement uses the `AddressOf` operator with an operand that represents a procedure of the Nullable<T> structure.

Error ID: BC32126

To correct this error

- Replace the procedure name in the `AddressOf` clause with an operand that is not a member of Nullable<T>.
- Write a class that wraps the method of Nullable<T> that you want to use. In the following example, the `NullableWrapper` class defines a new method named `GetValueOrDefault`. Because this new method is not a member of Nullable<T>, it can be applied to `nullInstance`, an instance of a nullable type, to form an argument for `AddressOf`.

```
Module Module1

    Delegate Function Deleg() As Integer

    Sub Main()
        Dim nullInstance As New Nullable(Of Integer)(1)

        Dim del As Deleg

        ' GetValueOrDefault is a method of the Nullable generic
        ' type. It cannot be used as an operand of AddressOf.
        ' del = AddressOf nullInstance.GetValueOrDefault

        ' The following line uses the GetValueOrDefault method
        ' defined in the NullableWrapper class.
        del = AddressOf (New NullableWrapper(
            Of Integer)(nullInstance)).GetValueOrDefault

        Console.WriteLine(del.Invoke())
    End Sub

    Class NullableWrapper(Of T As Structure)
        Private m_Value As Nullable(Of T)

        Sub New(ByVal Value As Nullable(Of T))
            m_Value = Value
        End Sub

        Public Function GetValueOrDefault() As T
            Return m_Value.Value
        End Function
    End Class
End Module
```

See also

- Nullable<T>

- AddressOf Operator
- Nullable Value Types
- Generic Types in Visual Basic

'Module' statements can occur only at file or namespace level

4/28/2019 • 2 minutes to read • Edit Online

`Module` statements must appear at the top of your source file immediately after `Option` and `Imports` statements, global attributes, and namespace declarations, but before all other declarations.

Error ID: BC30617

To correct this error

- Move the `Module` statement to the top of your namespace declaration or source file.

See also

- Module Statement

Name <membername> is not CLS-compliant

4/28/2019 • 2 minutes to read • Edit Online

An assembly is marked as `<CLSCompliant(True)>` but exposes a member with a name that begins with an underscore (`_`).

A programming element can contain one or more underscores, but to be compliant with the Language Independence and Language-Independent Components (CLS), it must not begin with an underscore. See Declared Element Names.

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40031

To correct this error

- If you have control over the source code, change the member name so that it does not begin with an underscore.
- If you require that the member name remain unchanged, remove the CLSCompliantAttribute from its definition or mark it as `<CLSCompliant(False)>`. You can still mark the assembly as `<CLSCompliant(True)>`.

See also

- Declared Element Names
- Visual Basic Naming Conventions

Name '<name>' is not declared

1/30/2019 • 2 minutes to read • Edit Online

A statement refers to a programming element, but the compiler cannot find an element with that exact name.

Error ID: BC30451

To correct this error

1. Check the spelling of the name in the referring statement. Visual Basic is case-insensitive, but any other variation in the spelling is regarded as a completely different name. Note that the underscore (`_`) is part of the name and therefore part of the spelling.

2. Check that you have the member access operator (`.`) between an object and its member. For example, if you have a TextBox control named `TextBox1`, to access its Text property you should type `TextBox1.Text`. If instead you type `TextBox1Text`, you have created a different name.

3. If the spelling is correct and the syntax of any object member access is correct, verify that the element has been declared. For more information, see Declared Elements.

4. If the programming element has been declared, check that it is in scope. If the referring statement is outside the region declaring the programming element, you might need to qualify the element name. For more information, see Scope in Visual Basic.

5. If you are not using a fully qualified type or type and member name (for example, your code refers to a property as `MethodInfo.Name` instead of `System.Reflection.MethodInfo.Name`), add an Imports statement.

6. If you are attempting to compile an SDK-style project (a project with a *.vbproj file that begins with the line `<Project Sdk="Microsoft.NET.Sdk">`), and the error message refers to a type or member in the Microsoft.VisualBasic.dll assembly, configure your application to compile with a reference to the Visual Basic Runtime Library. By default, a subset of the library is embedded in your assembly in an SDK-style project.

   For example, the following example fails to compile because the Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger method cannot be found. It is not embedded in the subset of the Visual Basic Runtime included with your application.

   ```
   Imports Microsoft.VisualBasic.CompilerServices

   Public Module Example
      Sub Main(args As String())
           Dim originalValue = args(0)
           Dim t = GetType(Int32)
           Dim i = Conversions.ChangeType(originalValue, t)
           Console.WriteLine($"'{originalValue}' --> {i}")
       End Sub
   End Module
   ```

   To address this error, add the `<VBRuntime>Default</VBRuntime>` element to the projects `<PropertyGroup>` section, as the following Visual Basic project file shows.

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
      <Reference Include="Microsoft.VisualBasic" />
    </ItemGroup>
  <PropertyGroup>
    <VBRuntime>Default</VBRuntime>
    <OutputType>Exe</OutputType>
    <RootNamespace>vbruntime</RootNamespace>
    <TargetFramework>net472</TargetFramework>
  </PropertyGroup>

</Project>
```

See also

- Declarations and Constants Summary
- Visual Basic Naming Conventions
- Declared Element Names
- References to Declared Elements

Name <namespacename> in the root namespace <fullnamespacename> is not CLS-compliant

4/28/2019 • 2 minutes to read • Edit Online

An assembly is marked as `<CLSCompliant(True)>`, but an element of the root namespace name begins with an underscore (`_`).

A programming element can contain one or more underscores, but to be compliant with the Language Independence and Language-Independent Components (CLS), it must not begin with an underscore. See Declared Element Names.

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40039

To correct this error

- If you require CLS compliance, change the root namespace name so that none of its elements begins with an underscore.
- If you require that the namespace name remain unchanged, then remove the CLSCompliantAttribute from the assembly or mark it as `<CLSCompliant(False)>`.

See also

- Namespace Statement
- Namespaces in Visual Basic
- /rootnamespace
- Application Page, Project Designer (Visual Basic)
- Declared Element Names
- Visual Basic Naming Conventions

Namespace or type specified in the Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

3/5/2019 • 2 minutes to read • Edit Online

Namespace or type specified in the Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found. Make sure the namespace or the type is defined and contains at least one public member. Make sure the alias name doesn't contain other aliases.

An `Imports` statement specifies a containing element that either cannot be found or does not define any `Public` members.

A *containing element* can be a namespace, class, structure, module, interface, or enumeration. The containing element contains members, such as variables, procedures, or other containing elements.

The purpose of importing is to allow your code to access namespace or type members without having to qualify them. Your project might also need to add a reference to the namespace or type. For more information, see "Importing Containing Elements" in References to Declared Elements.

If the compiler cannot find the specified containing element, then it cannot resolve references that use it. If it finds the element but the element does not expose any `Public` members, then no reference can be successful. In either case it is meaningless to import the element.

Keep in mind that if you import a containing element and assign an import alias to it, then you cannot use that import alias to import another element. The following code generates a compiler error.

```
Imports winfrm = System.Windows.Forms

' The following statement is INVALID  because it reuses an import alias.

Imports behave = winfrm.Design.Behavior`
```

Error ID: BC40056

To correct this error

1. Verify that the containing element is accessible from your project.
2. Verify that the specification of the containing element does not include any import alias from another import.
3. Verify that the containing element exposes at least one `Public` member.

See also

- Imports Statement (.NET Namespace and Type)
- Namespace Statement
- Public
- Namespaces in Visual Basic
- References to Declared Elements

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found

4/9/2019 • 2 minutes to read • Edit Online

Namespace or type specified in the project-level Imports '<qualifiedelementname>' doesn't contain any public member or cannot be found. Make sure the namespace or the type is defined and contains at least one public member. Make sure the alias name doesn't contain other aliases.

An import property of a project specifies a containing element that either cannot be found or does not define any `Public` members.

A *containing element* can be a namespace, class, structure, module, interface, or enumeration. The containing element contains members, such as variables, procedures, or other containing elements.

The purpose of importing is to allow your code to access namespace or type members without having to qualify them. Your project might also need to add a reference to the namespace or type. For more information, see "Importing Containing Elements" in References to Declared Elements.

If the compiler cannot find the specified containing element, then it cannot resolve references that use it. If it finds the element but the element does not expose any `Public` members, then no reference can be successful. In either case it is meaningless to import the element.

You use the **Project Designer** to specify elements to import. Use the **Imported namespaces** section of the **References** page. You can get to the **Project Designer** by double-clicking the **My Project** icon in **Solution Explorer**.

Error ID: BC40057

To correct this error

1. Open the **Project Designer** and switch to the **Reference** page.
2. In the **Imported namespaces** section, verify that the containing element is accessible from your project.
3. Verify that the containing element exposes at least one `Public` member.

See also

- References Page, Project Designer (Visual Basic)
- Managing Project and Solution Properties
- Public
- Namespaces in Visual Basic
- References to Declared Elements

Need property array index

4/28/2019 • 2 minutes to read • Edit Online

This property value consists of an array rather than a single value. You did not specify the index for the property array you tried to access.

To correct this error

- Check the component's documentation to find the range for the indexes appropriate for the array. Specify an appropriate index in your property access statement.

See also

- Error Types
- Talk to Us

Nested function does not have a signature that is compatible with delegate '<delegatename>'

4/28/2019 • 2 minutes to read • Edit Online

A lambda expression has been assigned to a delegate that has an incompatible signature. For example, in the following code, delegate `Del` has two integer parameters.

```
Delegate Function Del(ByVal p As Integer, ByVal q As Integer) As Integer
```

The error is raised if a lambda expression with one argument is declared as type `Del`:

```
' Neither of these is valid.
' Dim lambda1 As Del = Function(n As Integer) n + 1
' Dim lambda2 As Del = Function(n) n + 1
```

Error ID: BC36532

To correct this error

- Adjust either the delegate definition or the assigned lambda expression so that the signatures are compatible.

See also

- Relaxed Delegate Conversion
- Lambda Expressions

No accessible 'Main' method with an appropriate signature was found in '<name>'

4/28/2019 • 2 minutes to read • Edit Online

Command-line applications must have a `Sub Main` defined. `Main` must be declared as `Public Shared` if it is defined in a class, or as `Public` if defined in a module.

Error ID: BC30737

To correct this error

- Define a `Public Sub Main` procedure for your project. Declare it as `Shared` if and only if you define it inside a class.

See also

- Structure of a Visual Basic Program
- Procedures

Non-CLS-compliant <membername> is not allowed in a CLS-compliant interface

4/28/2019 • 2 minutes to read • Edit Online

A property, procedure, or event in an interface is marked as `<CLSCompliant(True)>` when the interface itself is marked as `<CLSCompliant(False)>` or is not marked.

For an interface to be compliant with the Language Independence and Language-Independent Components (CLS), all its members must be compliant.

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40033

To correct this error

- If you require CLS compliance and have control over the interface source code, mark the interface as `<CLSCompliant(True)>` if all its members are compliant.
- If you require CLS compliance and do not have control over the interface source code, or if it does not qualify to be compliant, define this member within a different interface.
- If you require that this member remain within its current interface, remove the CLSCompliantAttribute from its definition or mark it as `<CLSCompliant(False)>`.

See also

- Interface Statement

Nullable type inference is not supported in this context

4/28/2019 • 2 minutes to read • Edit Online

Value types and structures can be declared nullable.

```
Dim a? As Integer
Dim b As Integer?
```

However, you cannot use the nullable declaration in combination with type inference. The following examples cause this error.

```
' Not valid.
' Dim c? = 10
' Dim d? = a
```

Error ID: BC36629

To correct this error

- Use an `As` clause to declare the variable as nullable.

See also

- Nullable Value Types
- Local Type Inference

Number of indices exceeds the number of dimensions of the indexed array

4/28/2019 • 2 minutes to read • Edit Online

The number of indices used to access an array element must be exactly the same as the rank of the array, that is, the number of dimensions declared for it.

Error ID: BC30106

To correct this error

- Remove subscripts from the array reference until the total number of subscripts equals the rank of the array. For example:

```
Dim gameBoard(3, 3) As String

' Incorrect code. The array has two dimensions.
gameBoard(1, 1, 1) = "X"
gameBoard(2, 1, 1) = "O"

' Correct code.
gameBoard(0, 0) = "X"
gameBoard(1, 0) = "O"
```

See also

- Arrays

Object or class does not support the set of events

4/9/2019 • 2 minutes to read • Edit Online

You tried to use a `WithEvents` variable with a component that cannot work as an event source for the specified set of events. For example, you wanted to sink the events of an object, then create another object that `Implements` the first object. Although you might think you could sink the events from the implemented object, this is not always the case. `Implements` only implements an interface for methods and properties. `WithEvents` is not supported for private `UserControls`, because the type info needed to raise the `ObjectEvent` is not available at run time.

To correct this error

1. You cannot sink events for a component that does not source events.

See also

- WithEvents
- Implements Statement

Object required (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

References to properties and methods often require an explicit object qualifier. This is such a case.

To correct this error

1. Check that references to an object property or method have valid object qualifier. Specify an object qualifier if you didn't provide one.
2. Check the spelling of the object qualifier and make sure the object is visible in the part of the program in which you are referencing it.
3. If a path is supplied to a host application's **File Open** command, check that the arguments in it are correct.
4. Check the object's documentation and make sure the action is valid.

See also

- Error Types

Object variable or With block variable not set

8/27/2019 • 2 minutes to read • Edit Online

An invalid object variable is being referenced. This error can occur for several reasons:

- A variable was declared without specifying a type. If a variable is declared without specifying a type, it defaults to type `Object`.

  For example, a variable declared with `Dim x` would be of type `Object;` a variable declared with `Dim x As String` would be of type `String`.

  > **TIP**
  > The `Option Strict` statement disallows implicit typing that results in an `Object` type. If you omit the type, a compile-time error will occur. See Option Strict Statement.

- You are attempting to reference an object that has been set to `Nothing`.
- You are attempting to access an element of an array variable that wasn't properly declared.

  For example, an array declared as `products() As String` will trigger the error if you try to reference an element of the array `products(3) = "Widget"`. The array has no elements and is treated as an object.

- You are attempting to access code within a `With...End With` block before the block has been initialized. A `With...End With` block must be initialized by executing the `With` statement entry point.

> **NOTE**
> In earlier versions of Visual Basic or VBA this error was also triggered by assigning a value to a variable without using the `Set` keyword (`x = "name"` instead of `Set x = "name"`). The `Set` keyword is no longer valid in Visual Basic .Net.

To correct this error

1. Set `Option Strict` to `On` by adding the following code to the beginning of the file:

   ```
   Option Strict On
   ```

   When you run the project, a compiler error will appear in the **Error List** for any variable that was specified without a type.

2. If you don't want to enable `Option Strict`, search your code for any variables that were specified without a type (`Dim x` instead of `Dim x As String`) and add the intended type to the declaration.

3. Make sure you aren't referring to an object variable that has been set to `Nothing`. Search your code for the keyword `Nothing`, and revise your code so that the object isn't set to `Nothing` until after you have referenced it.

4. Make sure that any array variables are dimensioned before you access them. You can either assign a dimension when you first create the array (`Dim x(5) As String` instead of `Dim x() As String`), or use the `ReDim` keyword to set the dimensions of the array before you first access it.

5. Make sure your `With` block is initialized by executing the `With` statement entry point.

See also

- Object Variable Declaration
- ReDim Statement
- With...End With Statement

Operator declaration must be one of: +,-,*,,/,^, &, Like, Mod, And, Or, Xor, Not, <<, >>...

4/9/2019 • 2 minutes to read • Edit Online

You can declare only an operator that is eligible for overloading. The following table lists the operators you can declare.

TYPE	OPERATORS
Unary	`+`, `-`, `IsFalse`, `IsTrue`, `Not`
Binary	`+`, `-`, `*`, `/`, `\`, `&`, `^`, `>>`, `<<`, `=`, `<>`, `>`, `>=`, `<`, `<=`, `And`, `Like`, `Mod`, `Or`, `Xor`
Conversion (unary)	`CType`

Note that the `=` operator in the binary list is the comparison operator, not the assignment operator.

Error ID: BC33000

To correct this error

1. Select an operator from the set of overloadable operators.
2. If you need the functionality of overloading an operator that you cannot overload directly, create a `Function` procedure that takes the appropriate parameters and returns the appropriate value.

See also

- Operator Statement
- Operator Procedures
- How to: Define an Operator
- How to: Define a Conversion Operator
- Function Statement

'Optional' expected

4/9/2019 • 2 minutes to read • Edit Online

An optional argument in a procedure declaration is followed by a required argument. Every argument following an optional argument must also be optional.

Error ID: BC30202

To correct this error

1. If the argument is intended to be required, move it to precede the first optional argument in the argument list.
2. If the argument is intended to be optional, use the `Optional` keyword.

See also

- Optional Parameters

Optional parameters must specify a default value

4/28/2019 • 2 minutes to read • Edit Online

Optional parameters must provide default values that can be used if no parameter is supplied by a calling procedure.

Error ID: BC30812

To correct this error

- Specify default values for optional parameters; for example:

