



A FONDO C#

Tom Archer

**Mc
Graw
Hill** **McGraw-Hill
Profesional**

Microsoft®

A fondo C#

A fondo C#

Tom Archer

Traducción

NADIA BARAJAS ESTORNELL
Licenciada en Filología Inglesa

DIEGO BLANCO MORENO
Ingeniero en Informática

JAVIER G. RECUENCO
Ingeniero en Informática

Revisión técnica

BALTASAR FERNÁNDEZ MANJÓN
Profesor de Lenguajes y Sistemas Informáticos
Escuela Superior de Informática
Universidad Complutense de Madrid



**MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SYDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO**

A FONDO C#

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 2001, respecto a la primera edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.

Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)

<http://www.mcgraw-hill.es>
profesional@mcgraw-hill.es

Traducido de la primera edición en inglés de

Inside C#

ISBN: 0-7356-1288-9

Copyright © 2001, por Tom Archer

Publicado por McGraw-Hill/Interamericana de España por acuerdo con el editor original,
Microsoft Corporation. Redmond. Washington. EE.UU.

The Microsoft Windows logo is a trademark of Microsoft Corporation.

Microsoft Press is a registered trademark of Microsoft Corporation.

In Argentina, Microsoft Press is a trademark of Microsoft Corporation.

Microsoft, Microsoft Press, MSDN, .NET logo, Visual Basic, Visual C++, Visual Studio, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

ISBN: 84-481-3246-7

Depósito legal: M. 39.434-2001

Editor: Carmelo Sánchez González

Preimpresión: Marasán, S. A.

Impreso en Fareso, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Mi padre siempre fue mi crítico más duro. Esto puede ser bueno y malo, depende del punto de vista de uno y de las circunstancias. Sin embargo, siempre quiso que me superase y que hiciera las cosas bien. Recuerdo el día que le dije que estaba escribiendo un libro para Microsoft Press. Estaba impresionado y orgulloso de mí.

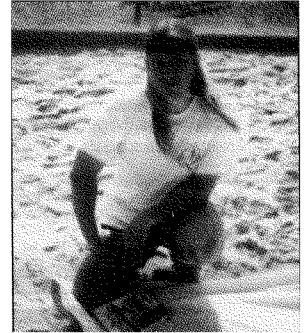
Sin duda alguna, fue uno de los días más felices de mi vida. Por desgracia, mi padre falleció el año pasado y no llegó a ver este libro terminado, pero sé que se habría alegrado mucho por mí.

*Como comienza la primera temporada de los Houston Astros sin mi padre (simplemente no es lo mismo), voy a dedicar este libro a la memoria de mi padre.
Te echo de menos, viejo.*

Tom Archer

Además de su hobby de ayudar a colegas desarrolladores por medio de webs para desarrolladores (CodeGuru y el recientemente creado TheCodeChannel), Tom Archer también hace consultoría sobre desarrollo en Microsoft Visual C++ y C#. Entre sus clientes se encuentran IBM, AT&T, Equifax y PeachTree Software. Uno de los logros por los que se siente más orgulloso es ser el programador jefe en dos aplicaciones ganadoras de premios (en IBM y PeachTree). Aunque generalmente huye por su timidez de las masas, actualmente enseña programación C# y .NET a pequeños grupos (individuales o empresas) para poder centrarse en una formación más personal.

Actualmente, Tom vive en Atlanta, Georgia. Es un ávido jugador de billar (sobre todo de las especialidades a nueve bolas y billar de bote-lla). Si por casualidad está alguna vez en Atlanta o sus alrededores y le apetece una buena partida, o simplemente intentar unas cuantas carambolas, mándele un e-mail. Puede localizar a Tom en el web <http://www.TheCodeChannel.com>.



Resumen del contenido

Parte I: SENTANDO LAS BASES

1. Fundamentos de la programación orientada a objetos 3
2. Introducción a Microsoft .NET 21
3. Hola, C# 33

Parte II: FUNDAMENTOS DE C#

4. El sistema de tipos 55
5. Clases 67
6. Métodos 97
7. Propiedades, arrays e indizadores 115
8. Atributos 133
9. Interfaces 151

Parte III: CÓMO ESCRIBIR CÓDIGO

10. Expresiones y operadores 175
11. Control del flujo de programa 199
12. Manejo de errores con excepciones 227
13. Sobrecarga de operadores y conversiones definidas por el usuario 247
14. Delegados y manejadores de eventos 259

Parte IV: C# AVANZADO

15. Programación multihilo 279
16. Cómo obtener información sobre metadatos con Reflection 301
17. Cómo interoperar con código no gestionado 317
18. Cómo trabajar con ensamblajes 339

Contenido

Prefacio	xvii
Introducción	xxi

Parte I: SENTANDO LAS BASES

1. Fundamentos de la programación orientada a objetos	3
<i>Todo es un objeto</i>	<i>4</i>
Objetos frente a clases	8
Instanciación	9
<i>Los tres principios de los lenguajes orientados a objetos</i>	<i>10</i>
Encapsulación	10
Herencia	13
Polimorfismo	16
2. Introducción a Microsoft .NET	21
<i>La plataforma .NET de Microsoft</i>	<i>21</i>
<i>El Framework .NET</i>	<i>22</i>
Windows DNA y .NET	22
El Common Language Runtime	23
Las bibliotecas de clases del Framework .NET	24
Microsoft Intermediate Language y los JITters	25
Sistema de tipos unificado	27
Metadatos y reflexión	28
Seguridad	29
Despliegue	29
Interoperabilidad con código no gestionado	30

3. Hola, C#	33
<i>Escribiendo su primera aplicación en C#</i>	33
Escoger un editor	33
«Hola, mundo»	36
Utilizar el compilador mediante línea de comandos	36
Ejecutar la aplicación	38
<i>Revisión del código</i>	38
Programación en un único paso	38
Clases y miembros	39
El método <i>Main</i>	39
El método <i>System.Console.WriteLine</i>	40
Espacio de nombres y la directiva <i>using</i>	40
Esqueleto de código	41
<i>¡Algo ha ido mal!</i>	42
Errores en tiempo de compilación	43
<i>Investigando a fondo ILDASM</i>	44
«Hola, mundo» en MSIL	44
<i>Guías de programación de C#</i>	47
Cuándo definir sus propios espacios de nombres	47
Guías de nomenclatura	48
Convenciones de nomenclatura estándar	48

Parte II: FUNDAMENTOS DE C#

4. El sistema de tipos	55
<i>Todo es un objeto</i>	55
<i>Tipos valor y tipos referencia</i>	56
Tipos valor	56
Tipos referencia	57
<i>Empaquetado y desempaquetado</i>	57
<i>La raíz de todos los tipos: System.Object</i>	58
<i>Tipos y alias</i>	59
<i>Conversión entre tipos</i>	59
<i>Espacios de nombres</i>	62
La palabra reservada <i>using</i>	62
<i>Beneficios de CTS</i>	64
Interoperabilidad del lenguaje	64
Jerarquía de objetos de raíz única	64
Seguridad de tipo	65
5. Clases	67
<i>Cómo definir clases</i>	67
<i>Miembros de clase</i>	68
<i>Modificadores de acceso</i>	69

<i>El método Main</i>	70
Argumentos de línea de comandos	71
Valores de retorno	71
Múltiples métodos <i>Main</i>	72
<i>Constructores</i>	73
Miembros estáticos y miembros de instancia	75
Inicializadores de constructor	76
<i>Constantes frente a campos de sólo lectura</i>	79
Constantes	79
Campos de sólo lectura	80
<i>Limpieza de objetos y gestión de recursos</i>	81
Un poquito de historia	82
Finalización determinista	83
Rendimiento	85
La solución perfecta	90
La solución (casi) perfecta	90
El patrón de diseño Liberar	92
<i>Herencia</i>	92
Interfaces múltiples	94
Clases selladas	95
6. Métodos	97
Parámetros ref y out de los métodos	97
Sobrecarga de métodos	101
Parámetros de método variables	104
Métodos virtuales	105
Redefinición de métodos	105
Polimorfismo	106
Métodos estáticos	112
Acceso a los miembros de clase	113
7. Propiedades, arrays e indizadores	115
Propiedades como campos inteligentes	115
Cómo definir y utilizar propiedades	116
Lo que está haciendo realmente el compilador	117
Propiedades de sólo lectura	119
Cómo heredar propiedades	120
Utilización avanzada de propiedades	120
Arrays	121
Cómo declarar arrays	122
Ejemplo de array unidimensional	122
Arrays multidimensionales	123
Cómo consultar el rango	125
Arrays irregulares	126
Cómo tratar objetos como arrays utilizando indizadores	127

Cómo definir indizadores	128
Ejemplo de indizador	129
Guías de diseño	131

8. Atributos 133

<i>Presentación de los atributos</i>	134
<i>Cómo definir atributos</i>	135
<i>Cómo preguntar sobre atributos</i>	136
Atributos de clase	136
Atributos de método	138
Atributos de campo	140
<i>Parámetros de atributos</i>	142
Parámetros posicionales y parámetros con nombre	142
Errores comunes con parámetros con nombre	144
Tipos de parámetros de atributo válidos	144
<i>El atributo AttributeUsage</i>	145
Cómo definir el objetivo de un atributo	145
Atributos de un solo uso y multiuso	147
Cómo especificar las reglas de herencia de atributos	148
<i>Identificadores de atributo</i>	148

9. Interfaces 151

<i>La utilización de la interfaz</i>	152
<i>Cómo declarar interfaces</i>	153
<i>Cómo implementar interfaces</i>	154
Cómo consultar la implementación mediante <i>is</i>	156
Cómo consultar la implementación mediante <i>as</i>	159
<i>Calificación explícita del nombre del miembro de la interfaz</i>	162
Ocultando nombre con interfaces	162
Cómo evitar la ambigüedad de nombres	165
<i>Interfaces y herencia</i>	168
<i>Cómo combinar interfaces</i>	171

Parte III: CÓMO ESCRIBIR CÓDIGO

10. Expresiones y operadores 175

<i>Definición de operadores</i>	175
<i>Precedencia de operadores</i>	176
Cómo determina C# la precedencia	177
Asociatividad por la izquierda y por la derecha	177
Utilización práctica	178
<i>Operadores en C#</i>	179
Operadores primarios de expresión	179
Operadores matemáticos	184

Operadores relacionales	191
Operadores de asignación	194
11. Control del flujo de programa	199
<i>Instrucciones de selección</i>	199
La instrucción <i>if</i>	199
La instrucción <i>switch</i>	204
<i>Instrucciones de iteración</i>	208
Instrucción <i>while</i>	208
La instrucción <i>do/while</i>	210
La instrucción <i>for</i>	211
La instrucción <i>foreach</i>	214
<i>Cómo bifurcar con instrucciones de salto</i>	216
La instrucción <i>break</i>	216
La instrucción <i>continue</i>	219
La infame instrucción <i>goto</i>	220
La instrucción <i>return</i>	225
12. Manejo de errores con excepciones	227
<i>Visión general del manejo de excepciones</i>	227
<i>Sintaxis básica de manejo de excepciones</i>	228
Cómo lanzar una excepción	229
Cómo capturar una excepción	229
Cómo relanzar una excepción	230
Poniendo orden con <i>finally</i>	231
<i>Comparación de técnicas de manejo de errores</i>	232
Beneficios del manejo de excepciones frente a los códigos de retorno ..	233
Cómo manejar errores en el contexto adecuado	234
Cómo mejorar la legibilidad del código	236
Cómo lanzar excepciones desde los constructores	237
<i>La Clase System.Exception</i>	237
Cómo construir un objeto <i>Exception</i>	238
Cómo utilizar la propiedad <i>StackTrace</i>	240
Cómo capturar múltiples tipos de excepciones	241
Cómo derivar sus propias clases <i>Exception</i>	242
<i>Cómo diseñar su código con manejo de excepciones</i>	243
Consideraciones de diseño con el bloque <i>try</i>	244
Consideraciones de diseño con el bloque <i>catch</i>	245
13. Sobrecarga de operadores y conversiones definidas por el usuario	247
<i>Sobrecarga de operadores</i>	247
Sintaxis y ejemplo	248
Operadores que se pueden sobrecargar	251
Restricciones en la sobrecarga de operadores	251
Guías de diseño	251

<i>Conversiones definidas por el usuario</i>	252
<i>Sintaxis y ejemplo</i>	252
14. Delegados y manejadores de eventos	259
<i>Delegados como métodos de devolución de llamada</i>	259
<i>Cómo definir delegados como miembros estáticos</i>	263
<i>Cómo crear delegados sólo cuando se necesiten</i>	264
<i>Composición de delegados</i>	266
<i>Cómo definir eventos con delegados</i>	271

Parte IV: C# AVANZADO

15. Programación multihilo	279
<i>Conceptos básicos sobre hilos</i>	279
<i>Hilos y multitarea</i>	280
<i>Alternancia de contexto (Context switching)</i>	280
<i>Una aplicación multihilo en C#</i>	281
<i>Cómo trabajar con hilos</i>	282
<i>AppDomain</i>	282
<i>La clase Thread</i>	283
<i>Planificación de los hilos</i>	286
<i>Seguridad y sincronización en los hilos de ejecución</i>	290
<i>Cómo proteger el código usando la clase Monitor</i>	290
<i>Utilización de bloqueos de monitor con la instrucción lock de C#</i>	294
<i>Cómo sincronizar código usando la clase Mutex</i>	295
<i>Seguridad en los hilos de ejecución y las clases .NET</i>	298
<i>Guías para usar hilos de ejecución</i>	298
<i>Cuándo usar hilos</i>	298
<i>Cuándo no usar hilos de ejecución</i>	299
16. Cómo obtener información sobre metadatos con Reflection	301
<i>La jerarquía de la API Reflection</i>	301
<i>La clase Type</i>	302
<i>Obteniendo el tipo de una instancia</i>	302
<i>Obteniendo el tipo de un nombre</i>	302
<i>Interrogando a los tipos</i>	303
<i>Cómo trabajar con ensamblajes y módulos</i>	305
<i>Iterar a través de los tipos de un ensamblaje</i>	306
<i>Cómo conseguir una lista de los módulos de un ensamblaje</i>	308
<i>Enlace en tiempo de ejecución con 'reflection'</i>	310
<i>Cómo crear y ejecutar código en tiempo de ejecución</i>	312
17. Cómo interoperar con código no gestionado	317
<i>Servicios de invocación de plataforma</i>	317
<i>Cómo declarar una función exportada desde la DLL</i>	318

Cómo utilizar funciones de devolución de llamada en C#	320
Envío de datos y <i>PInvoke</i>	322
<i>Escribiendo código inseguro</i>	323
Cómo utilizar punteros en C#	323
La instrucción <i>fixed</i>	325
<i>Interoperabilidad COM</i>	326
Un mundo indómito	326
Poniéndonos en marcha	327
Cómo generar metadatos desde una biblioteca de tipos COM	328
Enlace en tiempo de compilación a componentes COM	331
Cómo utilizar descubrimiento dinámico de tipos para seleccionar interfa- ces COM	332
Enlace en tiempo de ejecución a componentes COM	334
Modelos de proceso COM	336
18. Cómo trabajar con ensamblajes	339
<i>Un vistazo a los ensamblajes</i>	339
Datos del manifiesto	340
<i>Beneficios de los ensamblajes</i>	341
Paquetización de ensamblajes	341
Despliegue de ensamblajes	341
Versiones de ensamblajes	342
<i>Cómo construir ensamblajes</i>	342
Cómo crear ensamblajes con módulos múltiples	343
<i>Cómo crear ensamblajes compartidos</i>	345
<i>Cómo trabajar con la caché global de ensamblajes</i>	346
Cómo examinar la caché	347
<i>Cómo gestionar versiones de ensamblajes</i>	349
QFE y la política de gestión de versiones por defecto	352
Cómo crear un archivo de configuración en modo seguro	352
Índice	355

Prefacio

He pasado toda mi carrera en Microsoft trabajando para aumentar la experiencia del desarrollador, normalmente centrándome en el incremento de su productividad. Mi trabajo abarca una amplia variedad de productos y tecnologías, pero nunca he estado tan ilusionado con el trabajo que hemos hecho para aumentar la experiencia del desarrollador como lo estoy ahora. La extensión y la profundidad de las entregas de las tecnologías Microsoft.Net son asombrosas. Estamos creando un gran lenguaje nuevo, derribando las barreras que tradicionalmente han dividido a los desarrolladores en mundos de lenguajes distintos a la vez que distantes y estamos haciendo posible también que los sitios Web cooperen para cubrir las necesidades de los usuarios. Cualquiera de los aspectos anteriores sería interesante por sí mismo, pero la combinación es verdaderamente irresistible.