```
Sub Proc1(ByVal X As Integer,
      Optional ByVal Y As String = "Default Value")
   MsgBox("Default argument is: " & Y)
End Sub
```

See also

- Optional

Ordinal is not valid

4/28/2019 • 2 minutes to read • Edit Online

Your call to a dynamic-link library (DLL) indicated to use a number instead of a procedure name, using the `#num` syntax. This error has the following possible causes:

- An attempt to convert the `#num` expression to an ordinal failed.
- The `#num` specified does not specify any function in the DLL.
- A type library has an invalid declaration resulting in internal use of an invalid ordinal number.

To correct this error

1. Make sure the expression represents a valid number, or call the procedure by name.
2. Make sure `#num` identifies a valid function in the DLL.
3. Isolate the procedure call causing the problem by commenting out the code. Write a `Declare` statement for the procedure, and report the problem to the type library vendor.

See also

- Declare Statement

Out of memory (Visual Basic Compiler Error)

4/28/2019 • 2 minutes to read • Edit Online

More memory was required than is available.

Error ID: BC2004

To correct this error

- Close unnecessary applications, documents and source files.
- Eliminate unnecessary controls and forms so fewer are loaded at one time
- Reduce the number of `Public` variables.
- Check available disk space.
- Increase the available RAM by installing additional memory or reallocating memory.
- Make sure that memory is freed when it is no longer needed.

See also

- Error Types

Out of stack space (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

The stack is a working area of memory that grows and shrinks dynamically with the demands of your executing program. Its limits have been exceeded.

To correct this error

1. Check that procedures are not nested too deeply.
2. Make sure recursive procedures terminate properly.
3. If local variables require more local variable space than is available, try declaring some variables at the module level. You can also declare all variables in the procedure static by preceding the `Property`, `Sub`, or `Function` keyword with `Static`. Or you can use the `Static` statement to declare individual static variables within procedures.
4. Redefine some of your fixed-length strings as variable-length strings, as fixed-length strings use more stack space than variable-length strings. You can also define the string at module level where it requires no stack space.
5. Check the number of nested `DoEvents` function calls, by using the `Calls` dialog box to view which procedures are active on the stack.
6. Make sure you did not cause an "event cascade" by triggering an event that calls an event procedure already on the stack. An event cascade is similar to an unterminated recursive procedure call, but it is less obvious, since the call is made by Visual Basic rather than an explicit call in the code. Use the `Calls` dialog box to view which procedures are active on the stack.

See also

- Memory Windows

Out of string space (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

With Visual Basic, you can use very large strings. However, the requirements of other programs and the way you work with your strings can still cause this error.

To correct this error

1. Make sure that an expression requiring temporary string creation during evaluation is not causing the error.
2. Remove any unnecessary applications from memory to create more space.

See also

- Error Types
- String Manipulation Summary

Overflow (Visual Basic Error)

4/28/2019 • 2 minutes to read • Edit Online

A literal represents a value outside the limits of the data type to which it is being assigned.

Error ID: BC30036

To correct this error

- Consult the value range for the target data type and rewrite the literal to conform to that range.

See also

- Data Types

Overflow (Visual Basic Run-Time Error)

4/9/2019 • 2 minutes to read • Edit Online

An overflow results when you attempt an assignment that exceeds the limits of the assignment's target.

To correct this error

1. Make sure that results of assignments, calculations, and data type conversions are not too large to be represented within the range of variables allowed for that type of value, and assign the value to a variable of a type that can hold a larger range of values, if necessary.
2. Make sure assignments to properties fit the range of the property to which they are made.
3. Make sure that numbers used in calculations that are coerced into integers do not have results larger than integers.

See also

- Int32.MaxValue
- Double.MaxValue
- Data Types
- Error Types

Path not found

4/28/2019 • 2 minutes to read • Edit Online

During a file-access or disk-access operation, the operating system was unable to find the specified path. The path to a file includes the drive specification plus the directories and subdirectories that must be traversed to locate the file. A path can be relative or absolute.

To correct this error

- Verify and respecify the path.

See also

- Error Types

Path/File access error

4/9/2019 • 2 minutes to read • Edit Online

During a file-access or disk-access operation, the operating system could not make a connection between the path and the file name.

To correct this error

1. Make sure the file specification is correctly formatted. A file name can contain a fully qualified (absolute) or relative path. A fully qualified path starts with the drive name (if the path is on another drive) and lists the explicit path from the root to the file. Any path that is not fully qualified is relative to the current drive and directory.
2. Make sure that you did not attempt to save a file that would replace an existing read-only file. If this is the case, change the read-only attribute of the target file, or save the file with a different file name.
3. Make sure you did not attempt to open a read-only file in sequential `Output` or `Append` mode. If this is the case, open the file in `Input` mode or change the read-only attribute of the file.
4. Make sure you did not attempt to change a Visual Basic project within a database or document.

See also

- Error Types

Permission denied (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

An attempt was made to write to a write-protected disk or to access a locked file.

To correct this error

1. To open a write-protected file, change the write-protection attribute of the file.
2. Make sure that another process has not locked the file, and wait to open the file until the other process releases it.
3. To access the registry, check that your user permissions include this type of registry access.

See also

- Error Types

Procedure call or argument is not valid (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

Some part of the call cannot be completed.

To correct this error

- Check the permitted ranges for arguments to make sure no arrangement exceeds the permitted values.

See also

- Error Types

Property '<propertyname>' doesn't return a value on all code paths

4/28/2019 • 2 minutes to read • Edit Online

Property '<propertyname>' doesn't return a value on all code paths. A null reference exception could occur at run time when the result is used.

A property `Get` procedure has at least one possible path through its code that does not return a value.

You can return a value from a property `Get` procedure in any of the following ways:

- Assign the value to the property name and then perform an `Exit Property` statement.
- Assign the value to the property name and then perform the `End Get` statement.
- Include the value in a Return Statement.

If control passes to `Exit Property` or `End Get` and you have not assigned any value to the property name, the `Get` procedure returns the default value of the property's data type. For more information, see "Behavior" in Function Statement.

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42107

To correct this error

- Check your control flow logic and make sure you assign a value before every statement that causes a return.

  It is easier to guarantee that every return from the procedure returns a value if you always use the `Return` statement. If you do this, the last statement before `End Get` should be a `Return` statement.

See also

- Property Procedures
- Property Statement
- Get Statement

Property array index is not valid

4/28/2019 • 2 minutes to read • Edit Online

The supplied value is not valid for a property array index.

To correct this error

- Check the component's documentation to make sure your index is within the valid range for the specified property.

See also

- Arrays

Property let procedure not defined and property get procedure did not return an object

4/9/2019 • 2 minutes to read • Edit Online

Certain properties, methods, and operations can only apply to `Collection` objects. You specified an operation or property that is exclusive to collections, but the object is not a collection.

To correct this error

1. Check the spelling of the object or property name, or verify that the object is a `Collection` object.
2. Look at the `Add` method used to add the object to the collection to be sure the syntax is correct and that any identifiers were spelled correctly.

See also

- Collection

Property not found

4/9/2019 • 2 minutes to read • Edit Online

This object does not support the specified property.

To correct this error

1. Check the spelling of the property's name.
2. Check the object's documentation to make sure you are not trying to access something like a "text" property when the object actually supports a "caption" or similarly named property.

See also

- Error Types

Property or method not found

4/28/2019 • 2 minutes to read • Edit Online

The referenced object method or object property is not defined.

To correct this error

- You may have misspelled the name of the object. To see what properties and methods are defined for an object, display the Object Browser. Select the appropriate object library to view a list of available properties and methods.

See also

- Error Types

Range variable <variable> hides a variable in an enclosing block, a previously defined range variable, or an implicitly declared variable in a query expression

4/28/2019 • 2 minutes to read • Edit Online

A range variable name specified in a `Select`, `From`, `Aggregate`, or `Let` clause duplicates the name of a range variable already specified previously in the query, or the name of a variable that is implicitly declared by the query, such as a field name or the name of an aggregate function.

Error ID: BC36633

To correct this error

- Ensure that all range variables in a particular query scope have unique names. You can enclose a query in parentheses to ensure that nested queries have a unique scope.

See also

- Introduction to LINQ in Visual Basic
- From Clause
- Let Clause
- Aggregate Clause
- Select Clause

Range variable name can be inferred only from a simple or qualified name with no arguments

4/9/2019 • 2 minutes to read • Edit Online

A programming element that takes one or more arguments is included in a LINQ query. The compiler is unable to infer a range variable from that programming element.

Error ID: BC36599

To correct this error

1. Supply an explicit variable name for the programming element, as shown in the following code:

```
Dim query = From var1 In collection1
            Select VariableAlias= SampleFunction(var1), var1
```

See also

- Introduction to LINQ in Visual Basic
- Select Clause

Reference required to assembly '<assemblyidentity>' containing type '<typename>', but a suitable reference could not be found due to ambiguity between projects '<projectname1>' and '<projectname2>'

4/9/2019 • 2 minutes to read • Edit Online

An expression uses a type, such as a class, structure, interface, enumeration, or delegate, that is defined outside your project. However, you have project references to more than one assembly defining that type.

The cited projects produce assemblies with the same name. Therefore, the compiler cannot determine which assembly to use for the type you are accessing.

To access a type defined in another assembly, the Visual Basic compiler must have a reference to that assembly. This must be a single, unambiguous reference that does not cause circular references among projects.

Error ID: BC30969

To correct this error

1. Determine which project produces the best assembly for your project to reference. For this decision, you might use criteria such as ease of file access and frequency of updates.
2. In your project properties, add a reference to the file that contains the assembly that defines the type you are using.

See also

- Managing references in a project
- References to Declared Elements
- Managing Project and Solution Properties
- Troubleshooting Broken References

Reference required to assembly '<assemblyname>' containing the base class '<classname>'

4/28/2019 • 2 minutes to read • Edit Online

Reference required to assembly '<assemblyname>' containing the base class '<classname>'. Add one to your project.

The class is defined in a dynamic-link library (DLL) or assembly that is not directly referenced in your project. The Visual Basic compiler requires a reference to avoid ambiguity in case the class is defined in more than one DLL or assembly.

Error ID: BC30007

To correct this error

- Include the name of the unreferenced DLL or assembly in your project references.

See also

- Managing references in a project
- Troubleshooting Broken References

Requested operation is not available because the runtime library function '<function>' is not defined.

3/5/2019 • 2 minutes to read • Edit Online

Visual Basic is attempting to make an internal call to a function in the Visual Basic Runtime (Microsoft.VisualBasic.dll) that cannot be found.

Error ID: BC35000

This error occurs in SDK-style projects (projects with a *.vbproj file that begins with the line `<Project Sdk="Microsoft.NET.Sdk">`). By default, only a subset of the Microsoft.VisualBasic.dll assembly is embedded in the application assembly, and <*function*> is not included in that subset.

To correct this error

Rather than embedding a subset of the Visual Basic Runtime in your assembly, you must compile with a reference to it. You do this by adding the following element to the `<PropertyGroup>` section of your *.vbproj file:

```
<VBRuntime>Default</VBRuntime>
```

See also

- **-vbruntime** compiler option

Resume without error

4/9/2019 • 2 minutes to read • Edit Online

A `Resume` statement appeared outside error-handling code, or the code jumped into an error handler even though there was no error.

To correct this error

1. Move the `Resume` statement into an error handler, or delete it.
2. Jumps to labels cannot occur across procedures, so search the procedure for the label that identifies the error handler. If you find a duplicate label specified as the target of a `GoTo` statement that isn't an `On Error GoTo` statement, change the line label to agree with its intended target.

See also

- Resume Statement
- On Error Statement

Return type of function '<procedurename>' is not CLS-compliant

5/15/2019 • 2 minutes to read • Edit Online

A `Function` procedure is marked as `<CLSCompliant(True)>` but returns a type that is marked as `<CLSCompliant(False)>`, is not marked, or does not qualify because it is a noncompliant type.

For a procedure to be compliant with the Language Independence and Language-Independent Components (CLS), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40027

To correct this error

- If the `Function` procedure must return this particular type, remove the CLSCompliantAttribute. The procedure cannot be CLS-compliant.
- If the `Function` procedure must be CLS-compliant, change the return type to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are returning a 16-bit integer to such a component, declare it as `Short` instead of `Integer` in your managed Visual Basic code.

'Set' accessor of property '<propertyname>' is not accessible

4/28/2019 • 2 minutes to read • Edit Online

A statement attempts to store the value of a property when it does not have access to the property's `Set` procedure.

If the Set Statement is marked with a more restrictive access level than its Property Statement, an attempt to set the property value could fail in the following cases:

- The `Set` statement is marked Private and the calling code is outside the class or structure in which the property is defined.
- The `Set` statement is marked Protected and the calling code is not in the class or structure in which the property is defined, nor in a derived class.
- The `Set` statement is marked Friend and the calling code is not in the same assembly in which the property is defined.

Error ID: BC31102

To correct this error

- If you have control of the source code defining the property, consider declaring the `Set` procedure with the same access level as the property itself.
- If you do not have control of the source code defining the property, or you must restrict the `Set` procedure access level more than the property itself, try to move the statement that sets the property value to a region of code that has better access to the property.

See also

- Property Procedures
- How to: Declare a Property with Mixed Access Levels

Some subkeys cannot be deleted

4/28/2019 • 2 minutes to read • Edit Online

An attempt has been made to delete a registry key, but the operation failed because some subkeys cannot be deleted. Usually this is due to a lack of permissions.

To correct this error

- Make sure you have sufficient permissions to delete the specified subkeys.

See also

- Microsoft.VisualBasic.MyServices.RegistryProxy
- DeleteSubKey
- RegistryPermission

Statement cannot end a block outside of a line 'If' statement

4/28/2019 • 2 minutes to read • Edit Online

A single-line `If` statement contains several statements separated by colons (:), one of which is an `End` statement for a control block outside the single-line `If`. Single-line `If` statements do not use the `End If` statement.

Error ID: BC32005

To correct this error

- Move the single-line `If` statement outside the control block that contains the `End If` statement.

See also

- If...Then...Else Statement

Statement is not valid in a namespace

4/28/2019 • 2 minutes to read • Edit Online

The statement cannot appear at the level of a namespace. The only declarations allowed at namespace level are module, interface, class, delegate, enumeration, and structure declarations.

Error ID: BC30001

To correct this error

- Move the statement to a location within a module, class, interface, structure, enumeration, or delegate definition.

See also

- Scope in Visual Basic
- Namespaces in Visual Basic

Statement is not valid inside a method/multiline lambda

4/28/2019 • 2 minutes to read • Edit Online

The statement is not valid within a `Sub`, `Function`, property `Get`, or property `Set` procedure. Some statements can be placed at the module or class level. Others, such as `Option Strict`, must be at namespace level and precede all other declarations.

Error ID: BC30024

To correct this error

- Remove the statement from the procedure.

See also

- Sub Statement
- Function Statement
- Get Statement
- Set Statement

String constants must end with a double quote

4/28/2019 • 2 minutes to read • Edit Online

String constants must begin and end with quotation marks.

ErrorID: BC30648

To correct this error

- Make sure the string literal ends with a quotation mark ("). If you paste values from other text editors, make sure the pasted character is a valid quotation mark and not one of the characters that resemble it, such as "smart" or "curly" quotation marks (" or ") or two single quotation marks ('').

See also

- Strings

Structure '<structurename>' must contain at least one instance member variable or at least one instance event declaration not marked 'Custom'

4/28/2019 • 2 minutes to read • Edit Online

A structure definition does not include any nonshared variables or nonshared noncustom events.

Every structure must have either a variable or an event that applies to each specific instance (nonshared) instead of to all instances collectively (Shared). Nonshared constants, properties, and procedures do not satisfy this requirement. In addition, if there are no nonshared variables and only one nonshared event, that event cannot be a `Custom` event.

Error ID: BC30941

To correct this error

- Define at least one variable or event that is not `Shared`. If you define only one event, it must be noncustom as well as nonshared.

See also

- Structures
- How to: Declare a Structure
- Structure Statement

'Sub Main' was not found in '<name>'

4/9/2019 • 2 minutes to read • Edit Online

`Sub Main` is missing, or the wrong location has been specified for it.

Error ID: BC30420

To correct this error

1. Supply the missing `Sub Main` statement, or if it exists, move it to the appropriate location in the code. For more information on `Sub Main`, see Main Procedure in Visual Basic.
2. Specify the location of the project's startup object in the **Startup form** box of the **Project Designer**.

See also

- Sub Statement
- Main Procedure in Visual Basic

Sub or Function not defined (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

A `Sub` or `Function` must be defined in order to be called. Possible causes of this error include:

- Misspelling the procedure name.
- Trying to call a procedure from another project without explicitly adding a reference to that project in the **References** dialog box.
- Specifying a procedure that is not visible to the calling procedure.
- Declaring a Windows dynamic-link library (DLL) routine or Macintosh code-resource routine that is not in the specified library or code resource.

To correct this error

1. Make sure that the procedure name is spelled correctly.
2. Find the name of the project containing the procedure you want to call in the **References** dialog box. If it does not appear, click the **Browse** button to search for it. Select the check box to the left of the project name, and then click **OK**.
3. Check the name of the routine.

See also

- Error Types
- Managing references in a project
- Sub Statement
- Function Statement

Subscript out of range (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

An array subscript is not valid because it falls outside the allowable range. The lowest subscript value for a dimension is always 0, and the highest subscript value is returned by the `GetUpperBound` method for that dimension.

To correct this error

- Change the subscript so it falls within the valid range.

See also

- Array.GetUpperBound
- Arrays

TextFieldParser is unable to complete the read operation because maximum buffer size has been exceeded

4/28/2019 • 2 minutes to read • Edit Online

The operation cannot be completed because the maximum buffer size (10,000,000 bytes) has been exceeded.

To correct this error

- Make sure there are no malformed fields in the file.

See also

- OpenTextFieldParser
- TextFieldParser
- How to: Read From Text Files with Multiple Formats
- Parsing Text Files with the TextFieldParser Object

The type for variable '<variablename>' will not be inferred because it is bound to a field in an enclosing scope

7/26/2019 • 2 minutes to read • Edit Online

The type for variable '<variablename>' will not be inferred because it is bound to a field in an enclosing scope. Either change the name of '<variablename>', or use the fully qualified name (for example, 'Me.variablename' or 'MyBase.variablename').

A loop control variable in your code has the same name as a field of the class or other enclosing scope. Because the control variable is used without an `As` clause, it is bound to the field in the enclosing scope, and the compiler does not create a new variable for it or infer its type.

In the following example, `Index`, the control variable in the `For` statement, is bound to the `Index` field in the `Customer` class. The compiler does not create a new variable for the control variable `Index` or infer its type.

```
Class Customer

    ' The class has a field named Index.
    Private Index As Integer

    Sub Main()

    ' The following line will raise this warning.
        For Index = 1 To 10
            ' ...
        Next

    End Sub
End Class
```

By default, this message is a warning. For information about how to hide warnings or how to treat warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42110

To address this warning

- Make the loop control variable local by changing its name to an identifier that is not also the name of a field of the class.

  ```
  For I = 1 To 10
  ```

- Clarify that the loop control variable binds to the class field by prefixing `Me.` to the variable name.

  ```
  For Me.Index = 1 To 10
  ```

- Instead of relying on local type inference, use an `As` clause to specify a type for the loop control variable.

  ```
  For Index As Integer = 1 To 10
  ```

Example

The following code shows the earlier example with the first correction in place.

```
Class Customer

    ' The class has a field named Index.
    Private Index As Integer

    Sub Main()

        For I = 1 To 10
            ' ...
        Next

    End Sub
End Class
```

See also

- Option Infer Statement
- For Each...Next Statement
- For...Next Statement
- How to: Refer to the Current Instance of an Object
- Local Type Inference
- Me, My, MyBase, and MyClass

This array is fixed or temporarily locked (Visual Basic)

4/28/2019 • 2 minutes to read • Edit Online

This error has the following possible causes:

- Using `ReDim` to change the number of elements of a fixed-size array.
- Redimensioning a module-level dynamic array, in which one element has been passed as an argument to a procedure. If an element is passed, the array is locked to prevent deallocating memory for the reference parameter within the procedure.
- Attempting to assign a value to a `Variant` variable containing an array, but the `Variant` is currently locked.

To correct this error

1. Make the original array dynamic rather than fixed by declaring it with `ReDim` (if the array is declared within a procedure), or by declaring it without specifying the number of elements (if the array is declared at the module level.
2. Determine whether you really need to pass the element, since it is visible within all procedures in the module.
3. Determine what is locking the `Variant` and remedy it.

See also

- Arrays

This key is already associated with an element of this collection

4/28/2019 • 2 minutes to read • Edit Online

The specified a key for a collection member already identifies another member of the collection. A key is a string specified in the `Add` method that uniquely identifies a specific member of a collection.

To correct this error

- Use a different key for this member.

See also

- Error Types

Too many files

4/9/2019 • 2 minutes to read • Edit Online

Either more files have been created in the root directory than the operating system permits, or more files have been opened than the number specified in the **files=** setting in your CONFIG.SYS file.

To correct this error

1. If your program is opening, closing, or saving files in the root directory, change your program so that it uses a subdirectory.
2. Increase the number of files specified in your **files=** setting in your CONFIG.SYS file, and restart your computer.

See also

- Error Types

Type '<typename>' has no constructors

4/9/2019 • 2 minutes to read • Edit Online

A type does not support a call to `Sub New()`. One possible cause is a corrupted compiler or binary file.

Error ID: BC30251

To correct this error

1. If the type is in a different project or in a referenced file, reinstall the project or file.
2. If the type is in the same project, recompile the assembly containing the type.
3. If the error recurs, reinstall the Visual Basic compiler.
4. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- Objects and Classes
- Talk to Us

Type <typename> is not CLS-compliant

5/15/2019 • 2 minutes to read • Edit Online

A variable, property, or function return is declared with a data type that is not CLS-compliant.

For an application to be compliant with the Language Independence and Language-Independent Components (CLS), it must use only CLS-compliant types.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

Error ID: BC40041

To correct this error

- If your application needs to be CLS-compliant, change the data type of this element to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If your application does not need to be CLS-compliant, you do not need to change anything. You should be aware of its noncompliance, however.

Type '<typename>' is not defined

4/28/2019 • 2 minutes to read • Edit Online

The statement has made reference to a type that has not been defined. You can define a type in a declaration statement such as `Enum`, `Structure`, `Class`, or `Interface`.

Error ID: BC30002

To correct this error

- Ensure that the type definition and its reference both use the same spelling.
- Ensure that the type definition is accessible to the reference. For example, if the type is in another module and has been declared `Private`, move the type definition to the referencing module or declare it `Public`.
- Ensure that the namespace of the type is not redefined within your project. If it is, use the `Global` keyword to fully qualify the type name. For example, if a project defines a namespace named `System`, the System.Object type cannot be accessed unless it is fully qualified with the `Global` keyword: `Global.System.Object`.
- If the type is defined, but the object library or type library in which it is defined is not registered in Visual Basic, click **Add Reference** on the **Project** menu, and then select the appropriate object library or type library.
- Ensure that the type is in an assembly that is part of the targeted .NET Framework profile. For more information, see Troubleshooting .NET Framework Targeting Errors.

See also

- Namespaces in Visual Basic
- Enum Statement
- Structure Statement
- Class Statement
- Interface Statement
- Managing references in a project

Type arguments could not be inferred from the delegate

4/28/2019 • 2 minutes to read • Edit Online

An assignment statement uses `AddressOf` to assign the address of a generic procedure to a delegate, but it does not supply any type arguments to the generic procedure.

Normally, when you invoke a generic type, you supply a type argument for each type parameter that the generic type defines. If you do not supply any type arguments, the compiler attempts to infer the types to be passed to the type parameters. If the context does not provide enough information for the compiler to infer the types, an error is generated.

Error ID: BC36564

To correct this error

- Specify the type arguments for the generic procedure in the `AddressOf` expression.

See also

- Generic Types in Visual Basic
- AddressOf Operator
- Generic Procedures in Visual Basic
- Type List
- Extension Methods

Type mismatch (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

You attempted to convert a value to another type in a way that is not valid.

To correct this error

1. Check the assignment to make sure it is valid.
2. Make sure you did not pass an object to a procedure that expects a single property or value.
3. Make sure you did not use a module or project name where an expression was expected.

See also

- Error Types

Type of '<variablename>' cannot be inferred because the loop bounds and the step variable do not widen to the same type

7/26/2019 • 2 minutes to read • Edit Online

You have written a `For...Next` loop in which the compiler cannot infer a data type for the loop control variable because the following conditions are true:

- The data type of the loop control variable is not specified with an `As` clause.
- The loop bounds and step variable contain at least two data types.
- No standard conversions exist between the data types.

Therefore, the compiler cannot infer the data type of a loop's control variable.

In the following example, the step variable is a character and the loop bounds are both integers. Because there is no standard conversion between characters and integers, this error is reported.

```
Dim stepVar = "1"c
Dim m = 0
Dim n = 20

' Not valid.
' For i = 1 To 10 Step stepVar
    ' Loop processing
' Next
```

Error ID: BC30982

To correct this error

- Change the types of the loop bounds and step variable as necessary so that at least one of them is a type that the others widen to. In the preceding example, change the type of `stepVar` to `Integer`.

  ```
  Dim stepVar = 1
  ```

  -or-

  ```
  Dim stepVar As Integer = 1
  ```

- Use explicit conversion functions to convert the loop bounds and step variable to the appropriate types. In the preceding example, apply the `Val` function to `stepVar`.

  ```
  For i = 1 To 10 Step Val(stepVar)
      ' Loop processing
  Next
  ```

See also

- Val
- For...Next Statement
- Implicit and Explicit Conversions
- Local Type Inference
- Option Infer Statement
- Type Conversion Functions
- Widening and Narrowing Conversions

Type of member '<membername>' is not CLS-compliant

5/14/2019 • 2 minutes to read • Edit Online

The data type specified for this member is not part of the Language Independence and Language-Independent Components (CLS). This is not an error within your component, because the .NET Framework and Visual Basic support this data type. However, another component written in strictly CLS-compliant code might not support this data type. Such a component might not be able to interact successfully with your component.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40025

To correct this error

- If your component interfaces only with other .NET Framework components, or does not interface with any other components, you do not need to change anything.
- If you are interfacing with a component not written for the .NET Framework, you might be able to determine, either through reflection or from documentation, whether it supports this data type. If it does, you do not need to change anything.
- If you are interfacing with a component that does not support this data type, you must replace it with the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `uint` is often 16 bits in other environments. If you are passing a 16-bit argument to such a component, declare it as `UShort` instead of `UInteger` in your managed Visual Basic code.

See also

- Reflection

Type of optional value for optional parameter <parametername> is not CLS-compliant

5/15/2019 • 2 minutes to read • Edit Online

A procedure is marked as `<CLSCompliant(True)>` but declares an Optional parameter with default value of a noncompliant type.

For a procedure to be compliant with the Language Independence and Language-Independent Components (CLS), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables. It also applies to the default values of optional parameters.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

When you apply the CLSCompliantAttribute attribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40042

To correct this error

- If the optional parameter must have a default value of this particular type, remove CLSCompliantAttribute. The procedure cannot be CLS-compliant.
- If the procedure must be CLS-compliant, change the type of this default value to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are accepting a 16-bit integer from such a component, declare it as `Short` instead of `Integer` in your managed Visual Basic code.

Type of parameter '<parametername>' is not CLS-compliant

5/15/2019 • 2 minutes to read • Edit Online

A procedure is marked as `<CLSCompliant(True)>` but declares a parameter with a type that is marked as `<CLSCompliant(False)>`, is not marked, or does not qualify because it is a noncompliant type.

For a procedure to be compliant with the Language Independence and Language-Independent Components (CLS), it must use only CLS-compliant types. This applies to the types of the parameters, the return type, and the types of all its local variables.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

When you apply the CLSCompliantAttribute to a programming element, you set the attribute's `isCompliant` parameter to either `True` or `False` to indicate compliance or noncompliance. There is no default for this parameter, and you must supply a value.

If you do not apply the CLSCompliantAttribute to an element, it is considered to be noncompliant.

By default, this message is a warning. For information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40028

To correct this error

- If the procedure must take a parameter of this particular type, remove the CLSCompliantAttribute. The procedure cannot be CLS-compliant.
- If the procedure must be CLS-compliant, change the type of this parameter to the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `int` is often 16 bits in other environments. If you are accepting a 16-bit integer from such a component, declare it as `Short` instead of `Integer` in your managed Visual Basic code.

Type parameters cannot be used as qualifiers

4/9/2019 • 2 minutes to read • Edit Online

A programming element is qualified with a qualification string that includes a type parameter.

A type parameter represents a requirement for a type that is to be supplied when the generic type is constructed. It does not represent a specific defined type. A qualification string must include only elements that are defined at compile time.

The following statements can generate this error.

```
Public Function checkText(Of c As System.Windows.Forms.Control)(
    ByVal badText As String) As Boolean

    Dim saveText As c.Text
    ' Insert code to look for badText within saveText.
End Function
```

Error ID: BC32098

To correct this error

1. Remove the type parameter from the qualification string, or replace it with a defined type.
2. If you need to use a constructed type to locate the programming element being qualified, you must use additional program logic.

See also

- References to Declared Elements
- Generic Types in Visual Basic
- Type List

Unable to create strong-named assembly from key file '<filename>': <error>

4/9/2019 • 2 minutes to read • Edit Online

A strong-named assembly could not be created from the specified key file.

Error ID: BC31026

To correct this error

1. Verify that the correct key file has been specified, and that it is not locked by another application.

See also

- Sn.exe (Strong Name Tool))

Unable to embed resource file '<filename>': <error message>

4/9/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error embedding a native COM+ resource file directly into the assembly.

Error ID: BC30143

To correct this error

1. Examine the quoted error message and consult the topic Al.exe. for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- Al.exe
- Talk to Us

Unable to emit assembly: <error message>

5/15/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (*Al.exe*, also known as Alink) to generate an assembly with a manifest, and the linker reports an error in the emission stage of creating the assembly.

Error ID: BC30145

To correct this error

1. Examine the quoted error message and consult the topic Al.exe for further explanation and advice.
2. Try signing the assembly manually, using either the Al.exe or the Sn.exe (Strong Name Tool).
3. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

To sign the assembly manually

1. Use the Sn.exe (Strong Name Tool)) to create a public/private key pair file.

   This file has an *.snk* extension.

2. Delete the COM reference that is generating the error from your project.
3. Open the Developer Command Prompt for Visual Studio.

   In Windows 10, enter **Developer command prompt** into the search box on the task bar. Then, select **Developer Command Prompt for VS 2017** from the results list.

4. Change the directory to the directory where you want to place your assembly wrapper.
5. Enter the following command:

   ```
   tlbimp <path to COM reference file> /out:<output assembly name> /keyfile:<path to .snk file>
   ```

   An example of the actual command you might enter is:

   ```
   tlbimp c:\windows\system32\msi.dll /out:Interop.WindowsInstaller.dll /keyfile:"c:\documents and
   settings\mykey.snk"
   ```

   TIP

   Use double quotation marks if a path or file contains spaces.

6. In Visual Studio, add a .NET Assembly reference to the file you just created.

See also

- Al.exe
- Sn.exe (Strong Name Tool)
- How to: Create a Public-Private Key Pair

- Talk to Us

Unable to find required file '<filename>'

4/28/2019 • 2 minutes to read • Edit Online

A file that is required by Visual Studio is missing or damaged.

Error ID: BC30655

To correct this error

- Reinstall Visual Studio.

See also

- Talk to Us

Unable to get serial port names because of an internal system error

4/9/2019 • 2 minutes to read • Edit Online

An internal error occurred when the `My.Computer.Ports.SerialPortNames` property was called.

To correct this error

1. See Debugger Basics for more troubleshooting information.
2. Note the circumstances under which the error occurred, and call Microsoft Product Support Services.

See also

- SerialPortNames
- Debugger Basics
- Talk to Us

Unable to link to resource file '<filename>': <error message>

4/9/2019 • 2 minutes to read • Edit Online

The Visual Basic compiler calls the Assembly Linker (Al.exe, also known as Alink) to generate an assembly with a manifest. The linker has reported an error linking to a native COM+ resource file from the assembly.

Error ID: BC30144

To correct this error

1. Examine the quoted error message and consult the topic Al.exe. for further explanation and advice.
2. If the error persists, gather information about the circumstances and notify Microsoft Product Support Services.

See also

- Al.exe
- Talk to Us

Unable to load information for class '<classname>'

4/9/2019 • 2 minutes to read • Edit Online

A reference was made to a class that is not available.

Error ID: BC30712

To correct this error

1. Verify that the class is defined and that you spelled the name correctly.
2. Try accessing one of the members declared in the module. In some cases, the debugging environment cannot locate members because the modules where they are declared have not been loaded yet.

See also

- Debugging in Visual Studio

Unable to write output to memory

4/9/2019 • 2 minutes to read • Edit Online

There was a problem writing output to memory.

Error ID: BC31020

To correct this error

1. Compile the program again to see if the error reoccurs.
2. If the error continues, save your work and restart Visual Studio.
3. If the error recurs, reinstall Visual Basic.
4. If the error persists after reinstallation, notify Microsoft Product Support Services.

See also

- Talk to Us

Unable to write temporary file because temporary path is not available

4/9/2019 • 2 minutes to read • Edit Online

Visual Basic could not determine the path where temporary files are stored.

Error ID: BC30698

To correct this error

1. Restart Visual Studio.
2. If the problem persists, reinstall Visual Studio.

See also

- Talk to Us

Unable to write to output file '<filename>': <error>

4/28/2019 • 2 minutes to read • Edit Online

There was a problem creating the file.

An output file cannot be opened for writing. The file (or the folder containing the file) may be opened for exclusive use by another process, or it may have its read-only attribute set.

Common situations where a file is opened exclusively are:

- The application is already running and using its files. To solve this problem, make sure that the application is not running.
- Another application has opened the file. To solve this problem, make sure that no other application is accessing the files. It is not always obvious which application is accessing your files; in that case, restarting the computer might be the easiest way to terminate the application.

If even one of the project output files is marked as read-only, this exception will be thrown.

Error ID: BC31019

To correct this error

1. Compile the program again to see if the error recurs.
2. If the error continues, save your work and restart Visual Studio.
3. If the error continues, restart the computer.
4. If the error recurs, reinstall Visual Basic.
5. If the error persists after reinstallation, notify Microsoft Product Support Services.

To check file attributes in File Explorer

1. Open the folder you are interested in.
2. Click the **Views** icon and choose **Details**.
3. Right-click the column header, and choose **Attributes** from the drop-down list.

To change the attributes of a file or folder

1. In **File Explorer**, right-click the file or folder and choose **Properties**.
2. In the **Attributes** section of the **General** tab, clear the **Read-only** box.
3. Press **OK**.

See also

- Talk to Us

Underlying type <typename> of Enum is not CLS-compliant

5/14/2019 • 2 minutes to read • Edit Online

The data type specified for this enumeration is not part of the Language Independence and Language-Independent Components (CLS). This is not an error within your component, because the .NET Framework and Visual Basic support this data type. However, another component written in strictly CLS-compliant code might not support this data type. Such a component might not be able to interact successfully with your component.

The following Visual Basic data types are not CLS-compliant:

- SByte Data Type
- UInteger Data Type
- ULong Data Type
- UShort Data Type

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC40032

To correct this error

- If your component interfaces only with other .NET Framework components, or does not interface with any other components, you do not need to change anything.
- If you are interfacing with a component not written for the .NET Framework, you might be able to determine, either through reflection or from documentation, whether it supports this data type. If it does, you do not need to change anything.
- If you are interfacing with a component that does not support this data type, you must replace it with the closest CLS-compliant type. For example, in place of `UInteger` you might be able to use `Integer` if you do not need the value range above 2,147,483,647. If you do need the extended range, you can replace `UInteger` with `Long`.
- If you are interfacing with Automation or COM objects, keep in mind that some types have different data widths than in the .NET Framework. For example, `uint` is often 16 bits in other environments. If you are passing a 16-bit argument to such a component, declare it as `UShort` instead of `UInteger` in your managed Visual Basic code.

See also

- Reflection (Visual Basic)
- Reflection

Using the iteration variable in a lambda expression may have unexpected results

4/28/2019 • 2 minutes to read • Edit Online

Using the iteration variable in a lambda expression may have unexpected results. Instead, create a local variable within the loop and assign it the value of the iteration variable.

This warning appears when you use a loop iteration variable in a lambda expression that is declared inside the loop. For example, the following example causes the warning to appear.

```
For i As Integer = 1 To 10
    ' The warning is given for the use of i.
    Dim exampleFunc As Func(Of Integer) = Function() i
Next
```

The following example shows the unexpected results that might occur.

```
Module Module1
    Sub Main()
        Dim array1 As Func(Of Integer)() = New Func(Of Integer)(4) {}

        For i As Integer = 0 To 4
            array1(i) = Function() i
        Next

        For Each funcElement In array1
            System.Console.WriteLine(funcElement())
        Next

    End Sub
End Module
```

The `For` loop creates an array of lambda expressions, each of which returns the value of the loop iteration variable `i`. When the lambda expressions are evaluated in the `For Each` loop, you might expect to see 0, 1, 2, 3, and 4 displayed, the successive values of `i` in the `For` loop. Instead, you see the final value of `i` displayed five times:

```
5
```

```
5
```

```
5
```

```
5
```

```
5
```

By default, this message is a warning. For more information about hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42324

To correct this error

- Assign the value of the iteration variable to a local variable, and use the local variable in the lambda expression.

```
Module Module1
    Sub Main()
        Dim array1 As Func(Of Integer)() = New Func(Of Integer)(4) {}

        For i As Integer = 0 To 4
            Dim j = i
            array1(i) = Function() j
        Next

        For Each funcElement In array1
            System.Console.WriteLine(funcElement())
        Next

    End Sub
End Module
```

See also

- Lambda Expressions

Value of type '<typename1>' cannot be converted to '<typename2>'

4/28/2019 • 2 minutes to read • Edit Online

Value of type '<typename1>' cannot be converted to '<typename2>'. Type mismatch could be due to the mixing of a file reference with a project reference to assembly '<assemblyname>'. Try replacing the file reference to '<filepath>' in project '<projectname1>' with a project reference to '<projectname2>'.

In a situation where a project makes both a project reference and a file reference, the compiler cannot guarantee that one type can be converted to another.

The following pseudo-code illustrates a situation that can generate this error.

```
' ================ Visual Basic project P1 ================

' P1 makes a PROJECT REFERENCE to project P2

' and a FILE REFERENCE to project P3.

Public commonObject As P3.commonClass

commonObject = P2.getCommonClass()

' ================ Visual Basic project P2 ================

' P2 makes a PROJECT REFERENCE to project P3

Public Function getCommonClass() As P3.commonClass

Return New P3.commonClass

End Function

' ================ Visual Basic project P3 ================

Public Class commonClass

End Class
```

Project `P1` makes an indirect project reference through project `P2` to project `P3`, and also a direct file reference to `P3`. The declaration of `commonObject` uses the file reference to `P3`, while the call to `P2.getCommonClass` uses the project reference to `P3`.

The problem in this situation is that the file reference specifies a file path and name for the output file of `P3` (typically p3.dll), while the project references identify the source project (`P3`) by project name. Because of this, the compiler cannot guarantee that the type `P3.commonClass` comes from the same source code through the two different references.

This situation typically occurs when project references and file references are mixed. In the preceding illustration, the problem would not occur if `P1` made a direct project reference to `P3` instead of a file reference.

Error ID: BC30955

To correct this error

- Change the file reference to a project reference.

See also

- Type Conversions in Visual Basic
- Managing references in a project

Value of type '<typename1>' cannot be converted to '<typename2>' (Multiple file references)

4/28/2019 • 2 minutes to read • Edit Online

Value of type '<typename1>' cannot be converted to '<typename2>'. Type mismatch could be due to mixing a file reference to '<filepath1>' in project '<projectname1>' with a file reference to '<filepath2>' in project '<projectname2>'. If both assemblies are identical, try replacing these references so both references are from the same location.

In a situation where a project makes more than one file reference to an assembly, the compiler cannot guarantee that one type can be converted to another.

Each file reference specifies a file path and name for the output file of a project (typically a DLL file). The compiler cannot guarantee that the output files come from the same source, or that they represent the same version of the same assembly. Therefore, it cannot guarantee that the types in the different references are the same type, or even that one can be converted to the other.

You can use a single file reference if you know that the referenced assemblies have the same assembly identity. The *assembly identity* includes the assembly's name, version, public key if any, and culture. This information uniquely identifies the assembly.

Error ID: BC30961

To correct this error

- If the referenced assemblies have the same assembly identity, then remove or replace one of the file references so that there is only a single file reference.
- If the referenced assemblies do not have the same assembly identity, then change your code so that it does not attempt to convert a type in one to a type in the other.

See also

- Type Conversions in Visual Basic
- Managing references in a project

Value of type 'type1' cannot be converted to 'type2'

4/28/2019 • 2 minutes to read • Edit Online

Value of type 'type1' cannot be converted to 'type2'. You can use the 'Value' property to get the string value of the first element of '<parentElement>'.

An attempt has been made to implicitly cast an XML literal to a specific type. The XML literal cannot be implicitly cast to the specified type.

Error ID: BC31194

To correct this error

- Use the `Value` property of the XML literal to reference its value as a `String`. Use the `CType` function, another type conversion function, or the Convert class to cast the value as the specified type.

See also

- Convert
- Type Conversion Functions
- XML Literals
- XML

Variable '<variablename>' hides a variable in an enclosing block

4/28/2019 • 2 minutes to read • Edit Online

A variable enclosed in a block has the same name as another local variable.

Error ID: BC30616

To correct this error

- Rename the variable in the enclosed block so that it is not the same as any other local variables. For example:

```
Dim a, b, x As Integer
If a = b Then
   Dim y As Integer = 20 ' Uniquely named block variable.
End If
```

- A common cause for this error is the use of `Catch e As Exception` inside an event handler. If this is the case, name the `Catch` block variable `ex` rather than `e`.
- Another common source of this error is an attempt to access a local variable declared within a `Try` block in a separate `Catch` block. To correct this, declare the variable outside the `Try...Catch...Finally` structure.

See also

- Try...Catch...Finally Statement
- Variable Declaration

Variable '<variablename>' is used before it has been assigned a value

4/28/2019 • 2 minutes to read • Edit Online

Variable '<variablename>' is used before it has been assigned a value. A null reference exception could result at run time.

An application has at least one possible path through its code that reads a variable before any value is assigned to it.

If a variable has never been assigned a value, it holds the default value for its data type. For a reference data type, that default value is Nothing. Reading a reference variable that has a value of `Nothing` can cause a NullReferenceException in some circumstances.

By default, this message is a warning. For more information on hiding warnings or treating warnings as errors, see Configuring Warnings in Visual Basic.

Error ID: BC42104

To correct this error

- Check your control flow logic and make sure the variable has a valid value before control passes to any statement that reads it.
- One way to guarantee that the variable always has a valid value is to initialize it as part of its declaration. See "Initialization" in Dim Statement.

See also

- Dim Statement
- Variable Declaration
- Troubleshooting Variables

Variable uses an Automation type not supported in Visual Basic

3/5/2019 • 2 minutes to read • Edit Online

You tried to use a variable defined in a type library or object library that has a data type not supported by Visual Basic.

To correct this error

- Use a variable of a type recognized by Visual Basic.

  -or-

- If you encounter this error while using `FileGet` or `FileGetObject`, make sure the file you are trying to use was written to with `FilePut` or `FilePutObject`.

See also

- Data Types

XML axis properties do not support late binding

4/28/2019 • 2 minutes to read • Edit Online

An XML axis property has been referenced for an untyped object.

Error ID: BC31168

To correct this error

- Ensure that the object is a strong-typed XElement object before referencing the XML axis property.

See also

- XML Axis Properties
- XML

XML comment exception must have a 'cref' attribute

4/28/2019 • 2 minutes to read • Edit Online

The <exception> tag provides a way to document the exceptions that may be thrown by a method. The required `cref` attribute designates the name of a member, which is checked by the documentation generator. If the member exists, it is translated to the canonical element name in the documentation file.

Error ID: BC42319

To correct this error

- Add the `cref` attribute to the exception as follows:

```
'''<exception cref="member">description</exception>
```

See also

- <exception>
- How to: Create XML Documentation
- XML Comment Tags

XML entity references are not supported

4/28/2019 • 2 minutes to read • Edit Online

An entity reference (for example, `©`) that is not defined in the XML 1.0 specification is included as a value for an XML literal. Only `&`, `"`, `<`, `>`, and `'` XML entity references are supported in XML literals.

Error ID: BC31180

To correct this error

- Remove the unsupported entity reference.

See also

- XML Literals and the XML 1.0 Specification
- XML Literals
- XML

XML literals and XML properties are not supported in embedded code within ASP.NET

4/28/2019 • 2 minutes to read • Edit Online

XML literals and XML properties are not supported in embedded code within ASP.NET. To use XML features, move the code to code-behind.

An XML literal or XML axis property is defined within embedded code (`<%= =>`) in an ASP.NET file.

Error ID: BC31200

To correct this error

- Move the code that includes the XML literal or XML axis property to an ASP.NET code-behind file.

See also

- XML Literals
- XML Axis Properties
- XML

XML namespace URI `http://www.w3.org/XML/1998/namespace` can be bound only to 'xmlns'

1/30/2019 • 2 minutes to read • Edit Online

The URI `http://www.w3.org/XML/1998/namespace` is used in an XML namespace declaration. This URI is a reserved namespace and cannot be included in an XML namespace declaration.

Error ID: BC31183

To correct this error

Remove the XML namespace declaration or replace the URI `http://www.w3.org/XML/1998/namespace` with a valid namespace URI.

See also

- Imports Statement (XML Namespace)
- XML Literals
- XML

Reference (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

This section provides links to reference information about various aspects of Visual Basic programming.

In This Section

Visual Basic Language Reference
Provides reference information for various aspects of the Visual Basic language.

Visual Basic Command-Line Compiler
Provides links to information about the command-line compiler, which provides an alternative to compiling programs from the Visual Studio IDE.

.NET Framework Reference Information
Provides links to information on working with the .NET Framework class library.

Visual Basic Language Specification
Provides links to the complete Visual Basic language specification, which contains detailed information on all aspects of the language.

Related Sections

General User Interface Elements (Visual Studio)
Contains topics for dialog boxes and windows used in Visual Studio.

XML Tools in Visual Studio
Provides links to topics on various XML tools available in Visual Studio.

Automation and Extensibility Reference
Provides links to topics covering automation and extensibility in Visual Studio, for both shared and language-specific components.

Visual Basic command-line compiler

10/24/2018 • 2 minutes to read • Edit Online

The Visual Basic command-line compiler provides an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). This section contains descriptions for the Visual Basic compiler options.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

In this section

Building from the Command Line
Describes the Visual Basic command-line compiler, which is provided as an alternative to compiling programs from within the Visual Studio IDE.

Visual Basic Compiler Options Listed Alphabetically
Lists compiler options in an alphabetical table

Visual Basic Compiler Options Listed by Category
Presents compiler options in functional groups.

Related sections

Visual Basic Guide
The starting point for the Visual Basic documentation.

Building from the Command Line (Visual Basic)

8/27/2019 • 2 minutes to read • Edit Online

A Visual Basic project is made up of one or more separate source files. During the process known as compilation, these files are brought together into one package—a single executable file that can be run as an application.

Visual Basic provides a command-line compiler as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). The command-line compiler is designed for situations in which you do not require the full set of features in the IDE—for example, when you are using or writing for computers with limited system memory or storage space.

To compile source files from within the Visual Studio IDE, choose the **Build** command from the **Build** menu.

> **TIP**
> When you build project files by using the Visual Studio IDE, you can display information about the associated **vbc** command and its switches in the output window. To display this information, open the Options Dialog Box, Projects and Solutions, Build and Run, and then set the **MSBuild project build output verbosity** to **Normal** or a higher level of verbosity. For more information, see How to: View, Save, and Configure Build Log Files.

You can compile project (.vbproj) files at a command prompt by using MSBuild. For more information, see Command-Line Reference and Walkthrough: Using MSBuild.

In This Section

How to: Invoke the Command-Line Compiler
Describes how to invoke the command-line compiler at the MS-DOS prompt or from a specific subdirectory.

Sample Compilation Command Lines
Provides a list of sample command lines that you can modify for your own use.

Related Sections

Visual Basic Command-Line Compiler
Provides lists of compiler options, organized alphabetically or by purpose.

Conditional Compilation
Describes how to compile particular sections of code.

Building and Cleaning Projects and Solutions in Visual Studio
Describes how to organize what will be included in different builds, choose project properties, and ensure that projects build in the correct order.

How to: Invoke the Command-Line Compiler (Visual Basic)

4/9/2019 • 2 minutes to read • Edit Online

You can invoke the command-line compiler by typing the name of its executable file into the command line, also known as the MS-DOS prompt. If you compile from the default Windows Command Prompt, you must type the fully qualified path to the executable file. To override this default behavior, you can either use the Developer Command Prompt for Visual Studio, or modify the PATH environment variable. Both allow you to compile from any directory by simply typing the compiler name.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To invoke the compiler using the Developer Command Prompt for Visual Studio

1. Open the Visual Studio Tools program folder within the Microsoft Visual Studio program group.
2. You can use the Developer Command Prompt for Visual Studio to access the compiler from any directory on your machine, if Visual Studio is installed.
3. Invoke the Developer Command Prompt for Visual Studio.
4. At the command line, type `vbc.exe` *sourceFileName* and then press ENTER.

   For example, if you stored your source code in a directory called `SourceFiles`, you would open the Command Prompt and type `cd SourceFiles` to change to that directory. If the directory contained a source file named `Source.vb`, you could compile it by typing `vbc.exe Source.vb`.

To set the PATH environment variable to the compiler for the Windows Command Prompt

1. Use the Windows Search feature to find Vbc.exe on your local disk.

   The exact name of the directory where the compiler is located depends on the location of the Windows directory and the version of the ".NET Framework" installed. If you have more than one version of the ".NET Framework" installed, you must determine which version to use (typically the latest version).
2. From your **Start** Menu, right-click **My Computer**, and then click **Properties** from the shortcut menu.
3. Click the **Advanced** tab, and then click **Environment Variables**.
4. In the **System** variables pane, select **Path** from the list and click **Edit**.
5. In the **Edit System** Variable dialog box, move the insertion point to the end of the string in the **Variable Value** field and type a semicolon (;) followed by the full directory name found in Step 1.
6. Click **OK** to confirm your edits and close the dialog boxes.

   After you change the PATH environment variable, you can run the Visual Basic compiler at the Windows Command Prompt from any directory on the computer.

To invoke the compiler using the Windows Command Prompt

1. From the **Start** menu, click on the **Accessories** folder, and then open the **Windows Command Prompt**.

2. At the command line, type `vbc.exe` *sourceFileName* and then press ENTER.

   For example, if you stored your source code in a directory called `SourceFiles`, you would open the Command Prompt and type `cd SourceFiles` to change to that directory. If the directory contained a source file named `Source.vb`, you could compile it by typing `vbc.exe Source.vb`.

See also

- Visual Basic Command-Line Compiler
- Conditional Compilation

Sample compilation command lines (Visual Basic)

8/27/2019 • 2 minutes to read • Edit Online

As an alternative to compiling Visual Basic programs from within Visual Studio, you can compile from the command line to produce executable (.exe) files or dynamic-link library (.dll) files.

The Visual Basic command-line compiler supports a complete set of options that control input and output files, assemblies, and debug and preprocessor options. Each option is available in two interchangeable forms: `-option` and `/option`. This documentation shows only the `-option` form.

The following table lists some sample command lines you can modify for your own use.

TO	USE
Compile File.vb and create File.exe	`vbc -reference:Microsoft.VisualBasic.dll File.vb`
Compile File.vb and create File.dll	`vbc -target:library File.vb`
Compile File.vb and create My.exe	`vbc -out:My.exe File.vb`
Compile File.vb and create both a library and a reference assembly named File.dll	`vbc -target:library -ref:.\debug\bin\ref\file.dll File.vb`
Compile all Visual Basic files in the current directory, with optimizations on and the `DEBUG` symbol defined, producing File2.exe	`vbc -define:DEBUG=1 -optimize -out:File2.exe *.vb`
Compile all Visual Basic files in the current directory, producing a debug version of File2.dll without displaying the logo or warnings	`vbc -target:library -out:File2.dll -nowarn -nologo -debug *.vb`
Compile all Visual Basic files in the current directory to Something.dll	`vbc -target:library -out:Something.dll *.vb`

TIP

When you build a project by using the Visual Studio IDE, you can display information about the associated **vbc** command with its compiler options in the output window. To display this information, open the Options Dialog Box, Projects and Solutions, Build and Run, and then set the **MSBuild project build output verbosity** to **Normal** or a higher level of verbosity.

See also

- Visual Basic Command-Line Compiler
- Conditional Compilation

Visual Basic compiler options listed alphabetically

7/19/2019 • 3 minutes to read • Edit Online

The Visual Basic command-line compiler is provided as an alternative to compiling programs from the Visual Studio integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted alphabetically.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

OPTION	PURPOSE
@ (Specify Response File)	Specifies a response file.
-?	Displays compiler options. This command is the same as specifying the `-help` option. No compilation occurs.
`-additionalfile`	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.
-addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
`-analyzer`	Run the analyzers from this assembly (Short form: -a)
-baseaddress	Specifies the base address of a DLL.
-bugreport	Creates a file that contains information that makes it easy to report a bug.
`-checksumalgorithm:<alg>`	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256. Due to collision problems with SHA1, Microsoft recommends SHA256 or better.
-codepage	Specifies the code page to use for all source code files in the compilation.
-debug	Produces debugging information.
-define	Defines symbols for conditional compilation.
-delaysign	Specifies whether the assembly will be fully or partially signed.
-deterministic	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
-doc	Processes documentation comments to an XML file.

OPTION	PURPOSE
-errorreport	Specifies how the Visual Basic compiler should report internal compiler errors.
-filealign	Specifies where to align the sections of the output file.
-help	Displays compiler options. This command is the same as specifying the `-?` option. No compilation occurs.
-highentropyva	Indicates whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).
-imports	Imports a namespace from a specified assembly.
-keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
-keyfile	Specifies a file that contains a key or key pair to give an assembly a strong name.
-langversion	Specify language version: 9|9.0|10|10.0|11|11.0.
-libpath	Specifies the location of assemblies referenced by the -reference option.
-linkresource	Creates a link to a managed resource.
-main	Specifies the class that contains the `Sub Main` procedure to use at startup.
-moduleassemblyname	Specifies the name of the assembly that a module will be a part of.
`-modulename:<string>`	Specify the name of the source module
-netcf	Sets the compiler to target the .NET Compact Framework.
-noconfig	Do not compile with Vbc.rsp.
-nologo	Suppresses compiler banner information.
-nostdlib	Causes the compiler not to reference the standard libraries.
-nowarn	Suppresses the compiler's ability to generate warnings.
-nowin32manifest	Instructs the compiler not to embed any application manifest into the executable file.
-optimize	Enables/disables code optimization.
-optioncompare	Specifies whether string comparisons should be binary or use locale-specific text semantics.

OPTION	PURPOSE
-optionexplicit	Enforces explicit declaration of variables.
-optioninfer	Enables the use of local type inference in variable declarations.
-optionstrict	Enforces strict language semantics.
-out	Specifies an output file.
`-parallel[+|-]`	Specifies whether to use concurrent build (+).
-platform	Specifies the processor platform the compiler targets for the output file.
`-preferreduilang`	Specify the preferred output language name.
-quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
-recurse	Searches subdirectories for source files to compile.
-reference	Imports metadata from an assembly.
-refonly	Outputs only a reference assembly.
-refout	Specifies the output path of a reference assembly.
-removeintchecks	Disables integer overflow checking.
-resource	Embeds a managed resource in an assembly.
-rootnamespace	Specifies a namespace for all type declarations.
`-ruleset:<file>`	Specify a ruleset file that disables specific diagnostics.
-sdkpath	Specifies the location of Mscorlib.dll and Microsoft.VisualBasic.dll.
-subsystemversion	Specifies the minimum version of the subsystem that the generated executable file can use.
-target	Specifies the format of the output file.
-utf8output	Displays compiler output using UTF-8 encoding.
-vbruntime	Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.
-verbose	Outputs extra information during compilation.
-warnaserror	Promotes warnings to errors.

OPTION	PURPOSE
-win32icon	Inserts an .ico file into the output file.
-win32manifest	Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.
-win32resource	Inserts a Win32 resource into the output file.

See also

- Visual Basic Compiler Options Listed by Category
- Manage project and solution properties

@ (Specify Response File) (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies a file that contains compiler options and source-code files to compile.

Syntax

```
@response_file
```

Arguments

`response_file`
Required. A file that lists compiler options or source-code files to compile. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

The compiler processes the compiler options and source-code files specified in a response file as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response-file options, such as the following.

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source-code files can appear on one line. A single compiler-option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the `#` symbol.

You can combine options specified on the command line with options specified in one or more response files. The compiler processes the command options as it encounters them. Therefore, command-line arguments can override previously listed options in response files. Conversely, options in a response file override options listed previously on the command line or in other response files.

Visual Basic provides the Vbc.rsp file, which is located in the same directory as the Vbc.exe file. The Vbc.rsp file is included by default unless the `-noconfig` option is used. For more information, see -noconfig.

> **NOTE**
> The `@` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following lines are from a sample response file.

```
# build the first output file
-target:exe
-out:MyExe.exe
source1.vb
source2.vb
```

Example

The following example demonstrates how to use the `@` option with the response file named `File1.rsp`.

```
vbc @file1.rsp
```

See also

- Visual Basic Command-Line Compiler
- -noconfig
- Sample Compilation Command Lines

-addmodule

8/22/2019 • 2 minutes to read • Edit Online

Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.

Syntax

```
-addmodule:fileList
```

Arguments

`fileList`

Required. Comma-delimited list of files that contain metadata but do not contain assembly manifests. File names containing spaces should be surrounded by quotation marks (" ").

Remarks

The files listed by the `fileList` parameter must be created with the `-target:module` option, or with another compiler's equivalent to `-target:module`.

All modules added with `-addmodule` must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time, but the module must be in the application directory at run time. If it is not, you get a TypeLoadException error.

If you specify (implicitly or explicitly) any-target (Visual Basic) option other than `-target:module` with `-addmodule`, the files you pass to `-addmodule` become part of the project's assembly. An assembly is required to run an output file that has one or more files added with `-addmodule`.

Use /reference (Visual Basic) to import metadata from a file that contains an assembly.

> **NOTE**
> The `-addmodule` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code creates a module.

```
' t1.vb
' Compile with vbc /target:module t1.vb.
' Outputs t1.netmodule.

Public Class TestClass
    Public i As Integer
End Class
```

The following code imports the module's types.

```
' t2.vb
' Compile with vbc /addmodule:t1.netmodule t2.vb.
Option Strict Off

Namespace NetmoduleTest
    Module Module1
        Sub Main()
            Dim x As TestClass
            x = New TestClass
            x.i = 802
            System.Console.WriteLine(x.i)
        End Sub
    End Module
End Namespace
```

When you run `t1`, it outputs `802`.

See also

- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- -reference (Visual Basic)
- Sample Compilation Command Lines

-baseaddress

5/14/2019 • 2 minutes to read • Edit Online

Specifies a default base address when creating a DLL.

Syntax

```
-baseaddress:address
```

Arguments

TERM	DEFINITION
`address`	Required. The base address for the DLL. This address must be specified as a hexadecimal number.

Remarks

The default base address for a DLL is set by the .NET Framework.

Be aware that the lower-order word in this address is rounded. For example, if you specify 0x11110001, it is rounded to 0x11110000.

To complete the signing process for a DLL, use the `-R` option of the Strong Naming tool (Sn.exe).

This option is ignored if the target is not a DLL.

TO SET -BASEADDRESS IN THE VISUAL STUDIO IDE
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Compile** tab. 3. Click **Advanced**. 4. Modify the value in the **DLL base address:** box. **Note:** The **DLL base address:** box is read-only unless the target is a DLL.

See also

- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- Sample Compilation Command Lines
- Sn.exe (Strong Name Tool))

-bugreport

8/22/2019 • 2 minutes to read • Edit Online

Creates a file that you can use when you file a bug report.

Syntax

```
-bugreport:file
```

Arguments

TERM	DEFINITION
`file`	Required. The name of the file that will contain your bug report. Enclose the file name in quotation marks (" ") if the name contains a space.

Remarks

The following information is added to `file`:

- A copy of all source-code files in the compilation.
- A list of the compiler options used in the compilation.
- Version information about your compiler, common language runtime, and operating system.
- Compiler output, if any.
- A description of the problem, for which you are prompted.
- A description of how you think the problem should be fixed, for which you are prompted.

Because a copy of all source-code files is included in `file`, you may want to reproduce the (suspected) code defect in the shortest possible program.

> **IMPORTANT**
> The `-bugreport` option produces a file that contains potentially sensitive information. This includes current time, compiler version, .NET Framework version, OS version, user name, the command-line arguments with which the compiler was run, all source code, and the binary form of any referenced assembly. This option can be accessed by specifying command-line options in the Web.config file for a server-side compilation of an ASP.NET application. To prevent this, modify the Machine.config file to disallow users from compiling on the server.

If this option is used with `-errorreport:prompt`, `-errorreport:queue`, or `-errorreport:send`, and your application encounters an internal compiler error, the information in `file` is sent to Microsoft Corporation. That information will help Microsoft engineers identify the cause of the error and may help improve the next release of Visual Basic. By default, no information is sent to Microsoft. However, when you compile an application by using `-errorreport:queue`, which is enabled by default, the application collects its error reports. Then, when the computer's administrator logs in, the error reporting system displays a pop-up window that enables the

administrator to forward to Microsoft any error reports that occurred since the logon.

> **NOTE**
>
> The `/bugreport` option is not available from within the Visual Studio development environment; it is available only when you compile from the command line.

Example

The following example compiles `T2.vb` and puts all bug-reporting information in the file `Problem.txt`.