Vamos a echar un vistazo a los pilares que conforman .Net y algunas tecnologías relacionadas:

- **C#, un nuevo lenguaje.** C# es el primer lenguaje orientado a componentes en la familia de lenguajes C y C++. Es un lenguaje de programación simple, moderno, orientado a objetos y con un sistema de tipos seguro derivado de C y C++. C# combina la alta productividad de Microsoft Visual Basic y la eficacia bruta de C++.
- **Common Language Runtime.** El Common Language Runtime (CLR) de alto rendimiento incluye un motor de ejecución, un recolector de basura, compilación just-in-time, un sistema de seguridad y un framework de clases muy amplio (el Framework .NET). El CLR fue diseñado desde cero para poder dar soporte a múltiples lenguajes.
- **Common Language Specification.** La Common Language Specification (CLS) describe un nivel común de funcionalidad de lenguaje. El nivel relativamente alto del denominador común del CLS permite la creación de un grupo de lenguajes compatibles con CLS. Cada miembro de este grupo disfruta de un doble beneficio: acceso completo a la funcionalidad del Framework .Net y una rica interoperabi-

lidad con otros lenguajes que también se ajustan al CLS. Por ejemplo, una clase Visual Basic puede heredar de una clase C# y redefinir sus métodos virtuales.

- **Un grupo rico de lenguajes que se orientan al CLR.** Los lenguajes que proporciona Microsoft orientados al CLR incluyen Visual Basic, Visual C++ con extensiones gestionadas, Visual C# y Jscript. Terceras partes están proporcionando otros muchos lenguajes —¡demasiados para enumerarlos aquí!
- **Servicios Web.** La World Wide Web de hoy está formada ante todo por sitios individuales. Mientras un usuario puede visitar numerosos sitios para llevar a cabo una determinada tarea, como concertar viajes para un grupo de personas, los sitios Web no cooperan entre sí. La próxima generación de Web se basará en redes cooperantes de sitios Web. La razón es simple: los sitios Web cooperantes pueden hacer un trabajo mejor para cubrir las necesidades de los usuarios. Las tecnologías de servicios Web de Microsoft facilitan la cooperación entre los sitios Web al capacitar la comunicación mediante protocolos estándares basados en XML, que son independientes tanto del lenguaje como de la plataforma. Muchos servicios importantes de la Red se basarán en C# y en el CLR que se ejecute en Windows, pero la arquitectura está verdaderamente abierta.
- **Visual Studio.NET.** Visual Studio.NET enlaza todas estas piezas y hace más fácil crear una amplia gama de componentes, aplicaciones y servicios en variedad de lenguajes de programación.

Ahora que hemos visto algunas tecnologías importantes relacionadas con C#, vamos a ver más de cerca el propio C#. Los desarrolladores han invertido mucho en el lenguaje que han elegido y por eso la responsabilidad de un nuevo lenguaje está en que pruebe su valor a través de una combinación de conservación sencilla, mejora incremental y una innovación previsor.

CONSERVACIÓN SENCILLA

Hipócrates dijo: «Ten por costumbre dos cosas: ayudar, o al menos no dañar». La parte del «no dañar» jugó un papel muy importante en nuestro diseño del C#. Si una característica de C o de C++ resolviera adecuadamente un problema, lo mantendríamos sin modificación alguna. Más aún, C# toma prestadas de C y de C++ áreas fundamentales tales como expresiones, instrucciones y sobre todo la sintaxis. Dado que un programa típico está constituido en gran parte por estas características, los desarrolladores de C y C++ en seguida se sienten cómodos con C#.

MEJORA INCREMENTAL

Se han hecho muchas mejoras incrementales —demasiadas para mencionarlas en este breve prefacio—, pero merece la pena llamar la atención sobre algunos cambios que eliminan algunos errores comunes y costosos en tiempo en C y C++:

- Las variables deben inicializarse antes de usarse, así se eliminan los defectos que se deben a variables no inicializadas.

- Las instrucciones como *if* o *while* requieren valores booleanos; así, un desarrollador que accidentalmente use el operador de asignación (=) en lugar del operador de igualdad (==), encontrará el error en tiempo de compilación.
- No se aceptan los casos sin cerrar en instrucciones *switch*; así, un desarrollador que accidentalmente omita un *break*, encontrará el error en tiempo de compilación.

INNOVACIÓN PREVISORA

Encontramos la innovación con más profundidad en el sistema de tipos de C#, que incluye los siguientes avances:

- El sistema de tipos de C# emplea gestión de memoria automática, liberando de ese modo a los desarrolladores de una gestión manual de memoria costosa en tiempo y propensa a errores. Al contrario que en la mayoría de los sistemas de tipos, el sistema de tipos de C# también permite la manipulación directa de tipos puntero y direcciones de memoria de objetos. (Estas técnicas de gestión de memoria manual sólo se permiten en ciertos contextos de seguridad).
- El sistema de tipos de C# está unificado; todo es un objeto. A través del uso innovador de conceptos como empaquetado y desempaquetado, C# llena el vacío entre tipos valor y tipos referencia, permitiendo que cualquier dato sea tratado como objeto.
- Las propiedades, métodos y eventos son fundamentales. Muchos lenguajes omiten el soporte intrínseco a propiedades y eventos, creando un desajuste innecesario entre el lenguaje y los frameworks asociados. Por ejemplo, si el framework soporta propiedades y el lenguaje no, incrementar una propiedad es poco natural [por ejemplo, `o.SetValue(o.GetValue()+1)`]. Si el lenguaje también soporta propiedades, la operación es sencilla (`o.Value++`).
- C# soporta atributos, que permiten la definición y uso de información declarativa sobre componentes. El poder definir nuevos tipos de información declarativa siempre ha sido una poderosa herramienta para los diseñadores de lenguajes. Ahora todos los desarrolladores de C# tienen esta capacidad.

A FONDO C#

En *A fondo C#*, Tom Archer establece la base introduciendo .NET y el CLR, explica los fundamentos de C# y profundiza en algunos conceptos avanzados del lenguaje C#. Su gran experiencia —como desarrollador o autor de libros sobre C++, J++ y Microsoft Windows— le permite explicar C# de tal manera que los lectores lo encontrarán ameno e informativo.

Lectores, espero que disfruten escribiendo sus primeros programas en C# y que sean los primeros de los muchos que escribirán en los años venideros.

Scott Wiltamuth
Miembro del equipo de diseño de C#
Microsoft Corporation

Introducción

POR QUÉ ESCRIBÍ ESTE LIBRO

Después de haber sido desarrollador durante veinte años —¡cada vez que lo pienso me siento más viejo!—, he alcanzado básicamente el punto en el que programar empezaba a cansarme un poco. No me malinterpreten: si fuera multimillonario y no necesitara trabajar, probablemente continuaría escribiendo código, porque verdaderamente disfruto haciéndolo más que con ninguna otra cosa. Sin embargo, estaba llegando al punto de pensar: «¡Se ha hecho todo!». Entonces llegaron Microsoft.NET y C#, y se abrió todo un mundo de posibilidades. He hablado con varios amigos que han tenido este mismo «redespertar» de carácter con la introducción de .NET. Tenemos esta nueva y excitante tecnología que por fin resuelve cosas con las que hemos estado luchando a brazo partido durante años (por ejemplo, entornos de desarrollo multilenguaje, los problemas de despliegue y gestión de versiones de sistemas grandes y complejos, etc.). Escribí este libro porque es emocionante escribir código otra vez. Porque otra vez me levanto por la mañana pensando en todo lo nuevo y fabuloso que voy a aprender en el día. Espero que según aprendan este lenguaje, compartan mi entusiasmo.

Cualquiera que escriba un libro sobre C# en el momento en el que éste se ha escrito ha tenido que aprender el lenguaje a la vez que escribía el libro. Si ha escrito una aplicación mientras se aprende el SDK o está utilizando el lenguaje —¿quién no lo ha hecho?—, sabe que puede ser una situación muy difícil. Ahora intente imaginar que escribe algo que ¡diez mil personas van a revisar después de que lo haya hecho! El mayor problema es que a mitad del proyecto —una vez que sabe lo que está haciendo— le entran las ganas y las prisas de rediseñar y repetir ¡todo entero! Obviamente, dentro de unos límites, eso no es práctico. En mi opinión, creo que este libro es una buena herramienta para aprender C#. Dado que yo lo aprendí mientras lo escribía, habrá muchas inconsistencias y algunas cosas que podría haber hecho mucho mejor. Sin embargo, si

tengo una oportunidad de hacer una segunda edición, puedo prometerle que usted y yo nos beneficiaremos de la evolución de mi curva personal de aprendizaje al haber escrito este texto.

Finalmente, me gustaría decir que agradezco cualquier información que se me facilitara con relación a este libro. No soy un «soy así de bueno porque escribo libros». Simplemente, soy un tipo normal que tuvo la gran suerte de tener la oportunidad de escribir este libro. Siempre estoy abierto a aprender de otros, y de hecho me encanta hacerlo. Pueden ponerse en contacto conmigo en <http://www.TheCodeChannel.com>.

QUIÉN DEBERÍA LEER ESTE LIBRO

Este libro va dirigido a toda persona que quiera iniciarse en el desarrollo con C# y .NET. Como ya he mencionado, esta es una plataforma emocionante y es la onda del futuro en lo que al desarrollo distribuido de Microsoft Windows se refiere. Este libro da por hecho que el lector tiene un conocimiento base de alguna de las familias de lenguajes C: C, C++ o Java. Creo que sólo hay otro prerequisite más, y es el deseo de aprender y explorar nuevas dimensiones en la escritura de aplicaciones. ¡Dado que usted tiene este libro en las manos, seguramente tenga ese deseo!

ORGANIZACIÓN DE ESTE LIBRO

Este libro ha sido cuidadosamente organizado en varias secciones, ordenadas secuencial y lógicamente; cada sección consiste en un grupo de capítulos dirigidos a una categoría específica de C# o al desarrollo de .NET.

El libro comienza con la Parte I, «Sentando las bases», una sección para programadores que se inician en C# y para aquellos que sean nuevos en el entorno de .NET. Los capítulos de esta sección tienen una introducción a .NET y dan ejemplos de cómo crear y probar sus primeras aplicaciones de C#.

En la Parte II, «Fundamentos de clase C#», presento las bases para definir y trabajar con clases en C#. Los capítulos de esta sección van orientados a darle una base sólida sobre qué miembros están soportados en C# (enumeraciones, propiedades, arrays y demás) y sobre cómo definirlos y usarlos en una aplicación C#.

Aunque ya llevará varios capítulos escribiendo código en lo que se refiere a definir miembros de clases específicas, en la Parte III, «Cómo escribir código», empezará a ver diferentes aspectos de tareas tales como controlar flujo de programa, manejo errores (con excepciones) y escribir manejadores de eventos con delegados.

El libro concluye con la Parte IV, «C# avanzado». Siendo un adicto a la informática típico, esta es la sección que más disfruté escribiendo. Incluye capítulos sobre programación multihilo, reflexión, trabajar con código con memoria no gestionada (incluyendo la interoperabilidad COM) y la gestión de versiones.

SOBRE EL CD COMPLEMENTARIO

Este libro contiene un CD complementario. Si tiene la Reproducción automática habilitada en Windows, verá una pantalla inicial cuando inserte el CD en el reproductor de CD-ROM que le proporcionará opciones de instalación. Para arrancar esta pantalla manualmente, ejecute StartCD desde el directorio raíz del CD. El programa StartCD le proporciona enlaces para el eBook contenido en el CD, un programa de instalación para los archivos de ejemplo del libro y la versión Beta 2 del Microsoft.Net Framework SDK, que necesitará para compilar y ejecutar los programas de ejemplo.

Los programas de ejemplo para el libro se encuentran en la carpeta Code. Puede ver los ejemplos desde el CD o puede instalarlos en el disco duro usando el instalador del StartCD.

NOTA Si no pudiera examinar los archivos de la carpeta de ejemplos, puede que sea porque tiene un reproductor de CD antiguo que no permite nombres de archivos largos. Si es así, para examinar los archivos debe instalarlos en el disco duro ejecutando el programa de instalación.

REQUISITOS DEL SISTEMA

Para sacarle el máximo partido a este libro, le recomiendo trabajar a través de los ejemplos de aplicaciones según vaya leyendo cada capítulo. Para hacer esto, necesitará instalar el último .NET Framework SDK *. En el momento de escribir esto, incluye el entorno de ejecución .NET y el compilador C#. Además, he evitado a propósito el uso de Visual Studio .NET para centrarme en el lenguaje y en el entorno de ejecución y no poner restricciones en su entorno de desarrollo particular. Por consiguiente, todos los ejemplos de este libro se compilarán y ejecutarán desde la línea de comandos.

AGRADECIMIENTOS

Antes que nada, me gustaría agradecer la ayuda de mi editor, Devon Musgrave. No pretendo de ninguna manera ser un «escritor»: Soy un programador que resulta que quiere ayudar a otros programadores, y escribir libros es una manera de hacerlo. Sin la ayuda de Devon, ajustando y dando forma a lo que yo decía para convertirlo en lo que realmente quería decir, este texto no sería en modo alguno legible. ¡Gracias, Devon!