```
vbc -bugreport:problem.txt t2.vb
```

See also

- Visual Basic Command-Line Compiler
- -debug (Visual Basic)
- -errorreport
- Sample Compilation Command Lines
- trustLevel Element for securityPolicy (ASP.NET Settings Schema)

-codepage (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Specifies the code page to use for all source-code files in the compilation.

Syntax

```
-codepage:id
```

Arguments

TERM	DEFINITION
`id`	Required. The compiler uses the code page specified by `id` to interpret the encoding of the source files.

Remarks

To compile source code saved with a specific encoding, you can use `-codepage` to specify which code page should be used. The `-codepage` option applies to all source-code files in your compilation. For more information, see Character Encoding in the .NET Framework.

The `-codepage` option is not needed if the source-code files were saved using the current ANSI code page, Unicode, or UTF-8 with a signature. Visual Studio saves all source-code files with the current ANSI code page by default, unless the user specifies another encoding in the **Encoding** dialog box. Visual Studio uses the **Encoding** dialog box to open source-code files saved with a different code page.

> **NOTE**
>
> The `-codepage` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- Visual Basic Command-Line Compiler

-debug (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Causes the compiler to generate debugging information and place it in the output file(s).

Syntax

```
-debug[+ | -]
' -or-
-debug:[full | pdbonly]
```

Arguments

TERM	DEFINITION
`+` | `-`	Optional. Specifying `+` or `/debug` causes the compiler to generate debugging information and place it in a .pdb file. Specifying `-` has the same effect as not specifying `/debug`.
`full` | `pdbonly`	Optional. Specifies the type of debugging information generated by the compiler. If you do not specify `/debug:pdbonly`, the default is `full`, which enables you to attach a debugger to the running program. The `pdbonly` argument allows source-code debugging when the program is started in the debugger, but it displays assembly-language code only when the running program is attached to the debugger.

Remarks

Use this option to create debug builds. If you do not specify `/debug`, `/debug+`, or `/debug:full`, you will be unable to debug the output file of your program.

By default, debugging information is not emitted (`/debug-`). To emit debugging information, specify `/debug` or `/debug+`.

For information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

TO SET -DEBUG IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**. 2. Click the **Compile** tab. 3. Click **Advanced Compile Options**. 4. Modify the value in the **Generate Debug Info** box.

Example

The following example puts debugging information in output file `App.exe`.

```
vbc -debug -out:app.exe test.vb
```

See also

- Visual Basic Command-Line Compiler
- /bugreport
- Sample Compilation Command Lines

-define (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Defines conditional compiler constants.

Syntax

```
-define:["]symbol[=value][,symbol[=value]]["]
' -or-
-d:["]symbol[=value][,symbol[=value]]["]
```

Arguments

TERM	DEFINITION
`symbol`	Required. The symbol to define.
`value`	Optional. The value to assign `symbol`. If `value` is a string, it must be surrounded by backslash/quotation-mark sequences (\") instead of quotation marks. If no value is specified, then it is taken to be True.

Remarks

The `-define` option has an effect similar to using a `#Const` preprocessor directive in your source file, except that constants defined with `-define` are public and apply to all files in the project.

You can use symbols created by this option with the `#If` ... `Then` ... `#Else` directive to compile source files conditionally.

`-d` is the short form of `-define`.

You can define multiple symbols with `-define` by using a comma to separate symbol definitions.

TO SET /DEFINE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Compile** tab. 3. Click **Advanced**. 4. Modify the value in the **Custom Constants** box.

Example

The following code defines and then uses two conditional compiler constants.

```
    ' Vbc /define:DEBUGMODE=True,TRAPERRORS=False test.vb
    Sub mysub()
#If debugmode Then
        ' Insert debug statements here.
         MsgBox("debug mode")
#Else
        ' Insert default statements here.
#End If
    End Sub
```

See also

- Visual Basic Command-Line Compiler
- #If...Then...#Else Directives
- #Const Directive
- Sample Compilation Command Lines

-delaysign

4/9/2019 • 2 minutes to read • Edit Online

Specifies whether the assembly will be fully or partially signed.

Syntax

```
-delaysign[+ | -]
```

Arguments

`+` | `-`

Optional. Use `-delaysign-` if you want a fully signed assembly. Use `-delaysign+` if you want to place the public key in the assembly and reserve space for the signed hash. The default is `-delaysign-`.

Remarks

The `-delaysign` option has no effect unless used with -keyfile or -keycontainer.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the compiler does not compute and store the signature but reserves space in the file so the signature can be added later.

For example, by using `-delaysign+`, a developer in an organization can distribute unsigned test versions of an assembly that testers can register with the global assembly cache and use. When work on the assembly is completed, the person responsible for the organization's private key can fully sign the assembly. This compartmentalization protects the organization's private key from disclosure, while allowing all developers to work on the assemblies.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

To set -delaysign in the Visual Studio integrated development environment

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Signing** tab.
3. Set the value in the **Delay sign only** box.

See also

- Visual Basic Command-Line Compiler
- -keyfile
- -keycontainer
- Sample Compilation Command Lines

-deterministic

4/11/2019 • 2 minutes to read • Edit Online

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

Syntax

```
-deterministic
```

Remarks

By default, compiler output from a given set of inputs is unique, since the compiler adds a timestamp and a GUID that is generated from random numbers. You use the `-deterministic` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same.

The compiler considers the following inputs for the purpose of determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
  - Source files
  - Referenced assemblies
  - Referenced modules
  - Resources
  - The strong name key file
  - @ response files
  - Analyzers
  - Rulesets
  - Additional files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding is not specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `/lib` or `/recurse`).
- The CLR platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

When sources are publicly available, deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. It can also be useful in a continuous build system for determining whether build steps that are dependent on changes to a binary need to be executed.

See also

- Visual Basic Command-Line Compiler

- Sample Compilation Command Lines

-doc

4/28/2019 • 2 minutes to read • Edit Online

Processes documentation comments to an XML file.

Syntax

```
-doc[+ | -]
' -or-
-doc:file
```

Arguments

TERM	DEFINITION
`+` | `-`	Optional. Specifying +, or just `-doc`, causes the compiler to generate documentation information and place it in an XML file. Specifying `-` is the equivalent of not specifying `-doc`, causing no documentation information to be created.
`file`	Required if `-doc:` is used. Specifies the output XML file, which is populated with the comments from the source-code files of the compilation. If the file name contains a space, surround the name with quotation marks (" ").

Remarks

The `-doc` option controls whether the compiler generates an XML file containing the documentation comments. If you use the `-doc:file` syntax, the `file` parameter specifies the name of the XML file. If you use `-doc` or `-doc+`, the compiler takes the XML file name from the executable file or library that the compiler is creating. If you use `-doc-` or do not specify the `-doc` option, the compiler does not create an XML file.

In source-code files, documentation comments can precede the following definitions:

- User-defined types, such as a class or interface
- Members, such as a field, event, property, function, or subroutine.

To use the generated XML file with the Visual Studio IntelliSense feature, let the file name of the XML file be the same as the assembly you want to support. Make sure the XML file is in the same directory as the assembly so that when the assembly is referenced in the Visual Studio project, the .xml file is found as well. XML documentation files are not required for IntelliSense to work for code within a project or within projects referenced by a project.

Unless you compile with `/target:module`, the XML file contains the tags `<assembly></assembly>`. These tags specify the name of the file containing the assembly manifest for the output file of the compilation.

See XML Comment Tags for ways to generate documentation from comments in your code.

TO SET -DOC IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Set the value in the **Generate XML documentation file** box.

Example

See Documenting Your Code with XML for a sample.

See also

- Visual Basic Command-Line Compiler
- Documenting Your Code with XML

-errorreport

10/24/2018 • 2 minutes to read • Edit Online

Specifies how the Visual Basic compiler should report internal compiler errors.

Syntax

```
-errorreport:{ prompt | queue | send | none }
```

Remarks

This option provides a convenient way to report a Visual Basic internal compiler error (ICE) to the Visual Basic team at Microsoft. By default, the compiler sends no information to Microsoft. However, if you do encounter an internal compiler error, this option allows you to report the error to Microsoft. That information will help Microsoft engineers identify the cause and may help improve the next release of Visual Basic.

A user's ability to send reports depends on machine and user policy permissions.

The following table summarizes the effect of the `-errorreport` option.

OPTION	BEHAVIOR
`prompt`	If an internal compiler error occurs, a dialog box comes up so that you can view the exact data that the compiler collected. You can determine if there is any sensitive information in the error report and make a decision on whether to send it to Microsoft. If you decide to send it, and the machine and user policy settings allow it, the compiler sends the data to Microsoft.
`queue`	Queues the error report. When you log in with administrator privileges, you can report any failures since the last time you were logged in (you will not be prompted to send reports for failures more than once every three days). This is the default behavior when the `-errorreport` option is not specified.
`send`	If an internal compiler error occurs, and the machine and user policy settings allow it, the compiler sends the data to Microsoft. The option `-errorreport:send` attempts to automatically send error information to Microsoft if reporting is enabled by the Windows Error Reporting system settings.
`none`	If an internal compiler error occurs, it will not be collected or sent to Microsoft.

The compiler sends data that includes the stack at the time of the error, which usually includes some source code. If `-errorreport` is used with the -bugreport option, then the entire source file is sent.

This option is best used with the /bugreport option, because it allows Microsoft engineers to more easily reproduce the error.

> **NOTE**
>
> The `-errorreport` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code attempts to compile `T2.vb`, and if the compiler encounters an internal compiler error, it prompts you to send the error report to Microsoft.

```
vbc -errorreport:prompt t2.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- -bugreport

-filealign

8/22/2019 • 2 minutes to read • Edit Online

Specifies where to align the sections of the output file.

Syntax

```
-filealign:number
```

Arguments

`number`
Required. A value that specifies the alignment of sections in the output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

Remarks

You can use the `-filealign` option to specify the alignment of sections in your output file. Sections are blocks of contiguous memory in a Portable Executable (PE) file that contains either code or data. The `-filealign` option lets you compile your application with a nonstandard alignment; most developers do not need to use this option.

Each section is aligned on a boundary that is a multiple of the `-filealign` value. There is no fixed default. If `-filealign` is not specified, the compiler picks a default at compile time.

By specifying the section size, you can change the size of the output file. Modifying section size may be useful for programs that will run on smaller devices.

> **NOTE**
> The `-filealign` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- Visual Basic Command-Line Compiler

-help, -? (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Displays the compiler options.

Syntax

```
-help
' -or-
-?
```

Remarks

If you include this option in a compilation, no output file is created and no compilation takes place.

> **NOTE**
> The `-help` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code displays help from the command line.

```
vbc -help
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-highentropyva (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Indicates whether a 64-bit executable or an executable that's marked by the /platform:anycpu compiler option supports high entropy Address Space Layout Randomization (ASLR).

Syntax

```
-highentropyva[+ | -]
```

Arguments

`+` | `-`

Optional. The option is off by default or if you specify `-highentropyva-`. The option is on if you specify `-highentropyva` or `-highentropyva+`.

Remarks

If you specify this option, compatible versions of the Windows kernel can use higher degrees of entropy when the kernel randomizes the address space layout of a process as part of ASLR. If the kernel uses higher degrees of entropy, a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it is more difficult to guess the location of a particular memory region.

When the option is on, the target executable and any modules on which it depends must be able to handle pointer values that are larger than 4 gigabytes (GB) when those modules are running as 64-bit processes.

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-imports (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Imports namespaces from a specified assembly.

Syntax

```
-imports:namespaceList
```

Arguments

TERM	DEFINITION
`namespaceList`	Required. Comma-delimited list of namespaces to be imported.

Remarks

The `-imports` option imports any namespace defined within the current set of source files or from any referenced assembly.

The members in a namespace specified with `-imports` are available to all source-code files in the compilation. Use the Imports Statement (.NET Namespace and Type) to use a namespace in a single source-code file.

TO SET /IMPORTS IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **References** tab. 3. Enter the namespace name in the box beside the **Add User Import** button. 4. Click the **Add User Import** button.

Example

The following code compiles when `/imports:system.globalization` is specified. Without it, successful compilation requires either that an `Imports System.Globalization` statement be included at the beginning of the source code file, or that the property be fully qualified as `System.Globalization.CultureInfo.CurrentCulture.Name`.

```
Module Example
   Public Sub Main()
      Console.WriteLine($"The current culture is {CultureInfo.CurrentCulture.Name}")
   End Sub
End Module
```

See also

- Visual Basic Command-Line Compiler
- References and the Imports Statement
- Sample Compilation Command Lines

-keycontainer

8/22/2019 • 2 minutes to read • Edit Online

Specifies a key container name for a key pair to give an assembly a strong name.

Syntax

```
-keycontainer:container
```

Arguments

TERM	DEFINITION
`container`	Required. Container file that contains the key. Enclose the file name in quotation marks ("") if the name contains a space.

Remarks

The compiler creates the sharable component by inserting a public key into the assembly manifest and by signing the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. The `-i` option installs the key pair into a container. For more information, see Sn.exe (Strong Name Tool)).

If you compile with `-target:module`, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with -addmodule.

You can also specify this option as a custom attribute (AssemblyKeyNameAttribute) in the source code for any Microsoft intermediate language (MSIL) module.

You can also pass your encryption information to the compiler with -keyfile. Use -delaysign if you want a partially signed assembly.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

> **NOTE**
> The `-keycontainer` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles source file `Input.vb` and specifies a key container.

```
vbc -keycontainer:key1 input.vb
```

See also

- Assemblies in .NET
- Visual Basic Command-Line Compiler

- -keyfile
- Sample Compilation Command Lines

-keyfile

8/22/2019 • 2 minutes to read • Edit Online

Specifies a file containing a key or key pair to give an assembly a strong name.

Syntax

```
-keyfile:file
```

Arguments

`file`
Required. File that contains the key. If the file name contains a space, enclose the name in quotation marks (" ").

Remarks

The compiler inserts the public key into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. For more information, see Sn.exe (Strong Name Tool)).

If you compile with `-target:module`, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with /addmodule.

You can also pass your encryption information to the compiler with -keycontainer. Use -delaysign if you want a partially signed assembly.

You can also specify this option as a custom attribute (AssemblyKeyFileAttribute) in the source code for any Microsoft intermediate language module.

In case both `-keyfile` and -keycontainer are specified (either by command-line option or by custom attribute) in the same compilation, the compiler first tries the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler does not find the key container, it tries the file specified with `-keyfile`. If this succeeds, the assembly is signed with the information in the key file, and the key information is installed in the key container (similar to `sn -i`) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

> **NOTE**
> The `-keyfile` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles source file `Input.vb` and specifies a key file.

```
vbc -keyfile:myfile.sn input.vb
```

See also

- Assemblies in .NET
- Visual Basic Command-Line Compiler
- -reference (Visual Basic)
- Sample Compilation Command Lines

-langversion (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Causes the compiler to accept only syntax that is included in the specified Visual Basic language version.

Syntax

```
-langversion:version
```

Arguments

`version`
Required. The language version to be used during the compilation. Accepted values are `9`, `10`, `11`, `12`, `14`, `15`, `15.3`, `15.5`, `default` and `latest`.

Any of the whole numbers may also be specified using `.0` as the minor version, for example, `11.0`.

You can see the list of all possible values by specifying `-langversion:?` on the command line.

Remarks

The `-langversion` option specifies what syntax the compiler accepts. For example, if you specify that the language version is 9.0, the compiler generates errors for syntax that is valid only in version 10.0 and later.

You can use this option when you develop applications that target different versions of the .NET Framework. For example, if you are targeting .NET Framework 3.5, you could use this option to ensure that you do not use syntax from language version 10.0.

You can set `-langversion` directly only by using the command line. For more information, see Targeting a Specific .NET Framework Version.

Example

The following code compiles `sample.vb` for Visual Basic 9.0.

```
vbc -langversion:9.0 sample.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- Targeting a Specific .NET Framework Version

-libpath

4/9/2019 • 2 minutes to read • Edit Online

Specifies the location of referenced assemblies.

Syntax

```
-libpath:dirList
```

Arguments

TERM	DEFINITION
`dirList`	Required. Semicolon-delimited list of directories for the compiler to look in if a referenced assembly is not found in either the current working directory (the directory from which you are invoking the compiler) or the common language runtime's system directory. If the directory name contains a space, enclose the name in quotation marks (" ").

Remarks

The `-libpath` option specifies the location of assemblies referenced by the -reference option.

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by `/libpath`.
4. Directories specified by the LIB environment variable.

The `-libpath` option is additive; specifying it more than once appends to any prior values.

Use `-reference` to specify an assembly reference.

TO SET /LIBPATH IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **References** tab. 3. Click the **Reference Paths...** button. 4. In the **Reference Paths** dialog box, enter the directory name in the **Folder:** box. 5. Click **Add Folder**.

Example

The following code compiles `T2.vb` to create an .exe file. The compiler looks in the working directory, in the root directory of the C: drive, and in the New Assemblies directory of the C: drive for assembly references.

```
vbc -libpath:c:\;"c:\New Assemblies" -reference:t2.dll t2.vb
```

See also

- Assemblies in .NET
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-link (Visual Basic)

8/22/2019 • 3 minutes to read • Edit Online

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

Syntax

```
-link:fileList
' -or-
-l:fileList
```

Arguments

TERM	DEFINITION
`fileList`	Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

Remarks

The `-link` option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see Walkthrough: Embedding Types from Managed Assemblies.

Using the `-link` option is especially useful when you are working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The `-link` option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types.

The `-link` option embeds only interfaces, structures, and delegates. Embedding COM classes is not supported.

> **NOTE**
> When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the CoClass causes an error.

To set the `-link` option in Visual Studio, add an assembly reference and set the `Embed Interop Types` property to **true**. The default for the `Embed Interop Types` property is **false**.

If you link to a COM assembly (Assembly A) which itself references another COM assembly (Assembly B), you

also have to link to Assembly B if either of the following is true:

- A type from Assembly A inherits from a type or implements an interface from Assembly B.
- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Use -libpath to specify the directory in which one or more of your assembly references is located.

Like the /reference compiler option, the `-link` compiler option uses the Vbc.rsp response file, which references frequently used .NET Framework assemblies. Use the -noconfig compiler option if you do not want the compiler to use the Vbc.rsp file.

The short form of `-link` is `-l`.

Generics and Embedded Types

The following sections describe the limitations on using generic types in applications that embed interop types.

Generic Interfaces

Generic interfaces that are embedded from an interop assembly cannot be used. This is shown in the following example.

```
' The following code causes an error if ISampleInterface is an embedded interop type.
Dim sample As ISampleInterface(Of SampleType)
```

Types That Have Generic Parameters

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction does not apply to interfaces. For example, consider the Range interface that is defined in the Microsoft.Office.Interop.Excel assembly. If a library embeds interop types from the Microsoft.Office.Interop.Excel assembly and exposes a method that returns a generic type that has a parameter whose type is the Range interface, that method must return a generic interface, as shown in the following code example.

```
Imports System.Collections.Generic
Imports Microsoft.Office.Interop.Excel

Class Utility
    ' The following code causes an error when called by a client assembly.
    Public Function GetRange1() As List(Of Range)
```

```
End Function

' The following code is valid for calls from a client assembly.
Public Function GetRange2() As IList(Of Range)
```

```
    End Function
End Class
```

In the following example, client code can call the method that returns the IList generic interface without error.

```
Module Client
    Public Sub Main()
        Dim util As New Utility()

        ' The following code causes an error.
        Dim rangeList1 As List(Of Range) = util.GetRange1()

        ' The following code is valid.
        Dim rangeList2 As List(Of Range) = CType(util.GetRange2(), List(Of Range))
    End Sub
End Module
```

Example

The following command line compiles source file `OfficeApp.vb` and reference assemblies from `COMData1.dll` and `COMData2.dll` to produce `OfficeApp.exe`.

```
vbc -link:COMData1.dll,COMData2.dll /out:OfficeApp.exe OfficeApp.vb
```

See also

- Visual Basic Command-Line Compiler
- Walkthrough: Embedding Types from Managed Assemblies
- -reference (Visual Basic)
- -noconfig
- -libpath
- Sample Compilation Command Lines
- Introduction to COM Interop

-linkresource (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Creates a link to a managed resource.

Syntax

```
-linkresource:filename[,identifier[,public|private]]
' -or-
-linkres:filename[,identifier[,public|private]]
```

Arguments

`filename`
Required. The resource file to link to the assembly. If the file name contains a space, enclose the name in quotation marks (" ").

`identifier`
Optional. The logical name for the resource. The name that is used to load the resource. The default is the name of the file. Optionally, you can specify whether the file is public or private in the assembly manifest, for example: `-linkres:filename.res,myname.res,public`. By default, `filename` is public in the assembly.