También quiero dar las gracias a Brian Johnson, el editor técnico del libro. Brian fue imprescindible cuando hubo que hacer coincidir el texto del libro con las aplicaciones de demo. También fue muy útil al ayudar a superar varios cambios de última hora en el

* Todos los ejemplos de código de este libro se han compilado utilizando la Beta 2 del .NET Framework SDK (*N. de los t.*).

compilador cuando intentábamos imprimir. Escribí este libro con una versión beta de C#. En el momento en que el compilador se libere al público en general, una o dos de las demos del libro pueden no funcionar. Sin embargo, nada de esto es culpa de Brian, ya que fue extremadamente meticuloso al probar todas y cada una de las demos. ¡Gracias, Brian!

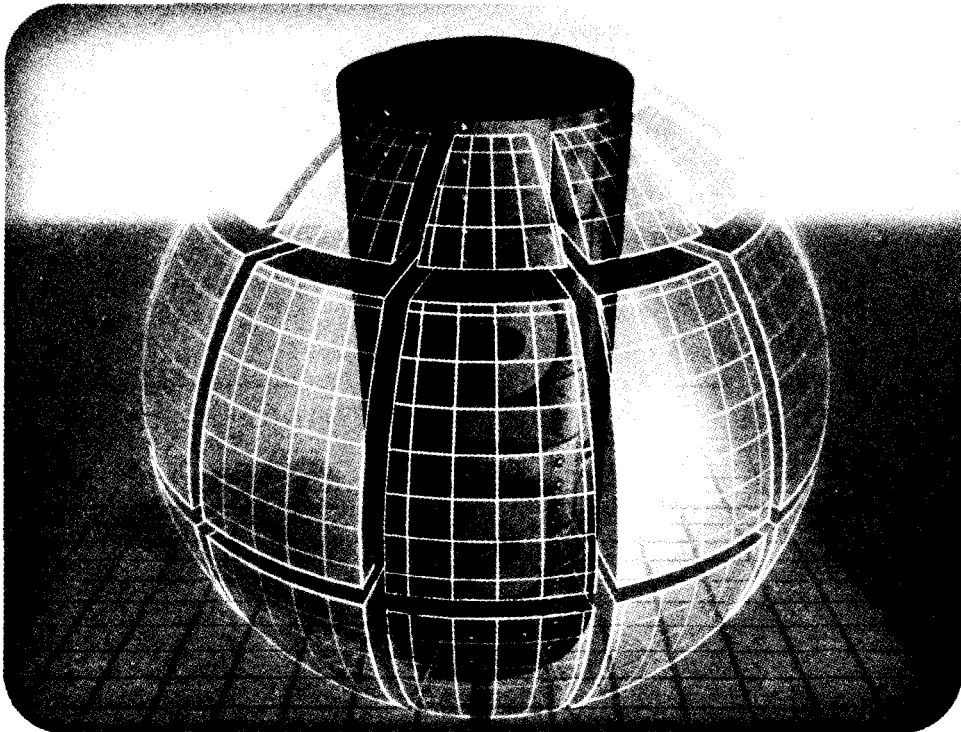
Hay dos personas más de Microsoft Press que también merecen mi agradecimiento: Anne Hamilton y Danielle Bird. Estas dos personas fueron responsables de levantar este proyecto y de darme la oportunidad de escribir este libro. Su fe en mí me ayudó en algunos momentos difíciles del libro, por lo que aprecio mucho su apoyo. ¡Gracias!

También quiero dar las gracias a los siguientes empleados de Microsoft que fueron pacientes con mis preguntas cuando intentaba aprender el lenguaje C# y el .NET BCL: Joe Nalewabau (que respondió algunas de las primeras preguntas sobre C#), Brian Harry (que me consiguió la información sobre la finalización determinista) y Steven Pratschner (que aportó su ayuda para la caché global de ensamblaje y el versionado de ensamblajes). También, ¡muchas gracias a Scott Wiltamuth por su maravilloso Prefacio!

Finalmente, quiero dar las gracias a Aravind Corera, que ayudó tremendamente en el capítulo sobre la escritura de código de memoria no gestionada. Fue su gran trabajo en la sección del COM lo que hace que ese capítulo sea especial. Un gran trabajo, Aravind. ¡Espero que podamos trabajar juntos otra vez!

Parte I

SENTANDO LAS BASES



Capítulo 1

Fundamentos de la programación orientada a objetos

La finalidad de este capítulo es guiarle a través de la terminología de la programación orientada a objetos (POO) y hacerle entender la importancia de los conceptos orientados a objetos en la programación. Muchos lenguajes, como C++ y Microsoft Visual Basic, se dice que «soportan objetos», pero pocos lenguajes realmente soportan todos los principios que constituyen la programación orientada a objetos. C# es uno de estos lenguajes: fue diseñado desde la base para ser un lenguaje verdaderamente orientado a objetos y basado en componentes. Así que, para sacar el máximo de este libro, necesita tener una buena base de los conceptos aquí presentados.

Sé que normalmente los lectores pasan por alto los capítulos conceptuales como éste porque quieren profundizar directamente en el código, pero a menos que se considere un «gurú del objeto», le animo a que lea este capítulo. Aquellos que estén familiarizados de alguna manera con la programación orientada a objetos, deberían beneficiarse al leerlo. También hay que tener en cuenta que los capítulos que siguen a éste harán referencia a la terminología y conceptos que se utilizan aquí.

Como ya he dicho, muchos lenguajes dicen estar orientados a objetos y basados en objetos, pero verdaderamente pocos lo están. C++ no lo está, a causa del innegable e insalvable hecho de que sus raíces están profundamente situadas en el lenguaje C. Demasiados ideales de POO han tenido que ser sacrificados en C++ para soportar el legado del código C. Incluso el lenguaje Java, tan bueno como es, tiene algunas limitaciones como lenguaje orientado a objetos. Especialmente nos referimos al hecho de que Java tiene *tipos primitivos* y *tipos objeto* que se tratan y se comportan de manera muy diferente. Sin embargo, este capítulo no se centra en comparar la fidelidad de diferentes lenguajes a los principios de la POO. Es más, este capítulo presentará una clase particular objetiva y agnóstica con relación a un lenguaje en particular sobre los mismos principios de la POO.

Antes de que empecemos, tendríamos que añadir que la programación orientada a objetos es mucho más que una frase comercial (aunque haya llegado a ser así por culpa de algunas personas), una nueva sintaxis o una nueva interfaz de programación de aplicación (API). La programación orientada a objetos es un conjunto completo de conceptos e ideas. Es una manera de pensar en el problema al que va dirigido un programa de ordenador y de afrontarlo de modo más intuitivo e incluso más productivo.

Mi primer empleo implicaba usar el lenguaje Pascal para programar el informe de la oficina de correos y las aplicaciones de itinerario para Holiday on Ice. Según fui cambiando de trabajos y de aplicaciones, programé en PL/I y RPG III (y RPG/400). Después de unos cuantos años más, empecé a programar aplicaciones en lenguaje C. En cada uno de esos ejemplos, era capaz de aplicar fácilmente el conocimiento que había adquirido en la experiencia anterior. La curva de aprendizaje para cada lenguaje sucesivo era más corta en comparación con la complejidad del lenguaje que estaba aprendiendo. Esta es la razón por la cual, hasta que empecé a programar en C++, todos los lenguajes que había usado eran lenguajes procedimentales que se diferenciaban principalmente sólo en la sintaxis.

Sin embargo, si usted es nuevo en la programación orientada a objetos, debe estar prevenido: *¡la experiencia anterior con otros lenguajes no orientados a objetos no le ayudará aquí!* La programación orientada a objetos es una manera diferente de pensar en cómo diseñar y programar soluciones a los problemas. De hecho, los estudios han demostrado que la gente que es nueva en programar aprende lenguajes orientados a objetos más rápidamente que los que empezamos con lenguajes procedimentales como Basic, COBOL y C. Estos individuos no tienen que «desaprender» ningún hábito procedimental que pueda estorbar su entendimiento de la POO. Empiezan con la pizarra limpia. Si ha estado programando durante muchos años con lenguajes procedimentales y C# es su primer lenguaje orientado a objetos, el mejor consejo que podemos darle es que mantenga la mente abierta y que lleve a cabo las ideas que presentamos aquí antes de levantar las manos y decir: «Puedo conseguir esto en [inserte el lenguaje procedimental de su elección]». Cualquiera que haya llegado de un contexto procedimental a la programación orientada a objetos ha pasado a través de esta curva de aprendizaje, y bien lo vale. Los beneficios de programar con un lenguaje orientado a objetos son incalculables, tanto en términos de escribir código más eficientemente como en tener un sistema que pueda ser fácilmente modificado y extendido una vez escrito. Sólo podría no parecer así al principio. Sin embargo, casi veinte años de desarrollo de software (incluyendo los últimos ocho con lenguajes orientados a objetos) me han enseñado que los conceptos de la POO, *cuando se aplican correctamente*, realmente cumplen su promesa. Sin más, vamos a remangarnos y ver de qué va todo ese lío.

TODO ES UN OBJETO

En un lenguaje orientado a objetos verdadero, toda entidad del dominio del problema se expresa a través del concepto de *objetos*. (Observe que en este libro usaré la definición Coad/Yourdon para «dominio del problema»; esto es, que un dominio del problema es el problema que intenta resolver, en términos de sus complejidades específicas, terminologías, retos, etc.). Como puede adivinar, los objetos son la idea central detrás de la pro-

gramación orientada a objetos. La mayoría de nosotros no vamos pensando en términos de estructuras, paquetes de datos, invocaciones a función ni punteros; en cambio, pensamos típicamente en términos de objetos. Veamos un ejemplo.

Si estuviera escribiendo una aplicación de facturación y necesitara cuadrar las líneas de detalle de la factura, ¿cuál de los siguientes enfoques mentales sería el más intuitivo desde la perspectiva del cliente?

- **Enfoque no orientado a objeto.** Tendré acceso a la estructura de datos representando una cabecera de factura. Esta estructura de cabecera de factura también incluirá una lista doblemente enlazada de estructuras de detalle de factura, cada una de las cuales contiene una línea de cantidad total. Aun así, para obtener el total de una factura, necesito declarar una variable llamada algo así como *totalInvoiceAmount* e inicializarla a 0, conseguir un puntero a la estructura de la cabecera de la factura, conseguir la cabecera de la lista enlazada de líneas de detalles y después recorrer la lista enlazada de líneas de detalles. Según lea cada estructura de línea de detalle, obtendré la variable miembro que contiene el total para esa línea e incrementaré mi variable *totalInvoiceAmount*.
- **Enfoque orientado a objeto.** Tendré un objeto factura y enviaré un mensaje a ese objeto para preguntarle por la cantidad total. No necesito pensar cómo se almacena la información internamente en el objeto, como tuve que hacer en la estructura de datos no orientada a objetos. Simplemente trato el objeto de manera natural, haciéndole preguntas por medio de mensajes. (El grupo de mensajes que un objeto puede procesar se llama colectivamente *interfaz* del objeto. En el siguiente párrafo explicaré por qué pensar en términos de interfaz mejor que en implementación, como he hecho aquí, es justificable en el enfoque orientado a objeto).

Obviamente, el enfoque orientado a objeto es más intuitivo y más cercano a cómo muchos de nosotros pensaríamos en la manera de afrontar un problema. En la segunda solución, el objeto factura probablemente recorra una *colección* de objetos de detalles de factura, enviando un mensaje a cada una preguntándoles la cantidad de cada línea. Sin embargo, si lo que está buscando es el total, *no se preocupe por cómo está hecho*. No se preocupe, porque uno de los principales dogmas de la programación orientada a objetos es la *encapsulación* —la habilidad de un objeto para esconder sus datos y métodos internos y de presentar una interfaz que hace, hablando desde el punto de vista del programa, accesibles las partes importantes del objeto. Los procedimientos internos sobre cómo lleva a cabo un objeto su trabajo no son importantes mientras que ese objeto pueda desempeñar ese trabajo. Simplemente se le presenta una interfaz de objeto, y usted utiliza esa interfaz para hacer que el objeto desarrolle una tarea determinada en su lugar. (Más adelante explicaré los conceptos de encapsulación e interfaces en este capítulo). Lo importante aquí es que los programas escritos para simular los objetos del mundo real para el dominio del problema son mucho más fáciles de diseñar y escribir porque nos permiten pensar de un modo más natural.

Observe que el segundo enfoque requería un objeto para desarrollar el trabajo en su lugar; esto es, obtener el total de líneas de detalle. Un objeto no sólo contiene datos, como la estructura. Los objetos, por definición, comprenden datos y métodos que traba-

jan con esos datos. Esto significa que cuando trabajamos con el dominio del problema podemos hacer algo más que diseñar las estructuras de datos necesarias. También podemos echar un vistazo a qué métodos deberían asociarse con un objeto determinado para que el objeto sea enteramente una encapsulación de parte de la funcionalidad. Los ejemplos que siguen aquí y en las siguientes secciones ayudan a ilustrar este concepto.

NOTA Los fragmentos de código en este capítulo presentan los conceptos de programación orientada a objetos. Tenga en cuenta que mientras presentamos muchos ejemplos de fragmentos de código en C#, los conceptos en sí mismos son genéricos en la POO y no son específicos de ningún lenguaje de programación. Con intención de comparar en este capítulo, también presentaré ejemplos en C, que no está orientado a objetos.

Digamos que va a escribir una aplicación para calcular la paga de Amy, la única empleada nueva de la compañía. Usando C, para asociar determinados datos con un empleado, programaría algo parecido a lo siguiente:

```
struct EMPLOYEE
```

```
{
    char szFirstName[25];
    char szLastName[25];

    int iAge;

    double dPayRate;
};
```

Así es como calcularía la paga de Amy usando la estructura *EMPLOYEE*:

```
void main()
```

```
{
    double dTotalPay;

    struct EMPLOYEE*pEmp;
    pEmp = (struct EMPLOYEE*)malloc(sizeof(struct EMPLOYEE));

    if (pEmp)
    {
        pEmp->dPayRate = 100;

        strcpy(pEmp->szFirstName, "Amy");
        strcpy(pEmp->szLastName, "Anderson");
        pEmp->iAge = 28;

        dTotalPay = pEmp->dPayRate * 40;
        printf("La paga total para %s %s es %0.2f",
            pEmp->szFirstName, pEmp->szLastName, dTotalPay);
    }
    free(pEmp);
}
```

En este ejemplo, el código se basa en los datos contenidos en una estructura y algo de código externo (a esta estructura) que la utiliza. Y ¿cuál es el problema? El principal problema es de abstracción: el usuario de la estructura *EMPLOYEE* debe saber bastante sobre los datos que se necesitan del empleado. ¿Por qué? Digamos que más tarde quiere cambiar el modo en que se calcula el porcentaje de la paga de Amy. Por ejemplo, podría querer factorizar los impuestos y otros tipos de tasas cuando determine la paga neta. No sólo tendría que cambiar todo el código cliente que utiliza la estructura *EMPLOYEE*, sino que también necesitaría documentar —para cualquier futuro programador de su compañía— el hecho de que ha habido un cambio en el uso.

Ahora echemos un vistazo a la versión C# de este ejemplo:

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }
    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        //Calcular la paga aquí.
        return (payRate * (double)hoursWorked);
    }
}

class EmployeeApp
{
    public static void Main()
    {
        Employee emp = new Employee ("Amy", "Anderson", 28, 100);
        Console.WriteLine("\nLa paga de Amy es $" + emp.CalculatePay(40));
    }
}
```

En la versión C# del ejemplo del *EmployeeApp*, el usuario del objeto puede llamar simplemente al método *CalculatePay* para conseguir que éste calcule su propia paga. La ventaja de este enfoque es que el usuario ya no necesita preocuparse por los mecanismos internos de cómo se calcula la paga exactamente. Si en algún momento en el futuro decidiera modificar cómo se calcula la paga, esa modificación no tendrá impacto en el código existente. Este nivel de abstracción es uno de los beneficios básicos de utilizar objetos.

Ahora, se podría comentar con razón que se podría haber abstraído el código C del cliente creando una función de acceso a la estructura *EMPLOYEE*. Sin embargo, el hecho de que se tuviera que crear esta función completamente aparte de la estructura en la que se está trabajando es exactamente el problema. Cuando se utiliza un lenguaje orientado a objetos como C#, los datos del objeto y los métodos que operan en esos datos (su interfaz) siempre están juntos.

Tenga en cuenta que sólo los métodos de un objeto deberían modificar las variables de éste. Como puede ver en el ejemplo anterior, cada variable miembro de *Employee* se declara con el modificador de acceso *protected*, excepto en el método *CalculatePay*, que se define como *public*. Los modificadores de acceso se utilizan para especificar el nivel de acceso que el código derivado de clase y el cliente tienen con relación a un determinado miembro de clase. En el caso del modificador *protected*, una clase derivada tendría acceso al miembro pero no al código del cliente. El modificador *public* hace accesible el miembro tanto a las clases derivadas como al código de cliente. Entraremos en más detalles sobre modificadores de acceso en el Capítulo 5, «Clases», pero lo que hay que recordar por ahora es que los modificadores facilitan proteger los miembros clave de la clase para que no se utilicen incorrectamente.

Objetos frente a clases

La diferencia entre una clase y un objeto es una fuente de confusión para los programadores nuevos en la terminología de la programación orientada a objetos. Para ilustrar la diferencia entre esos dos términos, vamos a hacer nuestro ejemplo *EmployeeApp* más realista asumiendo que estamos trabajando no con un solo empleado, sino con una compañía entera.

Usando el lenguaje C, podríamos definir un array de empleados —basado en la estructura *EMPLOYEE*— y empezar desde aquí. Como no sabemos cuántos empleados podríamos llegar a contratar algún día en nuestra compañía, podríamos crear este array con un número estático de elementos, por ejemplo 10.000. Sin embargo, dado que nuestra compañía actualmente sólo tiene a Amy como única empleada, esto no sería exactamente el uso más eficiente de recursos. En lugar de esto, normalmente crearíamos una lista enlazada de estructuras *EMPLOYEE* y reservaríamos memoria dinámicamente a medida que se necesitara en nuestra nueva aplicación de nómina.

Si se nos permite opinar, lo que estamos haciendo es exactamente lo que no deberíamos. Estamos consumiendo energía mental pensando sobre el lenguaje y sobre la máquina —en términos de cuánta memoria hay que asignar y dónde—, en lugar de concentrarnos en el dominio del problema. Al utilizar objetos, podemos centrarnos en la lógica del negocio en lugar de en las necesidades de la máquina para resolver el problema.

Hay muchas maneras de definir una clase y distinguirla de un objeto. Puede pensar en una clase como un tipo simple (simplemente como *char*, *int* o *long*) que tiene métodos asociados a él. Un objeto es un ejemplo de un tipo o una clase. Sin embargo, la definición que más me gusta es la que dice que una clase es un diseño de un objeto. Usted, como desarrollador, crea este diseño como un ingeniero crearía el diseño de una casa. Una vez

que el diseño está completo, sólo tiene un diseño para un determinado tipo de casa. Sin embargo, todas las personas que quieran pueden comprar el diseño y tener construida la misma casa. Del mismo modo, una clase es un diseño para un determinado conjunto de funcionalidad, y un objeto creado tomando como base una determinada clase tiene toda la funcionalidad de esa clase a partir de la que se ha construido.

Instanciación

Un término único de la programación orientada a objetos, la *instanciación* o *ejemplificación*, es simplemente el hecho de crear una instancia o ejemplar de una clase. Esta instancia es un objeto. En el siguiente ejemplo, todo lo que vamos a hacer es crear una clase, o especificación, para un objeto. En otras palabras, no se ha asignado ninguna memoria de momento porque sólo tenemos el diseño de un objeto, no un objeto en sí mismo.

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        //Calcular el pago aquí.
        return (payRate * (double)hoursWorked);
    }
}
```

Para instanciar esta clase y utilizarla, tenemos que declarar una instancia de esa clase de una manera parecida a esta:

```
public static void Main()
{
    Employee emp = new Employee ("Amy", "Anderson", 28, 100);
}
```

En este ejemplo, *emp* se declara como tipo *Employee* y es instanciado utilizando el operador *new*. La variable *emp* representa una *instancia de la clase Employee* y se considera un objeto *Employee*. Después de la instanciación, podemos comunicarnos con este

objeto a través de sus miembros públicos. Por ejemplo, podemos invocar el método *CalculatePay* del objeto *emp*. No podemos hacerlo si no tenemos un objeto real. (Hay una excepción a esto y es cuando tratamos con métodos estáticos. Trataremos los métodos estáticos en los Capítulos 5 y 6, «Métodos»).

Eche un vistazo al siguiente código C#:

```
public static void Main()
{
    Employee emp = new Employee();
    Employee emp2 = new Employee();
}
```

Aquí tenemos dos instancias —*emp* y *emp2*— de la misma clase *Employee*. Mientras que por programa cada objeto tiene las mismas capacidades, cada instancia contendrá sus propios datos de instancia y se tratará por separado. Del mismo modo podemos crear un array completo o colección de esos objetos *Employee*. El Capítulo 7, «Propiedades, arrays e indizadores», tratará los arrays con detalle. Sin embargo, el punto que queremos resaltar aquí es que la mayoría de los lenguajes orientados a objetos poseen la capacidad de definir un array de objetos. Esto, a su vez, proporciona la capacidad de agrupar objetos fácilmente y de moverse por ellos invocando métodos del array de objetos o suscribiéndose al array. Compare esto con el trabajo que habría tenido que hacer con una lista enlazada, en la que habría tenido que enlazar manualmente cada elemento de la lista con su elemento anterior y posterior.

LOS TRES PRINCIPIOS DE LOS LENGUAJES ORIENTADOS A OBJETOS

Según Bjarne Stroustrup, autor del lenguaje de programación C++, para que un lenguaje se llame a sí mismo orientado a objetos debe soportar tres conceptos: objetos, clases y herencia. Sin embargo, ha llegado a pensarse más comúnmente que los lenguajes orientados a objetos son lenguajes contruidos sobre el trípode *encapsulación*, *herencia* y *polimorfismo*. La razón de este cambio de filosofía es que con el paso de los años hemos llegado a darnos cuenta de que la encapsulación y el polimorfismo son partes tan integrantes de la construcción de sistemas orientados a objetos como la clase y la herencia.

Encapsulación

Como ya mencionamos anteriormente, la encapsulación, a veces llamada *ocultación de la información*, es la capacidad de ocultar los procesos internos de un objeto a sus usuarios y proporcionar una interfaz sólo para los miembros que quiera que el cliente tenga posibilidad de manipular directamente. Sin embargo, también hablamos de abstracción en el mismo contexto, por eso en esta sección aclararemos cualquier confusión relacionada con estos dos conceptos tan parecidos. La encapsulación proporciona el vínculo entre la interfaz externa de una clase —esto es, los miembros públicos visibles a los

usuarios de la clase— y sus detalles de implementación interna. La ventaja de la encapsulación para el desarrollador de la clase es que puede exponer los miembros de una clase que quedarán estáticos, o inamovibles, mientras se ocultan los procesos internos más volátiles y dinámicos. Como ya vio anteriormente en este capítulo, la encapsulación se consigue en C# gracias a la asignación de un modificador de acceso —*public*, *private* o *protected*— a cada miembro de una clase.

Cómo diseñar abstracciones

Una abstracción se refiere a cómo se representan determinados problemas en el espacio del programa. Los lenguajes de programación en sí mismos proporcionan abstracciones. Piense en ello de esta manera: ¿Cuándo fue la última vez que tuvo que preocuparse de los registros y las pilas de la UCP (Unidad Central de Proceso)? Incluso si al principio aprendió a programar en ensamblador, apostamos que ha pasado mucho tiempo desde que tuvo que preocuparse sobre ese tipo de detalles de tan bajo nivel y específicos de la máquina. La razón es que la mayoría de los lenguajes de programación le abstraen de esos detalles de tal manera que puede centrarse en el dominio del problema.

Los lenguajes orientados a objetos le dan la posibilidad de declarar clases cuyos nombres e interfaces imitan muy de cerca las entidades del dominio del problema del mundo real de tal manera que utilizar los objetos supone un «sentimiento» más natural hacia ellos. El resultado de eliminar los elementos no relacionados directamente con la resolución del problema dado es que usted será capaz de centrarse específicamente en el problema y en una mayor productividad. De hecho, parafraseando a Bruce Eckel en *Thinking in Java* (Prentice Hall Computer Books, 2000), la habilidad para resolver la mayoría de los problemas dependerá generalmente de la calidad de la abstracción que se esté utilizando.

Así, ese es un nivel de abstracción. Si lo lleva un poco más allá, necesita pensar, como desarrollador de clases, en términos de cómo puede diseñar mejores abstracciones para los clientes de la clase de forma que permitan al cliente centrarse en su tarea actual y no atascarse con los detalles de cómo trabaja nuestra clase. En este momento vendría bien esta pregunta: «¿Cómo se relaciona la interfaz de una clase con su abstracción?». La interfaz de la clase es la implementación de la abstracción.

Utilizaré una analogía familiar, que viene de los cursos de programación, para ayudar a concretar estos conceptos: los procedimientos internos de las máquinas expendedoras. Los procedimientos internos de una máquina expendedora están realmente bastante interconectados. Para desempeñar su trabajo, la máquina tiene que aceptar efectivo y un determinado sistema monetario, devolver cambio y dispensar el producto elegido. Sin embargo, la máquina expendedora tiene un grupo finito de funciones que necesita expresar a sus usuarios. Esta interfaz se expresa a través de una ranura para monedas, botones para seleccionar el producto deseado, una palanca para reclamar el cambio, una ranura que devuelva el cambio y un conducto inclinado para dispensar el producto seleccionado. Cada una de estas partes representa una parte de la interfaz de la máquina. Las máquinas dispensadoras, por lo general, no han cambiado mucho desde su invención. Esto es porque a pesar del hecho de que los procedimientos internos han ido cambiando según

evolucionaba la tecnología, la interfaz básica no ha necesitado cambiar mucho. Una parte integral al diseño de una interfaz de clase es tener un conocimiento lo suficientemente profundo del dominio del problema. Este conocimiento le ayudará a crear una interfaz que dé al usuario acceso a la información y métodos a la vez que le aísla de los procedimientos internos de la clase. Necesita diseñar una interfaz no sólo para resolver problemas de hoy, sino también para abstraer lo suficiente los aspectos internos de la clase de forma que los miembros privados de ésta puedan someterse a cambios ilimitados sin que afecte al código existente.

Otro aspecto igualmente importante del diseño de la abstracción de una clase es tener en mente en todo momento al programador del sistema o aplicación cliente. Imagine que está escribiendo un motor de base de datos genérica. Si es usted un gurú de las bases de datos, estaría perfectamente habituado a tratar con términos como *cursores*, *control de confirmación* (*commit*) y *tuplas*. Sin embargo, la mayoría de los desarrolladores que no han programado mucho para bases de datos no van a estar tan familiarizados con estos términos. Utilizando estos términos que son extraños a los clientes de su clase, va a burlar el propósito entero de la abstracción —aumentar la productividad del programador representando el dominio del problema en términos naturales.

Otro ejemplo de cuándo debemos pensar en el cliente es cuando determinamos qué miembros de la clase deben ser accesibles públicamente. De nuevo, algo de conocimiento del dominio del problema y de los clientes de su clase lo obviaría. En nuestro ejemplo del motor de datos, probablemente no querría que sus clientes pudieran acceder directamente a los miembros que representan los búfers de datos internos. Cómo se definen estos búferes de datos podría fácilmente cambiar en el futuro. Además, como estos búferes son críticos para toda operación de su máquina, querría asegurarse de que se modifica sólo a través de sus métodos. Así se puede asegurar que se toman todas las precauciones necesarias.

NOTA Podría pensar que los sistemas orientados a objetos se diseñan básicamente para que sea más fácil crear clases. Aunque esta característica proporciona beneficios de productividad a corto plazo, a largo plazo los beneficios sólo se consiguen después de darse cuenta de que la POO existe para hacer la programación más fácil para los clientes de la clase. Tenga siempre en consideración al programador, que es quien va a instanciar o derivar de las clases que usted crea cuando diseña sus clases.

Los beneficios de una buena abstracción

El diseño de la abstracción de sus clases de la manera más útil para que los programadores las usen es de suprema importancia al desarrollar software reutilizable. Si puede desarrollar una interfaz estable y estática que persista a través de los cambios de la implementación, necesitará modificar menos su aplicación a lo largo del tiempo. Por ejemplo, piense en nuestro anterior código de ejemplo de nóminas. En el caso de un

objeto *Employee* y de la funcionalidad de la nómina, sólo van a ser relevantes algunos métodos, como *CalculatePay*, *GetAddress* y *GetEmployeeType*. Si conoce el dominio del problema de una aplicación de nóminas, puede determinar fácilmente, en un grado bastante alto, los métodos que van a necesitar los usuarios de esta clase. Habiendo dicho esto, si combina el conocimiento íntimo del dominio del problema con la previsión y la planificación del diseño de esta clase, puede estar razonablemente seguro de que la mayor parte de su interfaz para esta clase quedará invariable, a pesar de los cambios futuros en la implementación real de la clase. Después de todo, desde la perspectiva de usuario, es sólo una clase *Employee*. Desde la posición ventajosa del usuario, no debería cambiar casi nada de versión a versión.

El desacoplamiento de los detalles de usuario y de implementación es lo que hace a un sistema completo más fácil de entender y por lo tanto más fácil de mantener. Contraste esto con lenguajes procedimentales como C, en el que cada módulo necesita explícitamente nombrar y acceder a los miembros de una determinada estructura. En ese caso, cada vez que los miembros de la estructura cambian, cada una de las líneas de código que se refieren a la estructura también debe cambiar.