Remarks

The `-linkresource` option does not embed the resource file in the output file; use the `-resource` option to do this.

The `-linkresource` option requires one of the `-target` options other than `-target:module`.

If `filename` is a .NET Framework resource file created, for example, by the Resgen.exe (Resource File Generator) or in the development environment, it can be accessed with members in the System.Resources namespace. (For more information, see ResourceManager.) To access all other resources at run time, use the methods that begin with `GetManifestResource` in the Assembly class.

The file name can be any file format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly.

The short form of `-linkresource` is `-linkres`.

> **NOTE**
> The `-linkresource` option is not available from the Visual Studio development environment; it is available only when you compile from the command line.

Example

The following code compiles `in.vb` and links to resource file `rf.resource`.

```
vbc -linkresource:rf.resource in.vb
```

See also

- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- -resource (Visual Basic)
- Sample Compilation Command Lines

-main

4/9/2019 • 2 minutes to read • Edit Online

Specifies the class or module that contains the `Sub Main` procedure.

Syntax

```
-main:location
```

Arguments

`location`
Required. The name of the class or module that contains the `Sub Main` procedure to be called when the program starts. This may be in the form **-main:module** or **-main:namespace.module**.

Remarks

Use this option when you create an executable file or Windows executable program. If the **-main** option is omitted, the compiler searches for a valid shared `Sub Main` in all public classes and modules.

See Main Procedure in Visual Basic for a discussion of the various forms of the `Main` procedure.

When `location` is a class that inherits from Form, the compiler provides a default `Main` procedure that starts the application if the class has no `Main` procedure. This lets you compile code at the command line that was created in the development environment.

```
' Compile with /r:System.dll,SYSTEM.WINDOWS.FORMS.DLL /main:MyC
Public Class MyC
    Inherits System.Windows.Forms.Form
End Class
```

To set -main in the Visual Studio integrated development environment

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Make sure the **Enable application framework** check box is not checked.
4. Modify the value in the **Startup object** box.

Example

The following code compiles `T2.vb` and `T3.vb`, specifying that the `Sub Main` procedure will be found in the `Test2` class.

```
vbc t2.vb t3.vb -main:Test2
```

See also

- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- Sample Compilation Command Lines
- Main Procedure in Visual Basic

-moduleassemblyname

8/22/2019 • 2 minutes to read • Edit Online

Specifies the name of the assembly that this module will be a part of.

Syntax

```
-moduleassemblyname:assembly_name
```

Arguments

TERM	DEFINITION
`assembly_name`	The name of the assembly that this module will be a part of.

Remarks

The compiler processes the `-moduleassemblyname` option only if the `-target:module` option has been specified. This causes the compiler to create a module. The module created by the compiler is valid only for the assembly specified with the `-moduleassemblyname` option. If you place the module in a different assembly, run-time errors will occur.

The `-moduleassemblyname` option is needed only when the following are true:

- A data type in the module needs access to a `Friend` type in a referenced assembly.
- The referenced assembly has granted friend assembly access to the assembly into which the module will be built.

For more information about creating a module, see /target (Visual Basic). For more information about friend assemblies, see Friend Assemblies.

> **NOTE**
> The `-moduleassemblyname` option is not available from within the Visual Studio development environment; it is available only when you compile from a command prompt.

See also

- How to: Build a Multifile Assembly
- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- -main
- -reference (Visual Basic)
- -addmodule
- Assemblies in .NET
- Sample Compilation Command Lines

- Friend Assemblies

-netcf

6/20/2019 • 2 minutes to read • Edit Online

Sets the compiler to target the .NET Compact Framework.

Syntax

```
-netcf
```

Remarks

The `-netcf` option causes the Visual Basic compiler to target the .NET Compact Framework rather than the full .NET Framework. Language functionality that is present only in the full .NET Framework is disabled.

The `-netcf` option is designed to be used with -sdkpath. The language features disabled by `-netcf` are the same language features not present in the files targeted with `-sdkpath`.

> **NOTE**
> The `-netcf` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. The `-netcf` option is set when a Visual Basic device project is loaded.

The `-netcf` option changes the following language features:

- The End <keyword> Statement keyword, which terminates execution of a program, is disabled. The following program compiles and runs without `-netcf` but fails at compile time with `-netcf`.

  ```
  Module Module1
      Sub Main()
          End   ' not valid to terminate execution with /netcf
      End Sub
  End Module
  ```

- Late binding, in all forms, is disabled. Compile-time errors are generated when recognized late-binding scenarios are encountered. The following program compiles and runs without `-netcf` but fails at compile time with `-netcf`.

```
Class LateBoundClass
    Sub S1()
    End Sub

    Default Property P1(ByVal s As String) As Integer
        Get
        End Get
        Set(ByVal Value As Integer)
        End Set
    End Property
End Class

Module Module1
    Sub Main()
        Dim o1 As Object
        Dim o2 As Object
        Dim o3 As Object
        Dim IntArr(3) As Integer

        o1 = New LateBoundClass
        o2 = 1
        o3 = IntArr

        ' Late-bound calls
        o1.S1()
        o1.P1("member") = 1

        ' Dictionary member access
        o1!member = 1

        ' Late-bound overload resolution
        LateBoundSub(o2)

        ' Late-bound array
        o3(1) = 1
    End Sub

    Sub LateBoundSub(ByVal n As Integer)
    End Sub

    Sub LateBoundSub(ByVal s As String)
    End Sub
End Module
```

- The Auto, Ansi, and Unicode modifiers are disabled. The syntax of the Declare Statement statement is also modified to `Declare Sub|Function name Lib "library" [Alias "alias"] [([arglist])]`. The following code shows the effect of `-netcf` on a compilation.

```
' compile with: /target:library
Module Module1
    ' valid with or without /netcf
    Declare Sub DllSub Lib "SomeLib.dll" ()

    ' not valid with /netcf
    Declare Auto Sub DllSub1 Lib "SomeLib.dll" ()
    Declare Ansi Sub DllSub2 Lib "SomeLib.dll" ()
    Declare Unicode Sub DllSub3 Lib "SomeLib.dll" ()
End Module
```

- Using Visual Basic 6.0 keywords that were removed from Visual Basic generates a different error when `-netcf` is used. This affects the error messages for the following keywords:
  - `Open`

- `Close`
- `Put`
- `Print`
- `Write`
- `Input`
- `Lock`
- `Unlock`
- `Seek`
- `Width`
- `Name`
- `FreeFile`
- `EOF`
- `Loc`
- `LOF`
- `Line`

Example

The following code compiles `Myfile.vb` with the .NET Compact Framework, using the versions of mscorlib.dll and Microsoft.VisualBasic.dll found in the default installation directory of the .NET Compact Framework on the C drive. Typically, you would use the most recent version of the .NET Compact Framework.

```
vbc -netcf -sdkpath:"c:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\v1.0.5000\Windows CE " myfile.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- -sdkpath

-noconfig

8/22/2019 • 2 minutes to read • Edit Online

Specifies that the compiler should not automatically reference the commonly used .NET Framework assemblies or import the `System` and `Microsoft.VisualBasic` namespaces.

Syntax

```
-noconfig
```

Remarks

The `-noconfig` option tells the compiler not to compile with the Vbc.rsp file, which is located in the same directory as the Vbc.exe file. The Vbc.rsp file references the commonly used .NET Framework assemblies and imports the `System` and `Microsoft.VisualBasic` namespaces. The compiler implicitly references the System.dll assembly unless the `-nostdlib` option is specified. The `-nostdlib` option tells the compiler not to compile with Vbc.rsp or automatically reference the System.dll assembly.

> **NOTE**
> The Mscorlib.dll and Microsoft.VisualBasic.dll assemblies are always referenced.

You can modify the Vbc.rsp file to specify additional compiler options that should be included in every Vbc.exe compilation (except when specifying the `-noconfig` option). For more information, see @ (Specify Response File).

The compiler processes the options passed to the `vbc` command last. Therefore, any option on the command line overrides the setting of the same option in the Vbc.rsp file.

> **NOTE**
> The `-noconfig` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

See also

- -nostdlib (Visual Basic)
- Visual Basic Command-Line Compiler
- @ (Specify Response File)
- -reference (Visual Basic)

-nologo (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Suppresses display of the copyright banner and informational messages during compilation.

Syntax

```
-nologo
```

Remarks

If you specify `-nologo`, the compiler does not display a copyright banner. By default, `-nologo` is not in effect.

> **NOTE**
> The `-nologo` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `T2.vb` and does not display a copyright banner.

```
vbc -nologo t2.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-nostdlib (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Causes the compiler not to automatically reference the standard libraries.

Syntax

```
-nostdlib
```

Remarks

The `-nostdlib` option removes the automatic reference to the System.dll assembly and prevents the compiler from reading the Vbc.rsp file. The Vbc.rsp file, which is located in the same directory as the Vbc.exe file, references the commonly used .NET Framework assemblies and imports the `System` and `Microsoft.VisualBasic` namespaces.

> **NOTE**
> The Mscorlib.dll and Microsoft.VisualBasic.dll assemblies are always referenced.

> **NOTE**
> The `-nostdlib` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `T2.vb` without referencing the standard libraries. You must set the `_MYTYPE` conditional-compilation constant to the string "Empty" to remove the `My` object.

```
vbc -nostdlib -define:_MYTYPE=\"Empty\" T2.vb
```

See also

- -noconfig
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- Customizing Which Objects are Available in My

-nowarn

4/2/2019 • 2 minutes to read • Edit Online

Suppresses the compiler's ability to generate warnings.

Syntax

```
-nowarn[:numberList]
```

Arguments

TERM	DEFINITION
`numberList`	Optional. Comma-delimited list of the warning ID numbers that the compiler should suppress. If the warning IDs are not specified, all warnings are suppressed.

Remarks

The `-nowarn` option causes the compiler to not generate warnings. To suppress an individual warning, supply the warning ID to the `-nowarn` option following the colon. Separate multiple warning numbers with commas.

You need to specify only the numeric part of the warning identifier. For example, if you want to suppress BC42024, the warning for unused local variables, specify `-nowarn:42024`.

For more information on the warning ID numbers, see Configuring Warnings in Visual Basic.

TO SET -NOWARN IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Compile** tab. 3. Select the **Disable all warnings** check box to disable all warnings. - or - To disable a particular warning, click **None** from the drop-down list adjacent to the warning.

Example

The following code compiles `T2.vb` and does not display any warnings.

```
vbc -nowarn t2.vb
```

Example

The following code compiles `T2.vb` and does not display the warnings for unused local variables (42024).

```
vbc -nowarn:42024 t2.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- Configuring Warnings in Visual Basic

-nowin32manifest (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Instructs the compiler not to embed any application manifest into the executable file.

Syntax

```
-nowin32manifest
```

Remarks

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step. For more information about virtualization, see ClickOnce Deployment on Windows Vista.

For more information about manifest creation, see -win32manifest (Visual Basic).

See also

- Visual Basic Command-Line Compiler
- Application Page, Project Designer (Visual Basic)

-optimize

4/2/2019 • 2 minutes to read • Edit Online

Enables or disables compiler optimizations.

Syntax

```
-optimize[ + | - ]
```

Arguments

TERM	DEFINITION
`+` | `-`	Optional. The `-optimize-` option disables compiler optimizations. The `-optimize+` option enables optimizations. By default, optimizations are disabled.

Remarks

Compiler optimizations make your output file smaller, faster, and more efficient. However, because optimizations result in code rearrangement in the output file, `-optimize+` can make debugging difficult.

All modules generated with `-target:module` for an assembly must use the same `-optimize` settings as the assembly. For more information, see -target (Visual Basic).

You can combine the `-optimize` and `-debug` options.

TO SET -OPTIMIZE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Compile** tab. 3. Click the **Advanced** button. 4. Modify the **Enable optimizations** check box.

Example

The following code compiles `T2.vb` and enables compiler optimizations.

```
vbc t2.vb -optimize
```

See also

- Visual Basic Command-Line Compiler
- -debug (Visual Basic)
- Sample Compilation Command Lines
- -target (Visual Basic)

-optioncompare

4/28/2019 • 2 minutes to read • Edit Online

Specifies how string comparisons are made.

Syntax

```
-optioncompare:{binary | text}
```

Remarks

You can specify `-optioncompare` in one of two forms: `-optioncompare:binary` to use binary string comparisons, and `-optioncompare:text` to use text string comparisons. By default, the compiler uses `-optioncompare:binary`.

In Microsoft Windows, the current code page determines the binary sort order. A typical binary sort order is as follows:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

Text-based string comparisons are based on a case-insensitive text sort order determined by your system's locale. A typical text sort order is as follows:

```
(A = a) < (À = à) < (B=b) < (E=e) < (Ê = ê) < (Z=z) < (Ø = ø)
```

To set -optioncompare in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Modify the value in the **Option Compare** box.

To set -optioncompare programmatically

- See Option Compare Statement.

Example

The following code compiles `ProjFile.vb` and uses binary string comparisons.

```
vbc -optioncompare:binary projFile.vb
```

See also

- Visual Basic Command-Line Compiler
- -optionexplicit
- -optionstrict
- -optioninfer
- Sample Compilation Command Lines
- Option Compare Statement
- Visual Basic Defaults, Projects, Options Dialog Box

-optionexplicit

4/9/2019 • 2 minutes to read • Edit Online

Causes the compiler to report errors if variables are not declared before they are used.

Syntax

```
-optionexplicit[+ | -]
```

Arguments

`+` | `-`

Optional. Specify `-optionexplicit+` to require explicit declaration of variables. The `-optionexplicit+` option is the default and is the same as `-optionexplicit`. The `-optionexplicit-` option enables implicit declaration of variables.

Remarks

If the source code file contains an Option Explicit statement, the statement overrides the `-optionexplicit` command-line compiler setting.

To set -optionexplicit in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Modify the value in the **Option Explicit** box.

Example

The following code compiles when `-optionexplicit-` is used.

```
Module Module1
    Sub Main()
        i = 99
        System.Console.WriteLine(i)
    End Sub
End Module
```

See also

- Visual Basic Command-Line Compiler
- -optioncompare
- -optionstrict
- -optioninfer
- Sample Compilation Command Lines
- Option Explicit Statement
- Visual Basic Defaults, Projects, Options Dialog Box

-optioninfer

8/22/2019 • 2 minutes to read • Edit Online

Enables the use of local type inference in variable declarations.

Syntax

```
-optioninfer[+ | -]
```

Arguments

TERM	DEFINITION
`+` | `-`	Optional. Specify `-optioninfer+` to enable local type inference, or `-optioninfer-` to block it. The `-optioninfer` option, with no value specified, is the same as `-optioninfer+`. The default value when the `-optioninfer` switch is not present is also `-optioninfer+`. The default value is set in the Vbc.rsp response file.

NOTE

You can use the `-noconfig` option to retain the compiler's internal defaults instead of those specified in vbc.rsp. The compiler default for this option is `-optioninfer-`.

Remarks

If the source code file contains an Option Infer Statement, the statement overrides the `-optioninfer` command-line compiler setting.

To set -optioninfer in the Visual Studio IDE

1. Select a project in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. On the **Compile** tab, modify the value in the **Option infer** box.

Example

The following code compiles `test.vb` with local type inference enabled.

```
vbc -optioninfer+ test.vb
```

See also

- Visual Basic Command-Line Compiler
- -optioncompare
- -optionexplicit
- -optionstrict

- Sample Compilation Command Lines
- Option Infer Statement
- Local Type Inference
- Visual Basic Defaults, Projects, Options Dialog Box
- Compile Page, Project Designer (Visual Basic)
- /noconfig
- Building from the Command Line

-optionstrict

4/28/2019 • 2 minutes to read • Edit Online

Enforces strict type semantics to restrict implicit type conversions.

Syntax

```
-optionstrict[+ | -]
-optionstrict[:custom]
```

Arguments

`+` | `-`
Optional. The `-optionstrict+` option restricts implicit type conversion. The default for this option is `-optionstrict-`. The `-optionstrict+` option is the same as `-optionstrict`. You can use both for permissive type semantics.

`custom`
Required. Warn when strict language semantics are not respected.

Remarks

When `-optionstrict+` is in effect, only widening type conversions can be made implicitly. Implicit narrowing type conversions, such as assigning a `Decimal` type object to an integer type object, are reported as errors.

To generate warnings for implicit narrowing type conversions, use `-optionstrict:custom`. Use `-nowarn:numberlist` to ignore particular warnings and `-warnaserror:numberlist` to treat particular warnings as errors.

To set -optionstrict in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties.**
2. Click the **Compile** tab.
3. Modify the value in the **Option Strict** box.

To set -optionstrict programmatically

- See Option Strict Statement.

Example

The following code compiles `Test.vb` using strict type semantics.

```
vbc -optionstrict+ test.vb
```

See also

- Visual Basic Command-Line Compiler
- -optioncompare

- -optionexplicit
- -optioninfer
- -nowarn
- -warnaserror (Visual Basic)
- Sample Compilation Command Lines
- Option Strict Statement
- Visual Basic Defaults, Projects, Options Dialog Box

-out (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

Specifies the name of the output file.

Syntax

```
-out:filename
```

Arguments

TERM	DEFINITION
`filename`	Required. The name of the output file the compiler creates. If the file name contains a space, enclose the name in quotation marks (" ").

Remarks

Specify the full name and extension of the file to create. If you do not, the .exe file takes its name from the source-code file containing the `Sub Main` procedure, and the .dll file takes its name from the first source-code file.

If you specify a file name without an .exe or .dll extension, the compiler automatically adds the extension for you, depending on the value specified for the `-target` compiler option.

TO SET -OUT IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Application** tab. 3. Modify the value in the **Assembly Name** box.

Example

The following code compiles `T2.vb` and creates output file `T2.exe`.

```
vbc t2.vb -out:t3.exe
```

See also

- Visual Basic Command-Line Compiler
- -target (Visual Basic)
- Sample Compilation Command Lines

-platform (Visual Basic)

5/30/2019 • 2 minutes to read • Edit Online

Specifies which platform version of common language runtime (CLR) can run the output file.

Syntax

```
-platform:{ x86 | x64 | Itanium | arm | anycpu | anycpu32bitpreferred }
```

Arguments

TERM	DEFINITION
`x86`	Compiles your assembly to be run by the 32-bit, x86-compatible CLR.
`x64`	Compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
`Itanium`	Compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.
`arm`	Compiles your assembly to be run on a computer with an ARM (Advanced RISC Machine) processor.
`anycpu`	Compiles your assembly to run on any platform. The application will run as a 32-bit application on 32-bit versions of Windows and as a 64-bit application on 64-bit versions of Windows. This flag is the default value.
`anycpu32bitpreferred`	Compiles your assembly to run on any platform. The application will run as a 32-bit application on both 32-bit and 64-bit versions of Windows. This flag is valid only for executables (.EXE) and requires .NET Framework 4.5.

Remarks

Use the `-platform` option to specify the type of processor targeted by the output file.

In general, .NET Framework assemblies written in Visual Basic will run the same regardless of the platform. However, there are some cases that behave differently on different platforms. These common cases are:

- Structures that contain members that change size depending on the platform, such as any pointer type.
- Pointer arithmetic that includes constant sizes.
- Incorrect platform invoke or COM declarations that use `Integer` for handles instead of IntPtr.
- Casting IntPtr to `Integer`.
- Using platform invoke or COM interop with components that do not exist on all platforms.

The **-platform** option will mitigate some issues if you know you have made assumptions about the architecture your code will run on. Specifically:

- If you decide to target a 64-bit platform, and the application is run on a 32-bit machine, the error message comes much earlier and is more targeted at the problem than the error that occurs without using this switch.
- If you set the `x86` flag on the option and the application is subsequently run on a 64-bit machine, the application will run in the WOW subsystem instead of running natively.

On a 64-bit Windows operating system:

- Assemblies compiled with `-platform:x86` will execute on the 32-bit CLR running under WOW64.
- Executables compiled with the `-platform:anycpu` will execute on the 64-bit CLR.
- A DLL compiled with the `-platform:anycpu` will execute on the same CLR as the process into which it loaded.
- Executables that are compiled with `-platform:anycpu32bitpreferred` will execute on the 32-bit CLR.

For more information about how to develop an application to run on a 64-bit version of Windows, see 64-bit Applications.

To set -platform in the Visual Studio IDE

1. In **Solution Explorer**, choose the project, open the **Project** menu, and then click **Properties**.
2. On the **Compile** tab, select or clear the **Prefer 32-bit** check box, or, in the **Target CPU** list, choose a value.

   For more information, see Compile Page, Project Designer (Visual Basic).

Example

The following example illustrates how to use the `-platform` compiler option.