Herencia

La herencia tiene relación con la habilidad del programador para especificar que una clase tiene una relación *especie de* con otra clase. A través de la herencia, puede crear (o derivar) una nueva clase que esté basada en una clase ya existente. Entonces puede modificar la clase de la manera que quiera y crear objetos nuevos del tipo derivado. Esta habilidad es la esencia de la creación de una jerarquía de clases. Fuera de la abstracción, la herencia es la parte más significativa del diseño global del sistema. Una *clase derivada* es la nueva clase que se está creando y la *clase base* es desde la que se deriva la nueva clase. La clase nueva derivada hereda todos los miembros de la clase base, para así posibilitarle que reutilice el trabajo anterior.

NOTA En C#, el hecho de que los miembros de la clase base sean heredados se controla mediante los modificadores de acceso utilizados para definir el miembro. Entraré a este nivel de detalle en el Capítulo 5. Para los propósitos que estamos tratando, puede asumir que una clase derivada heredará todos los miembros de la clase base.

Como ejemplo de cómo y cuándo utilizar la herencia, vamos a echar un vistazo de nuevo a nuestro ejemplo *EmployeeApp*. En este ejemplo, tendríamos casi con certeza diferentes tipos de empleados, como asalariados, por contrato y por horas. Mientras todos estos objetos *Employee* tendrían una interfaz similar, en muchos casos funcionarían internamente de manera distinta. Por ejemplo, el método *CalculatePay* funcionaría de manera diferente para un empleado asalariado que para uno por contrato. Sin embargo, usted quiere la misma interfaz *CalculatePay* para sus usuarios sin tener en cuenta el tipo de empleado.

Si es usted nuevo en la programación orientada a objeto, debe estar preguntándose: «¿Por qué necesito objetos incluso aquí? ¿Por qué no puedo simplemente tener una estructura *EMPLOYEE* con un miembro del tipo empleado y obtener así una función similar a esta?»:

```
Double CalculatePay(EMPLOYEE* pEmployee,int iHoursWorked)
{
    //Validar el puntero a pEmployee.
    if (pEmployee->type == SALARIED)
    {
        //Hacer el proceso W-2 del empleado.
    }
    else if (pEmployee->type == CONTRACTOR)
    {
        //Hacer el proceso 1099.
    }
    else if (pEmployee->type == HOURLY)
    {
        //Hacer el proceso por horas.
    }
    else
    {
        //Hacer el proceso empresa-a-empresa.
    }
    //Devolver el valor de una de las
    //sentencias compuestas anteriores.
}
```

Este código tiene un par de problemas. Primero, que el éxito de la función *CalculatePay* está estrechamente relacionado con la estructura *EMPLOYEE*. Como ya mencionamos anteriormente, un acoplamiento tan fuerte como éste es un problema porque cualquier modificación de la estructura *EMPLOYEE* romperá este código. Como programador de orientación a objetos, la última cosa que querrá hacer es cargar a los usuarios de su clase con la necesidad de conocer los intrincados detalles del diseño de ésta. Sería como un fabricante de máquinas expendedoras que le pide que entienda los mecanismos internos de la máquina expendedora antes de que compre un refresco.

En segundo lugar, el código no fomenta la reutilización. Una vez que empiece a ver cómo la herencia fomenta la reutilización, se dará cuenta de que las clases y los objetos son algo bueno. En este caso, simplemente definiría todos los miembros para la clase base, que funcionaría igual sin tener en cuenta el tipo de empleado. Cualquier clase derivada heredaría esta funcionalidad y después cambiaría lo que fuera necesario. Así es como se vería en C#:

```
class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
}
```

```

        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public double CalculatePay(int hoursWorked)
    {
        //Calcular el pago aquí.
        return (payRate * (double)hoursWorked);
    }
}

class SalariedEmployee : Employee
{
    public string SocialSecurityNumber;

    public void CalculatePay (int hoursWorked)
    {
        //Calcular el pago para un empleado W-2.
    }
}

class ContractEmployee : Employee
{
    public string FederalTaxId;

    public void CalculatePay (int hoursWorked)
    {
        //Calcular el pago para un empleado por contrato.
    }
}

```

Merece la pena resaltar tres características del ejemplo anterior:

- La clase base, *Employee*, define una cadena llamada *EmployeeId*, que es heredada tanto por la clase *SalariedEmployee* como por la clase *ContractEmployee*. Las dos clases derivadas no hacen nada para obtener este miembro —lo heredan automáticamente como resultado de ser derivadas de la clase *Employee*.
- Las dos clases derivadas implementan sus propias versiones de *CalculatePay*. Sin embargo, se dará cuenta de que las dos heredan la interfaz, y aunque cambian los procedimientos para cubrir sus necesidades específicas, el código del usuario se queda igual.
- Las dos clases derivadas añaden nuevos miembros que ya se heredaban de la clase base. La clase *SalariedEmployee* define una cadena *SocialSecurityNumber* y la clase *ContractEmployee* incluye una definición para el miembro *FederalTaxId*.

Ha visto en este pequeño ejemplo que la herencia le permite reutilizar código heredando funcionalidad de las clases base. E incluso va más lejos, permitiéndole extender la clase base u otras antecesoras añadiendo sus propios métodos y variables.

Cómo definir la herencia adecuada

Para referirnos al importante asunto de la herencia usaremos un término del libro de Marshall Cline y Greg Lomow *C++ FAQs* (Addison-Wesley, 1998): *sustitutibilidad*. La substitutibilidad significa que el comportamiento anunciado de la clase derivada es sustituible por el de la clase base. Piense en esta afirmación por un momento; es la regla sencilla más importante relacionada con la creación de jerarquías de las clases que funcionan que aprenderá. (Por «funcionar» queremos decir que soportan la prueba de tiempo y de entrega en las promesas de la POO referidas al código reutilizable y extensible).

Otra regla a seguir que hay que tener en cuenta cuando está creando su jerarquía de clases es que *una clase derivada debería requerir no más y prometer no menos que su clase base en cualquier interfaz heredada*. No adherirse a esta regla inutiliza el código existente. La interfaz de una clase es un contrato obligatorio entre ella misma y los programadores que utilizan la clase. Cuando un programador tiene una referencia a una clase derivada, siempre puede tratar esa clase como si fuera una instancia de la clase base. A esto se le llama *conversión de tipo al de la clase base* (*upcasting*). En nuestro ejemplo, si un cliente tiene una referencia a un objeto *ContractEmployee*, también tiene una referencia implícita a la base de ese objeto, un objeto *Employee*. Por lo tanto, por definición, *ContractEmployee* siempre debería ser capaz de funcionar como su clase base. Observe que esta regla se aplica sólo a la funcionalidad de la clase base. Una clase derivada puede elegir añadir tanto comportamiento propio como quiera. Este comportamiento será más restrictivo en cuanto a requerimientos y promesas que el comportamiento de la clase base. Por esta razón, esta regla se aplica sólo a miembros heredados, porque el código existente tendrá un contrato sólo con esos miembros.

Polimorfismo

La mejor y más concisa definición de polimorfismo que he oído lo define como funcionalidad que permite al código antiguo invocar nuevo código. Este es probablemente el mayor beneficio de la programación orientada a objetos, porque le permiten extender o mejorar su sistema sin romper o modificar el código existente.

Digamos que usted escribe un método que necesita iterar a través de una colección de objetos *Employee*, invocando el método *CalculatePay* de cada objeto. Eso funciona bien cuando su compañía tiene sólo un tipo de empleado, porque puede entonces insertar el tipo de objeto exacto en la colección. Sin embargo, ¿qué pasa cuando empieza a contratar otros tipos de empleados? Por ejemplo, si tiene una clase llamada *Employee* y ésta implementa la funcionalidad de un empleado asalariado, ¿qué hace cuando empieza a contratar empleados por contrato cuyos salarios tienen que ser calculados de manera diferente? Bien, en un lenguaje procedimental, modificaría la función para manejar el nuevo tipo de empleado, ya que el código antiguo posiblemente no pueda saber cómo manejar el nuevo código. Una solución orientada a objetos maneja las diferencias como ésta a través del polimorfismo.

Usando nuestro ejemplo, usted definiría una clase base llamada *Employee*. A continuación, definiría una clase derivada para cada tipo de empleado (como hemos visto

anteriormente). Cada clase de empleado derivada debería así tener su propia implementación del método *CalculatePay*. Aquí comienza la magia. Con el polimorfismo, cuando tenga un puntero a un objeto convertido al tipo de la clase base e invoque a ese método del objeto, el entorno de ejecución del lenguaje asegurará que se llame a la versión correcta del método. Aquí está el código para ilustrar lo que estamos diciendo:

```
using System;

class Employee
{
    public Employee(string firstName, string lastName,
                    int age, double payRate)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.payRate = payRate;
    }

    protected string firstName;
    protected string lastName;
    protected int age;
    protected double payRate;

    public virtual double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("Employee.CalculatePay");
        return 42; //Valor imaginario
    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee(string firstName, string lastName,
                            int age, double payRate)
        : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("SalariedEmployee.CalculatePay");
        return 42; //Valor imaginario
    }
}

class ContractorEmployee : Employee
{
    public ContractorEmployee(string firstName, string lastName,
                              int age, double payRate)
        : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("ContractorEmployee.CalculatePay");
    }
}
```

```

        return 42; //Valor imaginario
    }
}

class HourlyEmployee : Employee
{
    public HourlyEmployee(string firstName, string lastName,
                           int age, double payRate)
        : base(firstName, lastName, age, payRate)
    {}

    public override double CalculatePay(int hoursWorked)
    {
        Console.WriteLine("HourlyEmployee.CalculatePay");
        return 42; //Valor imaginario
    }
}

class PolyApp
{
    protected Employee[] employees;

    protected void LoadEmployees()
    {
        Console.WriteLine("Cargando empleados...");
        //En una aplicación real, probablemente leeríamos esto
        //de una base de datos.
        employees = new Employee[3];

        employees[0] = new SalariedEmployee ("Amy", "Anderson", 28, 100);
        employees[1] = new ContractorEmployee ("John", "Maffei", 35, 110);
        employees[2] = new HourlyEmployee ("Lani", "Ota", 2000, 5);

        Console.WriteLine("\n");
    }

    protected void CalculatePay()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay(40);
        }
    }

    public static void Main()
    {
        PolyApp app = new PolyApp();

        app.LoadEmployees();
        app.CalculatePay();
    }
}

```

Al compilar y ejecutar esta aplicación obtendremos los siguientes resultados:

```
c:\>PolyApp
```

Cargando empleados...

```
SalariedEmployee.CalculatePay
ContractorEmployee.CalculatePay
HourlyEmployee.CalculatePay
```

Observe que el polimorfismo proporciona al menos dos beneficios. Primero le da la capacidad de agrupar objetos que tienen una clase base común y tratarlos consistentemente. En el ejemplo anterior, aunque técnicamente tenemos tres tipos de objetos diferentes —*SalariedEmployee*, *ContractorEmployee* y *HourlyEmployee*—, podemos tratarlos como objetos *Employee* porque todos derivan de la base clase *Employee*. Así es como podemos agruparlos en un array que se define como un array de objetos *Employee*. A causa del polimorfismo, cuando llamamos a uno de estos métodos de objeto, el entorno de ejecución asegurará que se llame al método correcto del objeto derivado.

La segunda ventaja es la que hemos mencionado al principio de esta sección: el código antiguo puede utilizar código nuevo. Observe que el método *PolyApp.CalculatePay* itera a través de su miembro array de objetos *Employee*. Como este método extrae implícitamente los objetos como objetos convertidos al tipo de la clase base *Employee* y la implementación del polimorfismo del entorno de ejecución asegura que se invoca al método de clase derivada correcto, podemos añadir otros tipos derivados de *Employee* al sistema, insertándolos en el array de objetos *Employee*, y todo nuestro código sigue funcionando sin que tengamos que cambiar nada del código original!

RESUMEN

Este capítulo le ha llevado como un torbellino por un viaje a través de la terminología y conceptos que se resguardan bajo el paraguas de la programación orientada a objetos. Un discurso completo sobre la materia requeriría varios capítulos, y por lo tanto le quitaría valor al objetivo de este libro. Sin embargo, es imprescindible una comprensión sólida de los fundamentos de la orientación a objetos para sacar el máximo partido del lenguaje C#.

Hemos cubierto bastantes ideas en este capítulo. El conocer la diferencia entre clases, objetos e interfaces y cómo se relacionan estos conceptos con soluciones efectivas es fundamental para entender los sistemas orientados a objetos. Las buenas soluciones orientadas a objetos también dependen de una sólida implementación de los tres principios básicos de la programación orientada a objetos: encapsulación, herencia y polimorfismo. Los conceptos presentados en este capítulo establecen las bases para el siguiente capítulo, que introduce el Framework Microsoft.Net y el Common Language Runtime.

Capítulo 2

Introducción a Microsoft .NET

Sin una comprensión sólida de .NET y del papel que juega el lenguaje C# en esta iniciativa de Microsoft, no entenderá del todo algunos de los elementos centrales de C# que se supone se apoyan en el entorno de ejecución .NET. La visión general de .NET presentada en este capítulo le ayudará a entender no sólo la terminología usada en todo este libro, sino también por qué ciertas características del lenguaje C# funcionan de la manera en que lo hacen.

Si lee algún grupo de noticias o lista de correo sobre .NET, puede ver algunos usuarios que se confunden con la terminología tecnológica. Con los nombres ambiguos y a veces contradictorios que se están barajando, está siendo difícil seguir la pista a los actores. Obviamente, parte del problema es que todo esto es muy nuevo. Lo primero que nos gustaría hacer es explicar algo de terminología relacionada con .NET.

LA PLATAFORMA .NET DE MICROSOFT

La idea que está detrás de Microsoft .NET es que .NET desplaza el foco de la informática desde un mundo en el que los dispositivos individuales y los sitios Web están simplemente conectados a través de Internet a uno en el que los dispositivos, recursos y ordenadores trabajan juntos para proporcionar soluciones más sustanciosas a los usuarios. La solución Microsoft .NET está formada por cuatro componentes centrales:

- .NET Building Block Services, o acceso por programa a ciertos servicios, como almacenamiento de archivos, calendario y Passport.NET (un servicio de comprobación de identidad).
- Software de dispositivo .NET, que se ejecutará en los nuevos dispositivos para Internet.
- La experiencia de usuario .Net, que incluye características como la interfaz natural, agentes de información y etiquetas inteligentes, una tecnología que automatiza los hipervínculos con la información relacionada con palabras y frases en documentos creados por el usuario.

- La infraestructura .NET, que comprende el Framework .NET, Microsoft Visual Studio.NET, .NET Enterprise Servers y Microsoft Windows.Net

La infraestructura .NET es la parte de .NET a la que la mayoría de los desarrolladores se refieren cuando hacen referencia a .NET. Puede asumirse que en cualquier momento que nos refiramos a .NET (sin que le anteceda un adjetivo) estaremos hablando de la infraestructura de .NET. La infraestructura .NET se refiere a todas las tecnologías que conforman el nuevo entorno para crear y ejecutar aplicaciones robustas, escalables y distribuidas. La parte de .NET que nos permite desarrollar estas aplicaciones es el Framework .NET.