```
vbc -platform:x86 myFile.vb
```

See also

- /target (Visual Basic)
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-quiet

4/16/2019 • 2 minutes to read • Edit Online

Prevents the compiler from displaying code for syntax-related errors and warnings.

Syntax

```
-quiet
```

Remarks

By default, `-quiet` is not in effect. When the compiler reports a syntax-related error or warning, it also outputs the line from source code. For applications that parse compiler output, it may be more convenient for the compiler to output only the text of the diagnostic.

In the following example, `Module1` outputs an error that includes source code when compiled without `-quiet`.

```
Module Module1
    Sub Main()
        x()
    End Sub
End Module
```

Output:

```
C:\projects\vb2.vb(3) : error BC30451: 'x' is not declared. It may be inaccessible due to its protection
level.

        x()
        ~
```

Compiled with `-quiet`, the compiler outputs only the following:

```
E:\test\t2.vb(3) : error BC30451: Name 'x' is not declared.
```

> **NOTE**
>
> The `-quiet` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `T2.vb` and does not display code for syntax-related compiler diagnostics:

```
vbc -quiet t2.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-recurse

8/22/2019 • 2 minutes to read • Edit Online

Compiles source-code files in all child directories of either the specified directory or the project directory.

Syntax

```
-recurse:[dir\]file
```

Arguments

`dir`
Optional. The directory in which you want the search to begin. If not specified, the search begins in the project directory.

`file`
Required. The file(s) to search for. Wildcard characters are allowed.

Remarks

You can use wildcards in a file name to compile all matching files in the project directory without using `-recurse`. If no output file name is specified, the compiler bases the output file name on the first input file processed. This is generally the first file in the list of files compiled when viewed alphabetically. For this reason, it is best to specify an output file using the `-out` option.

> **NOTE**
> The `-recurse` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following command compiles all Visual Basic files in the current directory.

```
vbc *.vb
```

The following command compiles all Visual Basic files in the `Test\ABC` directory and any directories below it, and then generates `Test.ABC.dll`.

```
vbc -target:library -out:Test.ABC.dll -recurse:Test\ABC\*.vb
```

See also

- Visual Basic Command-Line Compiler
- -out (Visual Basic)
- Sample Compilation Command Lines

-reference (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

Causes the compiler to make type information in the specified assemblies available to the project you are currently compiling.

Syntax

```
-reference:fileList
' -or-
-r:fileList
```

Arguments

TERM	DEFINITION
`fileList`	Required. Comma-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks.

Remarks

The file(s) you import must contain assembly metadata. Only public types are visible outside the assembly. The /addmodule option imports metadata from a module.

If you reference an assembly (Assembly A) which itself references another assembly (Assembly B), you need to reference Assembly B if:

- A type from Assembly A inherits from a type or implements an interface from Assembly B.
- A field, property, event, or method that has a return type or parameter type from Assembly B is invoked.

Use -libpath to specify the directory in which one or more of your assembly references is located.

For the compiler to recognize a type in an assembly (not a module), it must be forced to resolve the type. One example of how you can do this is to define an instance of the type. Other ways are available to resolve type names in an assembly for the compiler. For example, if you inherit from a type in an assembly, the type name then becomes known to the compiler.

The Vbc.rsp response file, which references commonly used .NET Framework assemblies, is used by default. Use `-noconfig` if you do not want the compiler to use Vbc.rsp.

The short form of `-reference` is `/r`.

Example

The following command compiles source file `Input.vb` and reference assemblies from `Metad1.dll` and `Metad2.dll` to produce `Out.exe`.

```
vbc -reference:metad1.dll,metad2.dll -out:out.exe input.vb
```

See also

- Visual Basic Command-Line Compiler
- -noconfig
- -target (Visual Basic)
- Public
- Sample Compilation Command Lines

-refonly (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The **-refonly** option indicates that the primary output of the compilation should be a reference assembly instead of an implementation assembly. The `-refonly` parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Syntax

```
-refonly
```

Remarks

Visual Basic supports the `-refout` switch starting with version 15.3.

Reference assemblies are metadata-only assemblies that contain metadata but no implementation code. They include type and member information for everything except anonymous types. The reason for using `throw null` bodies (as opposed to no bodies) is so that PEVerify could run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level ReferenceAssembly attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes will refuse to load reference assemblies for execution (but they can still be loaded in a reflection-only context). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only; otherwise, the runtime throws a BadImageFormatException.

The `-refonly` and `-refout` options are mutually exclusive.

See also

- -refout
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-refout (Visual Basic)

4/2/2019 • 2 minutes to read • Edit Online

The **-refout** option specifies a file path where the reference assembly should be output.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Syntax

```
-refout:filepath
```

Arguments

`filepath` The path and filename of the reference assembly. It should generally be in a sub-folder of the primary assembly. The recommended convention (used by MSBuild) is to place the reference assembly in a "ref/" sub-folder relative to the primary assembly. All folders in `filepath` must exist; the compiler does not create them.

Remarks

Visual Basic supports the `-refout` switch starting with version 15.3.

Reference assemblies are metadata-only assemblies that contain metadata but no implementation code. They include type and member information for everything except anonymous types. Their method bodies are replaced with a single `throw null` statement. The reason for using `throw null` method bodies (as opposed to no bodies) is so that PEVerify can run and pass (thus validating the completeness of the metadata).

Reference assemblies include an assembly-level ReferenceAssembly attribute. This attribute may be specified in source (then the compiler won't need to synthesize it). Because of this attribute, runtimes refuse to load reference assemblies for execution (but they can still be loaded in a reflection-only context). Tools that reflect on assemblies need to ensure they load reference assemblies as reflection-only; otherwise, the runtime throws a BadImageFormatException.

The `-refout` and `-refonly` options are mutually exclusive.

See also

- -refonly
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-removeintchecks

4/2/2019 • 2 minutes to read • Edit Online

Turns overflow-error checking for integer operations on or off.

Syntax

```
-removeintchecks[+ | -]
```

Arguments

TERM	DEFINITION
`+` | `-`	Optional. The `-removeintchecks-` option causes the compiler to check all integer calculations for overflow errors. The default is `-removeintchecks-`. Specifying `-removeintchecks` or `-removeintchecks+` prevents error checking and can make integer calculations faster. However, without error checking, and if data type capacities are overflowed, incorrect results may be stored without raising an error.

TO SET -REMOVEINTCHECKS IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Click the **Advanced** button.
4. Modify the value of the **Remove integer overflow checks** box.

Example

The following code compiles `Test.vb` and turns off integer overflow-error checking.

```
vbc -removeintchecks+ test.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-resource (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

Embeds a managed resource in an assembly.

Syntax

```
-resource:filename[,identifier[,public|private]]
' -or-
-res:filename[,identifier[,public|private]]
```

Arguments

TERM	DEFINITION
`filename`	Required. The name of the resource file to embed in the output file. By default, `filename` is public in the assembly. Enclose the file name in quotation marks (" ") if it contains a space.
`identifier`	Optional. The logical name for the resource; the name used to load it. The default is the name of the file. Optionally, you can specify whether the resource is public or private in the assembly manifest, as with the following: `-res:filename.res, myname.res, public`

Remarks

Use `-linkresource` to link a resource to an assembly without placing the resource file in the output file.

If `filename` is a .NET Framework resource file created, for example, by the Resgen.exe (Resource File Generator) or in the development environment, it can be accessed with members in the System.Resources namespace (see ResourceManager for more information). To access all other resources at run time, use one of the following methods: GetManifestResourceInfo, GetManifestResourceNames, or GetManifestResourceStream.

The short form of `-resource` is `-res`.

For information about how to set `-resource` in the Visual Studio IDE, see Managing Application Resources (.NET).

Example

The following code compiles `In.vb` and attaches resource file `Rf.resource`.

```
vbc -res:rf.resource in.vb
```

See also

- Visual Basic Command-Line Compiler

- -win32resource
- -linkresource (Visual Basic)
- -target (Visual Basic)
- Sample Compilation Command Lines

-rootnamespace

1/23/2019 • 2 minutes to read • Edit Online

Specifies a namespace for all type declarations.

Syntax

```
-rootnamespace:namespace
```

Arguments

TERM	DEFINITION
`namespace`	The name of the namespace in which to enclose all type declarations for the current project.

Remarks

If you use the Visual Studio executable file (Devenv.exe) to compile a project created in the Visual Studio integrated development environment, use `-rootnamespace` to specify the value of the RootNamespace property. See Devenv Command Line Switches for more information.

Use the common language runtime MSIL Disassembler (`Ildasm.exe`) to view the namespace names in your output file.

TO SET -ROOTNAMESPACE IN THE VISUAL STUDIO INTEGRATED DEVELOPMENT ENVIRONMENT
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Application** tab. 3. Modify the value in the **Root Namespace** box.

Example

The following code compiles `In.vb` and encloses all type declarations in the namespace `mynamespace`.

```
vbc -rootnamespace:mynamespace in.vb
```

See also

- Visual Basic Command-Line Compiler
- Ildasm.exe (IL Disassembler)
- Sample Compilation Command Lines

-sdkpath

8/22/2019 • 2 minutes to read • Edit Online

Specifies the location of mscorlib.dll and Microsoft.VisualBasic.dll.

Syntax

```
-sdkpath:path
```

Arguments

`path`
The directory containing the versions of mscorlib.dll and Microsoft.VisualBasic.dll to use for compilation. This path is not verified until it is loaded. Enclose the directory name in quotation marks (" ") if it contains a space.

Remarks

This option tells the Visual Basic compiler to load the mscorlib.dll and Microsoft.VisualBasic.dll files from a non-default location. The `-sdkpath` option was designed to be used with -netcf. The .NET Compact Framework uses different versions of these support libraries to avoid the use of types and language features not found on the devices.

> **NOTE**
> The `-sdkpath` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. The `-sdkpath` option is set when a Visual Basic device project is loaded.

You can specify that the compiler should compile without a reference to the Visual Basic Runtime Library by using the `-vbruntime` compiler option. For more information, see -vbruntime.

Example

The following code compiles `Myfile.vb` with the .NET Compact Framework, using the versions of Mscorlib.dll and Microsoft.VisualBasic.dll found in the default installation directory of the .NET Compact Framework on the C drive. Typically, you would use the most recent version of the .NET Compact Framework.

```
vbc -netcf -sdkpath:"c:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\v1.0.5000\Windows
CE " myfile.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- -netcf
- -vbruntime

-target (Visual Basic)

5/14/2019 • 4 minutes to read • Edit Online

Specifies the format of compiler output.

Syntax

```
-target:{exe | library | module | winexe | appcontainerexe | winmdobj}
```

Remarks

The following table summarizes the effect of the `-target` option.

OPTION	BEHAVIOR
`-target:exe`	Causes the compiler to create an executable console application. This is the default option when no `-target` option is specified. The executable file is created with an .exe extension. Unless otherwise specified with the `/out` option, the output file name takes the name of the input file that contains the `Sub Main` procedure. Only one `Sub Main` procedure is required in the source-code files that are compiled into an .exe file. Use the `-main` compiler option to specify which class contains the `Sub Main` procedure.
`-target:library`	Causes the compiler to create a dynamic-link library (DLL). The dynamic-link library file is created with a .dll extension. Unless otherwise specified with the `-out` option, the output file name takes the name of the first input file. When building a DLL, a `Sub Main` procedure is not required.

OPTION	BEHAVIOR
`-target:module`	Causes the compiler to generate a module that can be added to an assembly. The output file is created with an extension of .netmodule. The .NET common language runtime cannot load a file that does not have an assembly. However, you can incorporate such a file into the assembly manifest of an assembly by using `-reference`. When code in one module references internal types in another module, both modules must be incorporated into an assembly manifest by using `-reference`. The -addmodule option imports metadata from a module.
`-target:winexe`	Causes the compiler to create an executable Windows-based application. The executable file is created with an .exe extension. A Windows-based application is one that provides a user interface from either the .NET Framework class library or with the Windows APIs. Unless otherwise specified with the `-out` option, the output file name takes the name of the input file that contains the `Sub Main` procedure. Only one `Sub Main` procedure is required in the source-code files that are compiled into an .exe file. In cases where your code has more than one class that has a `Sub Main` procedure, use the `-main` compiler option to specify which class contains the `Sub Main` procedure
`-target:appcontainerexe`	Causes the compiler to create an executable Windows-based application that must be run in an app container. This setting is designed to be used for Windows 8.x Store applications. The **appcontainerexe** setting sets a bit in the Characteristics field of the Portable Executable file. This bit indicates that the app must be run in an app container. When this bit is set, an error occurs if the `CreateProcess` method tries to launch the application outside of an app container. Aside from this bit setting, **-target:appcontainerexe** is equivalent to **-target:winexe**. The executable file is created with an .exe extension. Unless you specify otherwise by using the `-out` option, the output file name takes the name of the input file that contains the `Sub Main` procedure. Only one `Sub Main` procedure is required in the source-code files that are compiled into an .exe file. If your code contains more than one class that has a `Sub Main` procedure, use the `-main` compiler option to specify which class contains the `Sub Main` procedure

OPTION	BEHAVIOR
`-target:winmdobj`	Causes the compiler to create an intermediate file that you can convert to a Windows Runtime binary (.winmd) file. The .winmd file can be consumed by JavaScript and C++ programs, in addition to managed language programs. The intermediate file is created with a .winmdobj extension. Unless you specify otherwise by using the `-out` option, the output file name takes the name of the first input file. A `Sub Main` procedure isn't required. The .winmdobj file is designed to be used as input for the WinMDExp export tool to produce a Windows metadata (WinMD) file. The WinMD file has a .winmd extension and contains both the code from the original library and the WinMD definitions that JavaScript, C++, and the Windows Runtime use.

Unless you specify `-target:module`, `-target` causes a .NET Framework assembly manifest to be added to an output file.

Each instance of Vbc.exe produces, at most, one output file. If you specify a compiler option such as `-out` or `-target` more than one time, the last one the compiler processes is put into effect. Information about all files in a compilation is added to the manifest. All output files except those created with `-target:module` contain assembly metadata in the manifest. Use Ildasm.exe (IL Disassembler) to view the metadata in an output file.

The short form of `-target` is `-t`.

To set -target in the Visual Studio IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Application** tab.
3. Modify the value in the **Application Type** box.

Example

The following code compiles `in.vb`, creating `in.dll`:

```
vbc -target:library in.vb
```

See also

- Visual Basic Command-Line Compiler
- -main
- -out (Visual Basic)
- -reference (Visual Basic)
- -addmodule
- -moduleassemblyname
- Assemblies in .NET
- Sample Compilation Command Lines

-subsystemversion (Visual Basic)

5/30/2019 • 2 minutes to read • Edit Online

Specifies the minimum version of the subsystem on which the generated executable file can run, thereby determining the versions of Windows on which the executable file can run. Most commonly, this option ensures that the executable file can leverage particular security features that aren't available with older versions of Windows.

> **NOTE**
>
> To specify the subsystem itself, use the -target compiler option.

Syntax

```
-subsystemversion:major.minor
```

Parameters

`major.minor`

The minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7 if you set the value of this option to 6.01, as the table later in this topic describes. You must specify the values for `major` and `minor` as integers.

Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

Remarks

The following table lists common subsystem versions of Windows.

WINDOWS VERSION	SUBSYSTEM VERSION
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Default values

The default value of the **-subsystemversion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
  - -target:appcontainerexe
  - -target:winmdobj
  - -platform:arm
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5, and you haven't set any of the compiler options that were specified earlier in this list.
- The default value is 4.00 if none of the previous conditions is true.

Setting this option

To set the **-subsystemversion** compiler option in Visual Studio, you must open the .vbproj file and specify a value for the `SubsystemVersion` property in the MSBuild XML. You can't set this option in the Visual Studio IDE. For more information, see "Default values" earlier in this topic or Common MSBuild Project Properties.

See also

- Visual Basic Command-Line Compiler
- MSBuild Properties

-utf8output (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Displays compiler output using UTF-8 encoding.

Syntax

```
-utf8output[+ | -]
```

Arguments

`+` | `-`

Optional. The default for this option is `-utf8output-`, which means compiler output does not use UTF-8 encoding. Specifying `-utf8output` is the same as specifying `-utf8output+`.

Remarks

In some international configurations, compiler output cannot be displayed correctly in the console. In such situations, use `-utf8output` and redirect compiler output to a file.

> **NOTE**
> The `-utf8output` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and directs the compiler to display output using UTF-8 encoding.

```
vbc -utf8output in.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-vbruntime

4/28/2019 • 2 minutes to read • Edit Online

Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.

Syntax

```
-vbruntime:{ - | + | * | path }
```

Arguments

-
Compile without a reference to the Visual Basic Runtime Library.

+
Compile with a reference to the default Visual Basic Runtime Library.

*
Compile without a reference to the Visual Basic Runtime Library, and embed core functionality from the Visual Basic Runtime Library into the assembly.

`path`
Compile with a reference to the specified library (DLL).

Remarks

The `-vbruntime` compiler option enables you to specify that the compiler should compile without a reference to the Visual Basic Runtime Library. If you compile without a reference to the Visual Basic Runtime Library, errors or warnings are logged on code or language constructs that generate a call to a Visual Basic runtime helper. (A *Visual Basic runtime helper* is a function defined in Microsoft.VisualBasic.dll that is called at runtime to execute a specific language semantic.)

The `-vbruntime+` option produces the same behavior that occurs if no `-vbruntime` switch is specified. You can use the `-vbruntime+` option to override previous `-vbruntime` switches.

Most objects of the `My` type are unavailable when you use the `-vbruntime-` or `-vbruntime:path` options.

Embedding Visual Basic Runtime core functionality

The `-vbruntime*` option enables you to compile without a reference to a runtime library. Instead, core functionality from the Visual Basic Runtime Library is embedded in the user assembly. You can use this option if your application runs on platforms that do not contain the Visual Basic runtime.

The following runtime members are embedded:

- Conversions class
- AscW(Char) method
- AscW(String) method

- ChrW(Int32) method
- vbBack constant
- vbCr constant
- vbCrLf constant
- vbFormFeed constant
- vbLf constant
- vbNewLine constant
- vbNullChar constant
- vbNullString constant
- vbTab constant
- vbVerticalTab constant
- Some objects of the `My` type

If you compile using the `-vbruntime*` option and your code references a member from the Visual Basic Runtime Library that is not embedded with the core functionality, the compiler returns an error that indicates that the member is not available.

Referencing a specified library

You can use the `path` argument to compile with a reference to a custom runtime library instead of the default Visual Basic Runtime Library.

If the value for the `path` argument is a fully qualified path to a DLL, the compiler will use that file as the runtime library. If the value for the `path` argument is not a fully qualified path to a DLL, the Visual Basic compiler will search for the identified DLL in the current folder first. It will then search in the path that you have specified by using the -sdkpath compiler option. If the `-sdkpath` compiler option is not used, the compiler will search for the identified DLL in the .NET Framework folder (`%systemroot%\Microsoft.NET\Framework\versionNumber`).

Example

The following example shows how to use the `-vbruntime` option to compile with a reference to a custom library.

```
vbc -vbruntime:C:\VBLibraries\CustomVBLibrary.dll
```

See also

- Visual Basic Core – New compilation mode in Visual Studio 2010 SP1
- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- -sdkpath

-verbose

8/22/2019 • 2 minutes to read • Edit Online

Causes the compiler to produce verbose status and error messages.

Syntax

```
-verbose[+ | -]
```

Arguments

`+` | `-`

Optional. Specifying `-verbose` is the same as specifying `-verbose+`, which causes the compiler to emit verbose messages. The default for this option is `-verbose-`.

Remarks

The `-verbose` option displays information about the total number of errors issued by the compiler, reports which assemblies are being loaded by a module, and displays which files are currently being compiled.

> **NOTE**
> The `-verbose` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and directs the compiler to display verbose status information.