El Framework .NET consiste en el Common Language Runtime (CLR) y en las bibliotecas de clases del Framework .NET, a veces llamadas Biblioteca de Clases Base (BCL). Piense en el CLR como en la máquina virtual en la que funcionan las aplicaciones .NET. Todos los lenguajes .NET tienen las bibliotecas de clases del Framework .NET a su disposición. Si está familiarizado con Microsoft Foundation Classes (MFC) o con Borland's Object Windows Library (OWL), también está familiarizado con las bibliotecas de clases. Las bibliotecas de clases Framework .NET incluyen soporte para todo, desde entrada/salida a archivos y a base de datos hasta XML y SOAP. De hecho, las bibliotecas de clases Framework .NET son tan vastas que fácilmente ocuparía todo un libro el dar una visión general superficial a todas las clases soportadas.

A modo de anotación en los márgenes (y como reconocimiento por mi edad), cuando usamos el término «máquina virtual», no nos referimos a Java Virtual Machine (JVM). Realmente estamos utilizando la definición tradicional del término. Hace varias décadas, antes de que Java no fuera más que otra palabra para una bebida oscura y caliente, IBM fue el primero en acuñar el término «máquina virtual». Una máquina virtual era una abstracción de sistema operativo de alto nivel sobre el que otros sistemas operativos podrían funcionar en un entorno completamente encapsulado. Cuando nos referimos al CLR como un tipo de máquina virtual, nos referimos al hecho de que el código que se ejecuta en el CLR se ejecuta en un entorno encapsulado y gestionado, separadamente de otros procesos de la máquina.

EL FRAMEWORK .NET

Vamos a echar un vistazo a lo que es el Framework .NET y qué proporciona. Lo primero que haremos será comparar .Net con un entorno de desarrollo de aplicación distribuida anterior a él. Después examinaremos una lista de las capacidades que .NET proporciona a los desarrolladores de aplicaciones para crear aplicaciones eficaces distribuidas más rápidamente.

Windows DNA y .NET

¿Sonaba familiar la frase que utilizamos anteriormente para describir .NET —«el nuevo entorno para crear y ejecutar aplicaciones robustas, escalables y distribuidas»? Si es

así, aquí está la razón: .NET es básicamente el descendiente de un intento anterior de satisfacer estas elevadas metas. Esa plataforma se llamó Windows DNA. Sin embargo, .NET es mucho más de lo que Windows DNA quería ser. Windows DNA era una plataforma de soluciones que se centraba en resolver problemas de negocio a través de la utilización de los productos de servidor Microsoft. Se utilizaba a veces el término «pegamento» con Windows DNA, como en «DNA define el pegamento que se utiliza para unir sistemas robustos, escalables y distribuidos». Sin embargo, aparte de ser una especificación técnica, Windows DNA no tenía ninguna pieza tangible. Ésta es sólo una de las mayores diferencias entre Windows DNA y .NET. Microsoft .NET no es sólo un grupo de especificaciones. También incluye algunos productos tangibles, como compiladores, bibliotecas de clases e incluso aplicaciones completas para el usuario final.

El Common Language Runtime

El CLR es el verdadero núcleo de .NET. Como sugiere el nombre, es un entorno de ejecución en el que las aplicaciones escritas en diferentes lenguajes pueden ejecutarse a la vez y llevarse bien; algo también conocido como *interoperabilidad multilenguaje*. ¿Cómo proporciona el CLR este agradable entorno para la interoperabilidad multilenguaje? El Common Language Specification (CLS) es un grupo de reglas a las que un compilador debe adherirse para crear aplicaciones .NET que se ejecuten en el CLR. Cualquiera, incluso usted o yo, que quiera escribir un compilador compatible con .NET, simplemente necesita adherirse a estas reglas, y ¡voilà!, las aplicaciones generadas desde nuestros compiladores se ejecutarán correctamente junto a cualquier otra aplicación .NET y tendrán la misma interoperabilidad.

Un concepto importante relacionado con el CLR es *el código gestionado*. El código gestionado es simplemente código que se ejecuta bajo los auspicios del CLR y que por lo tanto está siendo gestionado por éste. Piense en ello de esta manera: en los entornos de Microsoft Windows de hoy tenemos ejecutándose procesos diferentes. La única regla que se requiere que sigan las aplicaciones es que se comporten correctamente en el entorno Windows. Estas aplicaciones se crean utilizando cualquier compilador de una multitud de compiladores completamente distintos. En otras palabras, las aplicaciones tienen que obedecer sólo las reglas más generales para ejecutarse bajo Windows.

El entorno de Windows tiene unas cuantas reglas globales con relación a cómo se deben comportar las aplicaciones en términos de comunicarse con cualquier otra, de reservar memoria, o incluso registrarse en el sistema operativo Windows para que haga el trabajo por ellas. Sin embargo, en un entorno de código gestionado, una serie de reglas se encargan de asegurar que todas las aplicaciones se comportan de manera globalmente uniforme, sin tener en cuenta el lenguaje en el que están escritas. El comportamiento uniforme de las aplicaciones .NET es la esencia de .NET y no se puede exagerar. Afortunadamente para todos, estas reglas globales afectan principalmente a los escritores de compiladores.

Las bibliotecas de clases del Framework .NET

Las bibliotecas de clases Framework .NET son tremendamente importantes para proporcionar interoperabilidad al lenguaje, porque permiten a los desarrolladores usar una sola interfaz de programación para toda la funcionalidad expuesta por el CLR. Si ha utilizado alguna vez más de un lenguaje distinto en el desarrollo para Windows, le encantará esta característica. De hecho, las bibliotecas de clases del Framework .NET están forjando una tendencia revolucionaria en el desarrollo de los compiladores. Antes de .NET, la mayoría de los escritores de compiladores desarrollaban un lenguaje con la habilidad de hacer la mayor parte de su trabajo. Incluso un lenguaje como C++, que se diseñó como una agrupación reducida de funcionalidad para ser utilizada junto con una biblioteca de clases, tiene al menos algo de funcionalidad por sí mismo. Sin embargo, en el mundo de .NET, los lenguajes se están volviendo poco más que interfaces sintácticas para las bibliotecas de clases de .NET.

Como ejemplo, vamos primero a echar un vistazo a la aplicación estándar en C++ «Hola, mundo» y después la compararemos con una aplicación que hace lo mismo en C#:

```
#include <iostream.h>

int main(int argc, char* argv[])
{
    cout << "¡Hola, mundo!" << endl;
    return 0;
}
```

Observe que la aplicación primero incluye un archivo de cabecera con la declaración de la función *cout*. La función *main* de la aplicación —el punto de entrada a una aplicación C/C++— utiliza la función *cout* para escribir la cadena «Hola, mundo» en el dispositivo de salida estándar. Sin embargo, lo más importante de observar aquí es que no se puede escribir esta aplicación en ningún lenguaje .NET sin las bibliotecas de clase Framework .NET. Esto es así: los lenguajes .NET ni siquiera tienen las características más básicas de compilación, como la habilidad de mostrar una cadena en la consola. Ahora sé que técnicamente la función *cout* se implementa en el entorno de ejecución de C/C++, que es una biblioteca en sí mismo. Sin embargo, las tareas básicas de C++, como el formateo de cadenas, entrada y salida de archivos y entrada y salida de pantalla están, al menos lógicamente, consideradas parte del lenguaje base. Con C# —o cualquier otro lenguaje .NET a este efecto—, el lenguaje en sí mismo casi no tiene capacidad para hacer ni siquiera las tareas más insignificantes sin las bibliotecas de clases Framework .NET.

Echemos un vistazo al ejemplo «Hola, mundo» en C# para ver lo que queremos decir:

```
using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hola, mundo");
    }
}
```

Así que, ¿qué significado tiene para usted este conjunto común de bibliotecas de clase?; ¿es esto bueno? Bien, depende de su posición estratégica. Un conjunto común de bibliotecas de clases significa que todos los lenguajes, teóricamente, tienen las mismas capacidades, porque todos tienen que utilizar estas bibliotecas de clases para todo, excepto para declarar variables.

Una queja que hemos observado en los foros de debate es: «¿Por qué tener múltiples lenguajes, si todos ellos tienen las mismas capacidades?». Cuesta entender esta queja. Los que hemos trabajado en muchos entornos multilinguaje, podemos certificar que no se obtiene gran beneficio al tener que recordar qué lenguaje puede hacer cada cosa con el sistema y cómo lo hace. Después de todo, nuestro trabajo como desarrolladores es producir código, no preocuparnos sobre si uno de nuestros lenguajes favoritos tiene esta o aquella ventaja.

Otra cuestión que hemos visto a menudo es: «Si todos esos lenguajes .NET hacen lo mismo, ¿por qué necesitamos más de uno?». La respuesta se relaciona con el hecho de que los programadores son animales de costumbres. Microsoft no quería elegir un lenguaje de los muchos disponibles y forzar a millones de programadores a echar por tierra sus años de experiencia en otros lenguajes. Un programador no sólo podría familiarizarse con una nueva API; él o ella podría tener que dominar una sintaxis completamente diferente. En cambio, un desarrollador puede continuar utilizando el lenguaje que mejor se ajuste a su trabajo. Después de todo, jugamos a tener productividad. Cambiar lo que no necesita cambiarse, no forma parte de esa ecuación.

NOTA Mientras en teoría las bibliotecas de clases Framework .NET permiten que los compiladores hagan que toda la funcionalidad del CLR esté disponible para los usuarios de un lenguaje, no siempre es así. Un punto de fricción en Microsoft entre el equipo de bibliotecas de clase Framework .NET y los diferentes equipos de compiladores es que, aunque el equipo de bibliotecas de clases del Framework .NET ha intentado exponer toda su funcionalidad a los diferentes lenguajes, no hay nada —aparte de ajustarse a los mínimos estándares del CLS— que requiera que los distintos equipos de compiladores implementen todas y cada una de las características. Cuando se preguntó a varios desarrolladores de Microsoft sobre esta discrepancia, nos dijeron que en lugar de que cada lenguaje tuviera acceso a toda de funcionalidad expuesta del Framework .NET, cada equipo de creación de un compilador ha decidido implementar sólo las características que crean que son más aplicables para sus usuarios. Afortunadamente para nosotros, sin embargo, C# resulta ser el lenguaje que parece haber proporcionado una interfaz a casi toda la funcionalidad del Framework .NET.

Microsoft Intermediate Language y los JITters

Para hacer más fácil a los escritores de lenguajes portar sus lenguajes a .Net, Microsoft desarrolló un lenguaje similar al lenguaje ensamblador llamado Microsoft Intermediate

Language (MSIL). Para compilar aplicaciones .NET, los compiladores toman el código fuente como entrada y producen MSIL como salida. El propio MSIL es un lenguaje completo en el que se pueden escribir aplicaciones. Sin embargo, como sucede con el lenguaje ensamblador, probablemente nunca lo utilizará, salvo en circunstancias excepcionales. Dado que MSIL es propiamente un lenguaje, cada equipo de compiladores toma sus propias decisiones sobre cuánto soportará del MSIL. Sin embargo, si usted es el escritor del compilador y quiere crear un lenguaje que interopere con otros lenguajes, debería limitarse a las características especificadas por el CLS.

Cuando compile una aplicación C# o cualquier aplicación escrita en un lenguaje compatible con el CLS, la aplicación se compila en MSIL. El CLR compila posteriormente este MSIL, al ejecutar por primera vez la aplicación, a instrucciones nativas de la UCP. (Realmente, sólo las funciones a las que se llama se compilan por primera vez cuando son invocadas). Sin embargo, como todos aquí somos adictos a los ordenadores y este libro se titula *A fondo C#*, veamos qué es lo que realmente está pasando internamente:

1. Usted escribe código fuente en C#.
2. Después lo compila usando el compilador C# (csc.exe) en un EXE.
3. El compilador C# genera el código MSIL y un manifiesto en una parte de sólo lectura del EXE que tiene una cabecera PE (Win 32-portable executable) estándar.

Hasta ahora, bien. Sin embargo, aquí está la parte importante: cuando el compilador crea la salida, también importa una función del entorno de ejecución .NET, llamada *_CorExeMain*.

4. Cuando se ejecuta la aplicación, el sistema operativo carga el PE, al igual que cualquier biblioteca de vínculos dinámicos de la que dependa (DLL), como la que exporta la función *_CorExeMain* (mscorlib.dll), igual que hace con cualquier PE válido.
5. Después, el cargador del sistema operativo salta al punto de entrada dentro del PE, que ha sido puesto ahí por el compilador C#. De nuevo, así es exactamente cómo se ejecuta cualquier otro PE en Windows.

Sin embargo, como el sistema operativo obviamente no puede ejecutar el código MSIL, el punto de entrada es simplemente un pequeño stub que salta a la función *_CorExeMain* en mscorlib.dll.

6. La función *_CorExeMain* comienza la ejecución del código MSIL que se colocó en el PE.
7. Como el código MSIL no puede ejecutarse directamente —porque no está en formato ejecutable por la máquina—, el CLR compila el MSIL utilizando un compilador just-in-time (JIT o JITter) en código nativo de la UCP según procese el MSIL. La compilación JIT sólo ocurre a medida que invocan métodos en el programa. El código ejecutable compilado está en la caché de la máquina y es recompilado sólo si hay algún cambio en el código fuente.

Se pueden utilizar tres diferentes JITters para convertir el MSIL en código nativo, dependiendo de las circunstancias:

- **Generación de código en tiempo de instalación.** La generación de código en tiempo de instalación compilará un ensamblaje (*assembly*) entero en código binario específico de la UCP, como hace un compilador C++. Un ensamblaje es el paquete de código que se envía al compilador (hablaremos con más detalle de los ensamblajes más adelante en este capítulo, en «Despliegue»). Esta compilación se hace en tiempo de instalación, cuando es menos probable que el usuario final se dé cuenta de que el ensamblaje está siendo compilado en ese mismo momento. La ventaja de la generación de código en tiempo de instalación es que permite compilar el ensamblaje entero justo antes de ejecutarlo. Como se compila el ensamblaje entero, no tiene que preocuparse de los problemas intermitentes de rendimiento cada vez que un método se ejecuta por primera vez en su código. Es como un plan de vacaciones de alojamientos en multipropiedad en el que se paga todo por adelantado. Mientras que es doloroso pagar por la multipropiedad para las vacaciones, la ventaja es que nunca tiene que preocuparse de volver a pagar el alojamiento. Cuando utilizarlo, y si merece la pena hacerlo, depende del tamaño de su sistema específico y su entorno de despliegue. Normalmente, si va a crear una aplicación de instalación para su sistema, debería seguir adelante y usar este JITter para que el usuario tenga una versión «independiente» completamente optimizada de la misma.
- **JIT.** El JITter por defecto se invoca en tiempo de ejecución —como describimos en la lista numerada anterior— cada vez que un método se llama por primera vez. Es similar al plan «pague cuando lo necesite», y se ejecuta de forma predeterminada si explícitamente no ejecuta el compilador PreJIT.
- **EconoJit.** Otro entorno de ejecución JITter, el EconoJIT, está diseñado especialmente para sistemas que tienen recursos limitados; por ejemplo, dispositivos portátiles con poca cantidad de memoria. La mayor diferencia entre este JITter y el JITter normal es la incorporación del *descarte de código*. El descarte de código permite al EconoJIT descartar el código generado o compilado si el sistema comienza a quedarse sin memoria. El beneficio es que se pide que se libere la memoria. Sin embargo, la desventaja es que si el código que se está descartando se vuelve a invocar, debe compilarse otra vez, como si nunca se hubiera invocado.

Sistema de tipos unificado

Una de las características básicas de cualquier entorno de desarrollo es su sistema de tipos. Después de todo, un entorno de desarrollo con una cantidad limitada de tipos o un sistema que limite la habilidad del programador para extender los tipos proporcionados por el sistema no es un entorno con una gran esperanza de vida. El entorno de ejecución .NET hace algo más que dar simplemente al desarrollador un sistema de tipos único y unificado que se utilice en todos los lenguajes compatibles con CLS. También permite a los escritores de lenguaje extender el sistema de tipos añadiendo nuevos tipos que se parecen y actúan igual que los tipos propios del sistema. Esto significa que usted, como desarrollador, puede utilizar *todos* los tipos de manera uniforme, sin tener en cuenta si son tipos predefinidos de .NET o tipos creados por el usuario. En el Capítulo 4, «El

sistema de tipos», trataremos los detalles del sistema de tipos y de cómo el compilador C# lo soporta.

Metadatos y reflexión

Como ya mencionamos en la anterior sección, «Microsoft Intermediate Language y los JITters», los compiladores compatibles con CLS toman su código fuente como entrada y generan código MSIL para que el entorno de ejecución lo compile (por medio de los JITters) y lo ejecute. Además de la correspondencia entre el código fuente y las secuencias de instrucciones MSIL, los compiladores compatibles con CLS realizan otra tarea igual de importante: incorporar metadatos en el EXE resultante.

Los *metadatos* son los datos que describen datos. En este contexto, los metadatos son la colección de elementos de programa que constituyen el EXE, como los tipos declarados y los métodos implementados. Puede ser que esto le suene vagamente familiar. Estos metadatos son parecidos a las bibliotecas de tipos (*typelibs*) generadas con componentes del Component Object Model (COM). Los metadatos no son sólo salidas de un compilador .NET sustancialmente más expresivas y completas que las *typelibs* COM a las que estamos acostumbrados, sino que los metadatos también están incluidos en el EXE. De esta manera, no hay posibilidad de perder los metadatos de la aplicación o de tener un par de archivos descabalados.

La razón para usar metadatos es sencilla. Permiten al entorno de ejecución .NET saber en tiempo de ejecución qué tipos reservarán memoria y qué métodos se invocarán. Esto posibilita que el entorno de ejecución se configure adecuadamente para ejecutar más eficientemente la aplicación. El mecanismo mediante el cual se consultan estos metadatos se llama *reflexión*. De hecho, las bibliotecas de clases del Framework .NET proporcionan un conjunto completo de métodos de reflexión que permiten que cualquier aplicación —no sólo el CLR— pueda consultar los metadatos de otra aplicación.

Las herramientas como el Visual Studio.NET utilizan estos métodos de reflexión para implementar características como IntelliSense. Con IntelliSense, mientras escribe un nombre de método, los argumentos de ese método aparecen en una lista en la pantalla. Visual Studio.NET lleva esta funcionalidad incluso más allá, mostrando todos los miembros de un tipo. Trataremos las API de reflexión en el Capítulo 15, «Programación multihilo».

Otra herramienta .NET increíblemente útil que se aprovecha de la reflexión es el Microsoft Framework .NET IL Disassembler (ILDASM). Esta eficaz utilidad analiza los metadatos de la aplicación objetivo y a continuación muestra la información de la aplicación en una jerarquía en forma de árbol. La Figura 2.1 ilustra cómo aparece la aplicación «Hola, mundo» C# en ILDASM.

La ventana en segundo plano de la Figura 2.1 es la ventana principal del IL Disassembler. Al hacer doble clic en el método *Main* en la vista de árbol, aparece la ventana en primer plano que muestra los detalles del método *Main*.

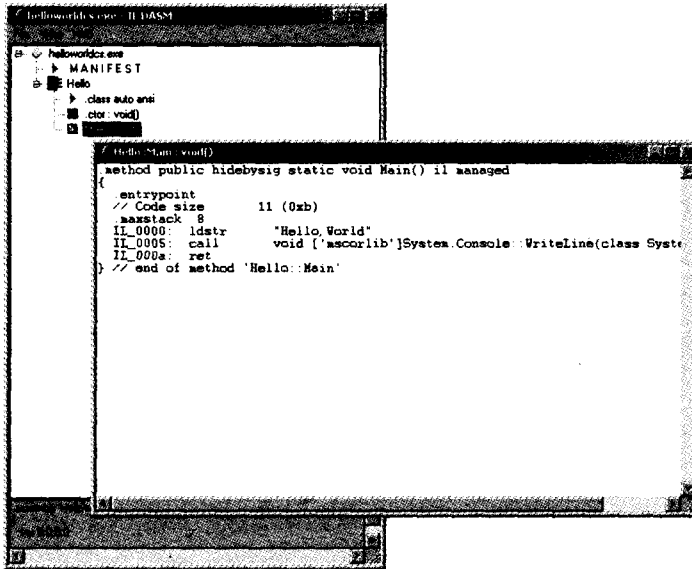


Figura 2.1. La aplicación C# «Hola, mundo» vista en formato ILDASM.

Seguridad

La faceta más importante de cualquier entorno de desarrollo de aplicaciones distribuida es cómo maneja la seguridad. Afortunadamente para aquellos de nosotros que nos hemos quejado durante mucho tiempo de que Microsoft nunca se tomaría en serio en el área de soluciones empresariales de servidor sin una aproximación completamente nueva hacia la seguridad, .NET pone muchos conceptos sobre la mesa. De hecho, la seguridad comienza tan pronto como se carga una clase por el CLR, ya que el cargador de clases es parte del esquema de seguridad de .NET. Por ejemplo, cuando se carga una clase en el entorno de ejecución .NET, se comprueban los factores relacionados con la seguridad tales como reglas de accesibilidad y requisitos de autoconsistencia. Además, los chequeos de seguridad aseguran que un trozo de código tenga las credenciales apropiadas para acceder a ciertos recursos. El código de seguridad asegura la determinación de los roles y la información de identidad. Estos chequeos de seguridad incluso se extienden a procesos y más allá de los límites de la máquina para asegurar que los datos sensibles no están comprometidos en entornos de ejecución distribuidos.

Despliegue

El despliegue es, con mucho, la tarea más horrible asociada al desarrollo de sistemas distribuidos extremadamente grandes. En realidad, como cualquier desarrollador de Windows puede decirle, el tratar con los diferentes archivos binarios, elementos del Registro,

componentes COM e instalación de bibliotecas de soporte de productos tales como Conectividad abierta a bases de datos (ODBC) y Objetos de acceso de datos (DAO) es suficiente para hacerle pensar dos veces en la elección de su carrera. Afortunadamente, el despliegue es una área en la que el equipo de diseño .NET obviamente se ha tomado mucho tiempo.

La clave para el despliegue de la aplicación .NET es el concepto de *ensamblajes*. Los ensamblajes son simplemente paquetes de comportamiento relacionado semánticamente que se construyen bien como entidades de un solo archivo o como entidades de múltiples archivos. Las especificaciones sobre cómo desplegar su aplicación variarán dependiendo de si está desarrollando una aplicación de servidor Web o una aplicación tradicional de escritorio para Windows. Sin embargo, con la introducción del ensamblaje como conjunto de funcionalidad completamente encapsulado, el despliegue puede ser tan simple como copiar los ensamblajes necesarios en una localización de destino.

Muchos de los problemas que causaron tantas molestias a los programadores antes del Framework .NET ya se han eliminado. Por ejemplo, no hay necesidad de registrar los componentes —como hacía con los componentes COM y los controles Microsoft ActiveX—, porque con los metadatos y la reflexión todos los componentes se describen a sí mismos. El entorno de ejecución .NET también lleva el seguimiento de los archivos, y las versiones de éstos, asociados con una aplicación. Por lo tanto, cualquier aplicación que se instale se asocia automáticamente con los archivos que forman parte de su ensamblaje. Si la instalación de una aplicación intenta sobrescribir un archivo necesario para otra aplicación, el entorno de ejecución .NET es lo suficientemente inteligente como para permitir a la aplicación de instalación que instale el archivo requerido, aunque el CLR no elimina la versión anterior de éste, porque todavía es necesaria para la primera aplicación.

Interoperabilidad con código no gestionado

Como cabría sospechar, el código *no gestionado* es código que no está controlado por el entorno de ejecución de .NET. Seamos claros con esto: este código aún se ejecuta en el entorno de ejecución de .NET. Sin embargo, el código no gestionado carece de las ventajas que tiene el código gestionado, como la recolección de basura, un sistema de tipos unificado y metadatos. Podría preguntarse por qué nadie querría ejecutar código no gestionado en el entorno de .NET. Bien, no lo haría si no lo hubiera decidido así. Es más, lo haría cuando se enfrentase con circunstancias que ofrecieran pocas alternativas. He aquí algunas situaciones por las que agradecerá que Microsoft ponga esta característica en .NET:

- **Código gestionado invocando funciones de DLL no gestionada.** Digamos que su aplicación necesita comunicarse con una DLL estilo C y la compañía que escribió la DLL no ha adoptado .NET tan deprisa como lo ha hecho su compañía. En este caso, todavía necesita invocar a esa DLL desde una aplicación .NET. Cubriremos esto con suficientes ejemplos en el Capítulo 16, «Cómo obtener información sobre metadatos con Reflection».

- **Código gestionado que utiliza componentes COM.** Por la misma razón por la que podría necesitar continuar permitiendo invocar a las funciones de DLL estilo C desde su aplicación .NET, también podría necesitar seguir dando soporte para componentes COM. Puede hacer esto creando un adaptador .NET para el componente COM de forma que el cliente gestionado crea que está trabajando con una clase .NET. Esto también se explica en el Capítulo 16.
- **Código no gestionado utilizando servicios .NET.** Esto es precisamente el problema contrario; usted quiere acceder a .NET desde un código no gestionado. Se resuelve utilizando la aproximación recíproca: se engaña al cliente COM para que crea que está utilizando un servidor COM, que realmente es un servicio .NET de algún tipo. También verá ejemplos de esto en el Capítulo 16.

RESUMEN

Microsoft.NET representa el cambio hacia un modelo de programación en el que los dispositivos, servicios y ordenadores trabajan juntos para proporcionar soluciones a los usuarios. Importante en este cambio es el desarrollo del Framework .NET y del CLR, que se muestran en la Figura 2.2. El Framework .NET contiene las bibliotecas de clases compartidas por los lenguajes compilados para ejecutarse en el CLR. Como C# fue diseñado para el CLR, usted no puede llevar a cabo ni siquiera las tareas más simples en C# sin el CLR y las bibliotecas de clases de Framework .Net. Para sacar el máximo partido a C# y al resto de este libro es necesario entender las características de estas tecnologías.

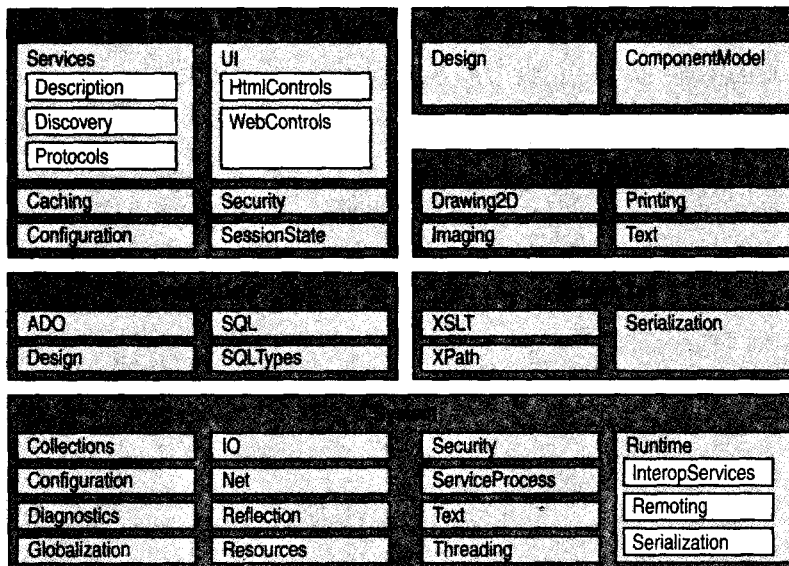


Figura 2.2. El Framework .NET contiene bibliotecas diseñadas para facilitar la interoperabilidad entre servicios, dispositivos y ordenadores.

Capítulo 3

Hola, C#

Antes de que nos metamos de lleno en el corazón de C# —Parte II, «Fundamentos de C#», y Parte III, «Como escribir código»—, consideramos que sería buena idea tener un capítulo de «iniciación». En este capítulo le llevaremos, en un rápido viaje, por el proceso de desarrollo de una aplicación simple con C#. Primero explicaremos las ventajas y desventajas de los diferentes editores que puede utilizar para escribir en C#. Una vez que haya seleccionado un editor, escribiremos la aplicación de ejemplo clásica «Hola, mundo» para conocer la sintaxis básica y la estructura para escribir aplicaciones en C#. Verá que, como en la mayoría de los lenguajes, la sintaxis es formulista y puede utilizar esta aplicación como plantilla para escribir las aplicaciones más básicas de C#. Entonces aprenderá a compilar utilizando el compilador desde la línea de comandos y aprenderá a ejecutar su nueva aplicación.

ESCRIBIENDO SU PRIMERA APLICACIÓN EN C#

Vayamos paso a paso para que pueda crear y ejecutar su primera aplicación C#.

Escoger un editor

Antes de escribir una aplicación C#, necesita escoger un editor. Las siguientes secciones describen los editores más comunes y muestran algunos hechos relevantes a la hora de escoger un editor para desarrollar con C#.

Bloc de notas

El Bloc de notas de Microsoft ha sido el editor más utilizado por los desarrolladores que usan el .NET Framework SDK para escribir las aplicaciones C# durante las primeras etapas del mismo. Se utilizó el Bloc de notas para este libro por algunas otras razones

que señalaré más adelante. Sin embargo, no recomendaría utilizar el Bloc de notas por las razones que se describen a continuación:

- Los archivos C# deberían guardarse con una extensión `.cs`. Sin embargo, en el cuadro de diálogo de la opción Guardar del Bloc de notas, si no tiene cuidado, al intentar nombrar a su archivo como `Test.cs` tendrá como resultado un archivo llamado `test.cs.txt`, a menos que se acuerde de cambiar la opción de la lista Guardar como archivos de tipo a «Todos los archivos».
- El Bloc de notas no dispone de números de línea —un gran inconveniente cuando el compilador informa de un error en una determinada línea.
- El Bloc de notas inserta ocho espacios para un tabulador, lo que significa que escribir algo más allá de «Hola, mundo» puede hacer a las aplicaciones difíciles de leer.
- El Bloc de notas no realiza sangrado automático cuando presiona la tecla `INTRO`. Por lo tanto, tiene que tabular manualmente hasta la columna deseada para introducir una línea de código.