```
vbc -verbose in.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-warnaserror (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Causes the compiler to treat the first occurrence of a warning as an error.

Syntax

```
-warnaserror[+ | -][:numberList]
```

Arguments

TERM	DEFINITION
+ | -	Optional. By default, `-warnaserror-` is in effect; warnings do not prevent the compiler from producing an output file. The `-warnaserror` option, which is the same as `-warnaserror+`, causes warnings to be treated as errors.
`numberList`	Optional. Comma-delimited list of the warning ID numbers to which the `-warnaserror` option applies. If no warning ID is specified, the `-warnaserror` option applies to all warnings.

Remarks

The `-warnaserror` option treats all warnings as errors. Any messages that would ordinarily be reported as warnings are instead reported as errors. The compiler reports subsequent occurrences of the same warning as warnings.

By default, `-warnaserror-` is in effect, which causes the warnings to be informational only. The `-warnaserror` option, which is the same as `-warnaserror+`, causes warnings to be treated as errors.

If you want only a few specific warnings to be treated as errors, you may specify a comma-separated list of warning numbers to treat as errors.

> **NOTE**
>
> The `-warnaserror` option does not control how warnings are displayed. Use the -nowarn option to disable warnings.

TO SET -WARNASERROR TO TREAT ALL WARNINGS AS ERRORS IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Make sure the **Disable all warnings** check box is unchecked.
4. Check the **Treat all warnings as errors** check box.

TO SET -WARNASERROR TO TREAT SPECIFIC WARNINGS AS ERRORS IN THE VISUAL STUDIO IDE

1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**.
2. Click the **Compile** tab.
3. Make sure the **Disable all warnings** check box is unchecked.
4. Make sure the **Treat all warnings as errors** check box is unchecked.
5. Select **Error** from the **Notification** column adjacent to the warning that should be treated as an error.

Example

The following code compiles `In.vb` and directs the compiler to display an error for the first occurrence of every warning it finds.

```
vbc -warnaserror in.vb
```

Example

The following code compiles `T2.vb` and treats only the warning for unused local variables (42024) as an error.

```
vbc -warnaserror:42024 t2.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines
- Configuring Warnings in Visual Basic

-win32icon

5/14/2019 • 2 minutes to read • Edit Online

Inserts an .ico file in the output file. This .ico file represents the output file in **File Explorer**.

Syntax

```
-win32icon:filename
```

Arguments

TERM	DEFINITION
`filename`	The .ico file to add to your output file. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

You can create an .ico file with the Microsoft Windows Resource Compiler (RC). The resource compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file. The `-win32icon` and `-win32resource` options are mutually exclusive.

See -linkresource (Visual Basic) to reference a .NET Framework resource file, or -resource (Visual Basic) to attach a .NET Framework resource file. See -win32resource to import a .res file.

TO SET -WIN32ICON IN THE VISUAL STUDIO IDE
1. Have a project selected in **Solution Explorer**. On the **Project** menu, click **Properties**. 2. Click the **Application** tab. 3. Modify the value in the **Icon** box.

Example

The following code compiles `In.vb` and attaches an .ico file, `Rf.ico`.

```
vbc -win32icon:rf.ico in.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

-win32manifest (Visual Basic)

8/22/2019 • 2 minutes to read • Edit Online

Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

Syntax

```
-win32manifest: fileName
```

Arguments

TERM	DEFINITION
`fileName`	The path of the custom manifest file.

Remarks

By default, the Visual Basic compiler embeds an application manifest that specifies a requested execution level of asInvoker. It creates the manifest in the same folder in which the executable file is built, typically the bin\Debug or bin\Release folder when you use Visual Studio. If you want to supply a custom manifest, for example to specify a requested execution level of highestAvailable or requireAdministrator, use this option to specify the name of the file.

> **NOTE**
> This option and the -win32resource option are mutually exclusive. If you try to use both options in the same command line, you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file/registry virtualization under the User Account Control feature in Windows Vista. For more information about virtualization, see ClickOnce Deployment on Windows Vista.

Your application will be subject to virtualization if either of the following conditions is true:

1. You use the `-nowin32manifest` option and you do not provide a manifest in a later build step or as part of a Windows Resource (.res) file by using the `-win32resource` option.
2. You provide a custom manifest that does not specify a requested execution level.

Visual Studio creates a default .manifest file and stores it in the debug and release directories alongside the executable file. You can view or edit the default app.manifest file by clicking **View UAC Settings** on the **Application** tab in the Project Designer. For more information, see Application Page, Project Designer (Visual Basic).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the `-nowin32manifest` option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista. This will prevent the compiler from creating and embedding a default manifest in the PE file.

Example

The following example shows the default manifest that the Visual Basic compiler inserts into a PE.

> **NOTE**
>
> The compiler inserts a standard application name MyApplication.app into the manifest XML. This is a workaround to enable applications to run on Windows Server 2003 Service Pack 3.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

See also

- Visual Basic Command-Line Compiler
- -nowin32manifest (Visual Basic)

-win32resource

8/22/2019 • 2 minutes to read • Edit Online

Inserts a Win32 resource file in the output file.

Syntax

```
-win32resource:filename
```

Arguments

`filename`
The name of the resource file to add to your output file. Enclose the file name in quotation marks (" ") if it contains a space.

Remarks

You can create a Win32 resource file with the Microsoft Windows Resource Compiler (RC).

A Win32 resource can contain version or bitmap (icon) information that helps identify your application in **File Explorer**. If you do not specify `-win32resource`, the compiler generates version information based on the assembly version. The `-win32resource` and `-win32icon` options are mutually exclusive.

See -linkresource (Visual Basic) to reference a .NET Framework resource file, or -resource (Visual Basic) to attach a .NET Framework resource file.

> **NOTE**
> The `-win32resource` option is not available from within the Visual Studio development environment; it is available only when compiling from the command line.

Example

The following code compiles `In.vb` and attaches a Win32 resource file, `Rf.res`:

```
vbc -win32resource:rf.res in.vb
```

See also

- Visual Basic Command-Line Compiler
- Sample Compilation Command Lines

Visual Basic compiler options listed by category

7/19/2019 • 3 minutes to read • Edit Online

The Visual Basic command-line compiler is provided as an alternative to compiling programs from within the Visual Studio integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted by functional category.

Every compiler option is available in two forms: **-option** and **/option**. The documentation only shows the -option form.

Compiler output

OPTION	PURPOSE
-nologo	Suppresses compiler banner information.
-utf8output	Displays compiler output using UTF-8 encoding.
-verbose	Outputs extra information during compilation.
`-modulename:<string>`	Specify the name of the source module
-preferreduilang	Specify a language for compiler output.

Optimization

OPTION	PURPOSE
-filealign	Specifies where to align the sections of the output file.
-optimize	Enables/disables optimizations.

Output files

OPTION	PURPOSE
-doc	Process documentation comments to an XML file.
-deterministic	Causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical.
-netcf	Sets the compiler to target the .NET Compact Framework.
-out	Specifies an output file.
-refonly	Outputs only a reference assembly.
-refout	Specifies the output path of a reference assembly.

OPTION	PURPOSE
-target	Specifies the format of the output.

.NET assemblies

OPTION	PURPOSE
-addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
-delaysign	Specifies whether the assembly will be fully or partially signed.
-imports	Imports a namespace from a specified assembly.
-keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
-keyfile	Specifies a file containing a key or key pair to give an assembly a strong name.
-libpath	Specifies the location of assemblies referenced by the -reference option.
-reference	Imports metadata from an assembly.
-moduleassemblyname	Specifies the name of the assembly that a module will be a part of.
`-analyzer`	Run the analyzers from this assembly (Short form: -a)
`-additionalfile`	Names additional files that don't directly affect code generation but may be used by analyzers for producing errors or warnings.

Debugging/error checking

OPTION	PURPOSE
-bugreport	Creates a file that contains information that makes it easy to report a bug.
-debug	Produces debugging information.
-nowarn	Suppresses the compiler's ability to generate warnings.
-quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
-removeintchecks	Disables integer overflow checking.
-warnaserror	Promotes warnings to errors.

OPTION	PURPOSE
`-ruleset:<file>`	Specify a ruleset file that disables specific diagnostics.

Help

OPTION	PURPOSE
-?	Displays the compiler options. This command is the same as specifying the `-help` option. No compilation occurs.
-help	Displays the compiler options. This command is the same as specifying the `-?` option. No compilation occurs.

Language

OPTION	PURPOSE
-langversion	Specify language version: 9|9.0|10|10.0|11|11.0.
-optionexplicit	Enforces explicit declaration of variables.
-optionstrict	Enforces strict type semantics.
-optioncompare	Specifies whether string comparisons should be binary or use locale-specific text semantics.
-optioninfer	Enables the use of local type inference in variable declarations.

Preprocessor

OPTION	PURPOSE
-define	Defines symbols for conditional compilation.

Resources

OPTION	PURPOSE
-linkresource	Creates a link to a managed resource.
-resource	Embeds a managed resource in an assembly.
-win32icon	Inserts an .ico file into the output file.
-win32resource	Inserts a Win32 resource into the output file.

Miscellaneous

OPTION	PURPOSE
@ (Specify Response File)	Specifies a response file.
-baseaddress	Specifies the base address of a DLL.
-codepage	Specifies the code page to use for all source code files in the compilation.
-errorreport	Specifies how the Visual Basic compiler should report internal compiler errors.
-highentropyva	Tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).
-main	Specifies the class that contains the `Sub Main` procedure to use at startup.
-noconfig	Do not compile with Vbc.rsp
-nostdlib	Causes the compiler not to reference the standard libraries.
-nowin32manifest	Instructs the compiler not to embed any application manifest into the executable file.
-platform	Specifies the processor platform the compiler targets for the output file.
-recurse	Searches subdirectories for source files to compile.
-rootnamespace	Specifies a namespace for all type declarations.
-sdkpath	Specifies the location of Mscorlib.dll and Microsoft.VisualBasic.dll.
-vbruntime	Specifies that the compiler should compile without a reference to the Visual Basic Runtime Library, or with a reference to a specific runtime library.
-win32manifest	Identifies a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.
`-parallel[+|-]`	Specifies whether to use concurrent build (+).
`-checksumalgorithm:<alg>`	Specify the algorithm for calculating the source file checksum stored in PDB. Supported values are: SHA1 (default) or SHA256. Due to collision problems with SHA1, Microsoft recommends SHA256 or better.

See also

- Visual Basic Compiler Options Listed Alphabetically
- Manage project and solution properties

.NET Framework Reference Information (Visual Basic)

5/14/2019 • 2 minutes to read • Edit Online

This topic provides links to information about how to work with the .NET Framework class library.

Related Sections

Getting Started

Provides a comprehensive overview of the .NET Framework and links to additional resources.

Class Library Overview

Introduces the classes, interfaces, and value types that help expedite and optimize the development process and provide access to system functionality.

Development Guide

Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application. This topic also provides information about dynamic programming, interoperability, extensibility, memory management, and threading.

Tools

Describes the tools that you can use to develop, configure, and deploy applications by using .NET Framework technologies.

.NET API Browser

Provides syntax, code examples, and related information for each class in the .NET Framework namespaces.

Visual Basic Language Specification

4/2/2019 • 2 minutes to read • Edit Online

The Visual Basic Language Specification is the authoritative source for answers to all questions about Visual Basic grammar and syntax. It contains detailed information about the language, including many points not covered in the Visual Basic reference documentation.

The specification is available on the Microsoft Download Center.

This site contains the VB 11 specification. It's built from the Markdown files contained in the dotnet/vblang GitHub repository.

Issues on the specification should be created in the dotnet/vblang repository. Or, if you're interested in fixing any errors you find, you may submit a Pull Request to the same repository.

See also

- Visual Basic Language Reference

NEXT

2 minutes to read

Visual Basic Language Walkthroughs

5/14/2019 • 2 minutes to read • Edit Online

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

Writing an Async Program

Shows how to create an asynchronous solution by using Async and Await.

Declaring and Raising Events

Illustrates how events are declared and raised in Visual Basic.

Handling Events

Shows how to handle events using either the standard `WithEvents` keyword or the new `AddHandler` / `RemoveHandler` keywords.

Creating and Implementing Interfaces

Shows how interfaces are declared and implemented in Visual Basic.

Defining Classes

Describes how to declare a class and its fields, properties, methods, and events.

Writing Queries in Visual Basic

Demonstrates how you can use Visual Basic language features to write Language-Integrated Query (LINQ) query expressions.

Implementing IEnumerable(Of T) in Visual Basic

Demonstrates how to create a class that implements the `IEnumerable(Of String)` interface and a class that implements the `IEnumerator(Of String)` interface to read a text file one line at a time.

Calling Windows APIs

Explains how to use `Declare` statements and call Windows APIs. Includes information about using attributes to control marshaling for the API call and how to expose an API call as a method of a class.

Creating COM Objects with Visual Basic

Demonstrates how to create COM objects in Visual Basic, both with and without the COM class template.

Implementing Inheritance with COM Objects

Demonstrates how to use Visual Basic 6.0 to create a COM object containing a class, and then use it as a base class in Visual Basic.

Determining Where My.Application.Log Writes Information

Describes the default `My.Application.Log` settings and how to determine the settings for your application.

Changing Where My.Application.Log Writes Information

Shows how to override the default `My.Application.Log` and `My.Log` settings for logging event information and cause the `Log` object to write to other log listeners.

Filtering My.Application.Log Output

Demonstrates how to change the default log filtering for the `My.Application.Log` object.

Creating Custom Log Listeners

Demonstrates how to create a custom log listener and configure it to listen to the output of the `My.Application.Log` object.

Embedding Types from Managed Assemblies

Describes how to create an assembly and a client program that embeds types from it.

Validating That Passwords Are Complex (Visual Basic)

Demonstrates how to check for strong-password characteristics and update a string parameter with information about which checks a password fails.

Encrypting and Decrypting Strings in Visual Basic

Shows how to use the DESCryptoServiceProvider class to encrypt and decrypt strings.

Manipulating Files and Folders in Visual Basic

Demonstrates how to use Visual Basic functions to determine information about a file, search for a string in a file, and write to a file.

Manipulating Files Using .NET Framework Methods

Demonstrates how to use .NET Framework methods to determine information about a file, search for a string in a file, and write to a file.

Persisting an Object in Visual Basic

Demonstrates how to create a simple object and persist its data to a file.

Walkthrough: Test-First Support with the Generate From Usage Feature

Demonstrates how to do test-first development, in which you first write unit tests and then write the source code to make the tests succeed.

.NET samples and tutorials

5/21/2019 • 5 minutes to read • Edit Online

The .NET documentation contains a set of samples and tutorials that teach you about .NET. This topic describes how to find, view, and download .NET Core, ASP.NET Core, and C# samples and tutorials. Find resources to learn the F# programming language on the F# Foundation's site. If you're interested in exploring C# using an online code editor, start with this interactive tutorial and continue with C# interactive tutorial. For instructions on how to view and download sample code, see the Viewing and downloading samples section.

.NET Core

Samples

Unit Testing in .NET Core using dotnet test

This guide shows you how to create an ASP.NET Core web app and associated unit tests. It starts by creating a simple web service app and then adds tests. It continues with creating more tests to guide implementing new features. The completed sample is available in the dotnet/samples repository on GitHub.

Tutorials

Writing .NET Core console apps using the CLI tools: A step-by-step guide

This guide shows you how to use the .NET Core CLI tooling to build cross-platform console apps. It starts with a basic console app and eventually spans multiple projects, including testing. You add features step-by-step, building your knowledge as you go. The completed sample is available in the dotnet/samples repository on GitHub.

Writing Libraries with Cross Platform Tools

This sample covers how to write libraries for .NET using cross-platform CLI tools. These tools provide an efficient and low-level experience that works across any supported operating system. The completed sample is available in the dotnet/samples repository on GitHub.

ASP.NET Core

See the ASP.NET Core tutorials. Many articles in the ASP.NET Core documentation have links to samples written for them.

C# language

Samples

Iterators

This sample demonstrates the syntax and features for creating and consuming C# iterators. The completed sample is available in the dotnet/samples repository on GitHub.

Indexers

This sample demonstrates the syntax and features for C# indexers. The completed sample is available in the dotnet/samples repository on GitHub.

Delegates and Events

This sample demonstrates the syntax and features for C# delegates and events. The completed sample is available in the dotnet/samples repository on GitHub. A second sample focused on events is also in the same repository.

Expression Trees

This sample demonstrates many of the problems that can be solved by using Expression Trees. The completed sample is available in the dotnet/samples repository on GitHub.

LINQ Samples

This series of samples demonstrate many of the features of Language Integrated Query (LINQ). The completed sample is available in the dotnet/samples repository on GitHub.

Managed COM server Sample

The COM server sample demonstrates the creation of a managed COM server and how it can be globally registered or consumed via RegFree COM.

Microsoft Office PIA Sample

The ExcelDemo sample demonstrates the consumption of Microsoft Office PIAs in .NET Core.

Tutorials

Console Application

This tutorial demonstrates Console I/O, the structure of a console app, and the basics of the task-based asynchronous programming model. The completed sample is available in the dotnet/samples repository on GitHub.

REST Client

This tutorial demonstrates web communications, JSON serialization, and object-oriented features of the C# language. The completed sample is available in the dotnet/samples repository on GitHub.

Working with LINQ

This tutorial demonstrates many of the features of LINQ and the language elements that support it. The completed sample is available in the dotnet/samples repository on GitHub.

Getting started with .NET Core on macOS using Visual Studio for Mac

This tutorial shows you how to build a simple .NET Core console app using Visual Studio for Mac.

Building a complete .NET Core solution on macOS using Visual Studio for Mac

This tutorial shows you how to build a complete .NET Core solution that includes a reusable library and unit testing.

Creating a .NET Core application that supports plugins

This tutorial shows you how to build a simple application on .NET Core that supports a plugin architecture. The completed sample is available in the dotnet/samples repository on GitHub.

Deploying to containers

Running ASP.NET MVC Applications in Windows Docker Containers

This tutorial demonstrates how to deploy an existing ASP.NET MVC app in a Windows Docker Container. The completed sample is available in the dotnet/samples repository on GitHub.

Running .NET Framework Console Applications in Windows Containers

This tutorial demonstrates how to deploy an existing console app in a Windows container. The completed sample is available in the dotnet/samples repository on GitHub.

Viewing and downloading samples

Many topics show source code and samples that are available for viewing or download from GitHub. To view a sample, just follow the sample link. To download the code, follow these instructions:

1. Download the repository that contains the sample code by performing one of the following procedures:
   - Download a ZIP of the repository to your local system. Un-ZIP the compressed archive.
   - Fork the repository and clone the fork to your local system. Forking and cloning permits you to make contributions to the documentation by committing changes to your fork and then creating a pull request for the official docs repository. For more information, see the .NET Documentation Contributing Guide and the ASP.NET Docs Contributing Guide.
   - Clone the repository locally. If you clone a docs repository directly to your local system, you won't be able to make commits directly against the official repository, so you won't be able to make documentation contributions later. Use the fork and clone procedure previously described if you want to preserve the opportunity to contribute to the documentation later.
2. Navigate within the repository's folders to the sample's location. The relative path to the sample's location appears in your browser's address bar when you follow the link to the sample.
3. To run a sample, you have several options:
   - Use the dotnet CLI tools: In a console window, navigate to the sample's folder and use dotnet CLI commands.
   - Use Visual Studio or Visual Studio for Mac: Open the sample by selecting **File > Open > Project/Solution** from the menu bar, navigate to the sample project folder, and select the project file (*.csproj* or *.fsproj*).
   - Use Visual Studio Code: Open the sample by selecting **File > Open Folder** from the menu bar and selecting the sample's project folder.
   - Use a different IDE that supports .NET Core projects.