Hay otras razones, pero como puede ver, el Bloc de notas *no es una buena elección* para desarrollar las aplicaciones de C#.

Visual Studio 6

Dado que mi pasado en el desarrollo para Microsoft Windows está estrechamente relacionado con el lenguaje Microsoft Visual C++, Microsoft Visual Studio 6 es el editor que he escogido. Visual Studio es un editor con un montón de características que incluye todas las necesarias para editar y guardar archivos de C#.

Una de las mayores ventajas de utilizar un editor de programación es que se resalta la sintaxis. Sin embargo, como Visual Studio 6 se distribuyó un par de años antes que C# y fue diseñado para desarrollar aplicaciones de Visual C++, necesitará darle algunos retoques para que resalte el código C# correctamente. El primer paso es cambiar la clave del Registro de Visual Studio. Localice la siguiente clave en su Registro utilizando Regedit.exe u otro editor del Registro:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\
Text Editor\Tabs\Language Settings\C/C++\FileExtensions
```

El valor contendrá una cadena como la siguiente:

```
cpp;cxx;c;h;hxx;hpp;inl;thl;tli;rc;rc2
```

Añadir la extensión `.cs` al final del valor. (Observe que es opcional poner un punto y coma final). El nuevo valor de Registro se vería así:

```
cpp;cxx;c;h;hxx;hpp;inl;thl;tli;rc;rc2;cs
```

Ahora, cuando abra un archivo en Visual Studio 6 con una extensión .cs, Visual Studio la reconocerá como un archivo compatible.

A continuación, necesita decirle a Visual Studio qué palabras son palabras reservadas en C#. Para hacer esto, cree y coloque en la misma carpeta en que se encuentre el archivo msdev.exe un archivo que se llame usertype.dat. Este archivo es de texto ASCII que contiene todas las palabras clave que deben ser resaltadas; coloque una palabra reservada en cada línea. Cuando Visual Studio arranque, se cargará este archivo. Por lo tanto, cuando haga un cambio en este archivo, debe rearrancar Visual Studio para ver sus cambios. Hemos incluido una copia de un archivo usertype.dat en el CD que acompaña a este libro, que lista todas las palabras reservadas para el lenguaje C#. La Figura 3.1 muestra cómo quedará su código C# una vez que haya seguido estos pasos.

Visual Studio.NET

Obviamente, si quiere el máximo de productividad en el entorno .NET, debería utilizar Visual Studio.NET. No sólo proporciona todas las herramientas y asistentes integrados para crear aplicaciones C#, sino que también incluye características para mejorar la productividad, como IntelliSense y Dynamic Help. Con IntelliSense, según escribe en un espacio de nombres o nombre de clase, los miembros se muestran automáticamente de manera que no tenga que recordar cada miembro de cada clase. IntelliSense también muestra todos los parámetros y sus tipos cuando teclea el nombre de un método y un paréntesis abierto. Visual Studio 6 también proporciona esta característica, pero obviamente no admite los tipos y las clases .NET. Dynamic Help es una característica nueva en Visual Studio. Mientras está escribiendo código en el editor, una ventana aparte



Figura 3.1. Una ventaja de usar un editor que proporcione resaltado de sintaxis como el que hay en Visual Studio 6 es que rápidamente podemos ver si lo que hemos escrito es una palabra reservada válida o no.

muestra temas relacionados con la palabra en la que está situada el cursor. Por ejemplo, si escribe la palabra reservada *namespace*, la ventana muestra hipervínculos hacia temas de ayuda que tratan del uso de la palabra reservada *namespace*.

Editores de terceros

No olvidemos que también hay un montón de editores de terceros populares por ahí, como Starbase's CodeWright y Visual SlickEdit de MicroEdge. No entraré en detalles sobre esos editores, pero aclararé que puede utilizar cualquier editor de terceros para escribir aplicaciones C#.

Lo que utilizamos para este libro

Como cuando se empezó este libro Visual Studio.NET estaba aún en pruebas, se ha utilizado Visual Studio 6. Esperamos que el haber utilizado un entorno que no fue creado para C# nos haya ayudado a mantener la promesa de que el libro será útil para cualquiera que desarrolle aplicaciones C#, sin tener en cuenta el entorno de desarrollo elegido. Como ya hemos mencionado antes, puede incluso escribir las demos de este libro utilizando el Bloc de notas y obtendrá los resultados esperados.

«Hola, mundo»

Si damos por hecho que se ha decidido por un entorno de desarrollo, veamos su primera aplicación C#, la aplicación clásica «Hola, mundo». Escriba el siguiente código en un archivo y guárdelo con el nombre HelloWorld.cs:

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hola, mundo");
    }
}
```

No nos preocupemos todavía por lo que hace cada línea de código. En este punto sólo nos importa presentar la primera aplicación, compilarla y ejecutarla. Una vez que lo hemos hecho —lo que validará que tenga el entorno correcto para crear y ejecutar aplicaciones C#—, entraremos en detalles sobre lo que hace este código.

Utilizar el compilador mediante línea de comandos

Si está utilizando un editor con una característica incorporada para crear aplicaciones C#, este paso podría ser algo raro. Sin embargo, para mantenernos agnósticos como sea

posible de cara a elegir un editor, vamos a utilizar el compilador de C# de línea de comandos (`csc.exe`) en todo este libro. Esto va a proporcionar dos beneficios. Primero, significa que no importa qué entorno esté utilizando, los pasos para construir las demos siempre van a funcionar. En segundo lugar, aprender las diferentes opciones del compilador le ayudará a largo plazo en situaciones en las que su editor no le proporcione un control total de cara a la compilación.

Una vez que ha escrito en el código del programa «Hola, mundo», abra una ventana de intérprete de comandos y desplácese a la carpeta donde se encuentra el archivo `HelloWorld.cs`. Ahora escriba lo siguiente:

```
csc HelloWorld.cs
```

Si todo funciona correctamente, debería ver resultados parecidos a los que se muestran en la Figura 3.2, donde el nombre y la versión del compilador se muestran junto con cualquier error o aviso. Como puede ver, estamos utilizando una versión beta del compilador en el momento de escribir este libro. A pesar de la versión del compilador que esté utilizado, no debería ver ningún error o aviso como resultado de este sencillo ejemplo.

Si recibiera un error del tipo *'csc' is not recognized as an internal or external command, operable program or batch file*, probablemente significa que no ha instalado el .NET SDK, que incluye el compilador C#.

Si invoca el compilador C# incorrectamente —pasándole un nombre de archivo no válido— o lo llama con el argumento `/?`, el compilador listará todos los argumentos posibles que pueden utilizarse cuando compile sus aplicaciones. No queremos atascarnos con esta charla describiendo todas las opciones que se pueden utilizar con el compilador; por el contrario, nos centraremos en las más importantes, y usted puede investigar el resto en sus ratos libres.



Figura 3.2. La compilación de `HelloWorld.cs` no debería producir errores ni avisos.

Ejecutar la aplicación

Ahora que ha construido la aplicación «Hola, mundo», ejecutémola para asegurarnos de que nuestro entorno de ejecución .NET está instalado correctamente. Todos los ejemplos de este libro serán «aplicaciones de consola», esto es, nos vamos a ajustar a C#; no se va a hacer nada específico para desarrollo en Windows. Como resultado, puede ejecutar estos ejemplos desde la línea de comandos o desde su editor si permite la ejecución de aplicaciones desde la línea de comandos.

Si está utilizando la opción de línea de comandos, abra un intérprete de comandos ahora y escriba **HelloWorld** en el símbolo del sistema donde construyó la aplicación. La salida debería ser parecida a esto:

```
d:\>HelloWorld
Hola, mundo
```

Si, por otra parte, está intentando ejecutar la aplicación desde un editor, y o no ve nada o aparece el símbolo del sistema, ejecuta la aplicación y desaparece rápidamente, altere su código de esta manera:

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hola, Mundo");
        string str = System.Console.ReadLine();
    }
}
```

La invocación de *System.Console.ReadLine* hará que la aplicación se pare hasta que usted haya pulsado la tecla **INTRO**, dándole así la oportunidad de ver que la aplicación ha impreso lo que esperaba. De aquí en adelante, los ejemplos de este libro no contendrán esta línea de código. Si está utilizando un editor para ejecutar estos ejemplos, necesitará acordarse de insertar esta línea antes del final de cada aplicación.

REVISIÓN DEL CÓDIGO

Ahora que hemos establecido que puede escribir y ejecutar una aplicación C#, vamos a revisar el código y a echar un vistazo a la estructura de una aplicación C#.

Programación en un único paso

Como lleva viendo en los primeros capítulos y en la aplicación «Hola, mundo», los ejemplos muestran los métodos de cada clase que se define en la propia definición de la clase. Esto no es un mero ejercicio de conveniencia por nuestra parte, como podrían pensar los programadores de C++. Cuando se programa en C++, tiene dos opciones:

puede implementar una función miembro de una clase directamente en la declaración de la clase —un ejemplo de *programación en línea* (*inline*, se programa una función junto a su definición, típico en C++)—, o puede separar la declaración de la clase y las definiciones de las funciones miembro en diferentes archivos. En C# no puede elegir.

Cuando define una clase en C#, debe definir todos los métodos en línea (los archivos de cabecera no existen). ¿Por qué es bueno esto? Permite a los escritores de clases crear código muy móvil, que es uno de los conceptos clave del entorno .NET. De esta manera, cuando escribe una clase C#, termina con un conjunto de funcionalidad completamente encapsulada que puede portar fácilmente a cualquier otro entorno de desarrollo sin preocuparse cómo incluyen archivos esos procesos del lenguaje o si tiene un mecanismo para incluir archivos dentro de archivos. Utilizando este enfoque, «programación en un solo paso», puede, por ejemplo, tomar una clase entera y depositarla en una Página de servidor activo (*Active Server Pages=ASP*), ¡y funcionará igual que si estuviera compilada como aplicación de escritorio Windows!

Clases y miembros

Lo primero que ve en una aplicación C# básica es el nombre de una clase o el de un espacio de nombres. Como aprendió en el Capítulo 1, «Fundamentos de la programación orientada a objetos», debería seleccionar un nombre de clase que describa el dominio de problema —por ejemplo, *Invoice*, *PurchaseOrder* o *Customer*. El principio y el final de la definición de una clase están marcados por las «llaves»: { y }, respectivamente. Todo lo que está dentro se considera parte de esa clase de C#. Observe que tenemos una clase llamada *HelloWorld* y todo en la aplicación se define en el contexto de esa clase.

Todos los miembros de una clase se definen en las llaves de la clase. Esto incluye métodos, campos, propiedades, indizadores, atributos e interfaces. Entraremos en los detalles específicos de cómo definir estos elementos distintos de C# en los próximos capítulos.

El método *Main*

Toda aplicación C# debe tener un método llamado *Main* definido en una de sus clases. No importa qué clase contenga el método —puede tener tantas clases como quiera en una determinada aplicación— mientras una clase tenga un método llamado *Main*. Además este método debe definirse como *public* y *static*. La palabra reservada *public* es un modificador acceso que le dice al compilador de C# que cualquiera puede invocar este método. Como ya hemos visto en el Capítulo 1, la palabra reservada *static* le dice al compilador que el método *Main* es un método estático global y que la clase no necesita ser instanciada para que se invoque el método. Esto tiene sentido una vez que lo piensa, porque de otra forma el compilador no sabría cómo o cuándo instanciar su clase. Como el método es estático, el compilador almacena la dirección del método como punto de entrada para que el entorno de ejecución .NET sepa dónde empezar la ejecución de su aplicación.

NOTA Los ejemplos de este capítulo muestran que el método *Main* vuelve *void* y que no recibe ningún parámetro. Sin embargo, puede definir su método *Main* de forma que devuelva un valor y también que reciba un array de parámetros. Estas opciones, así como la forma de recorrer los argumentos pasados al método *Main* de una aplicación, se tratarán en el Capítulo 5, «Clases».

El método *System.Console.WriteLine*

El método *System.Console.WriteLine* escribe la cadena indicada —seguida de un fin de línea— al dispositivo de salida estándar. En la mayoría de los casos, a menos que haga algo original para cambiarlo o esté usando un editor que redirija la salida a una ventana, significa que la cadena saldrá en una ventana de consola.

Espacio de nombres y la directiva *using*

En el Capítulo 2, «Introducción a Microsoft .NET», aprendió que la biblioteca de clases de Framework .NET está organizada como un entramado jerárquico de espacios de nombres. Esto puede provocar que algunos nombres sean bastante largos cuando un determinado tipo o clase está ubicado en el cuarto o quinto nivel de profundidad de la jerarquía. Sin embargo, para ahorrar tiempo al escribir, C# proporciona la directiva *using*. Vamos a echar un vistazo a un ejemplo de cómo funciona esta directiva. En nuestra aplicación «Hola, mundo», tenemos la siguiente línea de código:

```
System.Console.WriteLine("Hola, mundo");
```

Escribir esto sólo una vez no es mucho trabajo, pero imagine que tiene que calificar por completo cada tipo o clase en una aplicación grande. La directiva *using* le da la capacidad de indicar una especie de camino de búsqueda para que si el compilador no entiende algo que usted ha escrito, busque en la lista de espacios de nombres la definición. Cuando incorporamos la directiva *using* en nuestro ejemplo, queda así:

```
using System;

class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hola, mundo");
    }
}
```

Cuando el compilador analice el método *Console.WriteLine*, determinará que el método está indefinido. Sin embargo, buscará entonces por los espacios de nombres especificados con los directivos *using* y encontrará el método en el espacio de nombres *System* y compilará el código sin error.

Observe que la directiva *using* se aplica a los espacios de nombres y no a las clases. En el ejemplo que estamos utilizando, *System* es el espacio de nombres, *Console* es la clase y *WriteLine* es un método estático que pertenece a *Console*. Por lo tanto, el siguiente código no sería válido:

```
using System.Console; //ERROR No puede utilizar
                      //la directiva using con una clase.

class HelloWorld
{
    public static void Main()
    {
        WriteLine("Hola, mundo");
    }
}
```

Aunque no puede especificar una clase en una directiva *using*, la siguiente variante de la directiva *using* le permite crear alias para las clases:

using alias = clase

Utilizando esta forma de la directiva *using*, puede escribir código de esta manera:

```
using output = System.Console;

class HelloWorld
{
    public static void Main()
    {
        output.WriteLine("Hola, mundo");
    }
}
```

Esto le da la flexibilidad de aplicar alias con significado a las clases anidadas a varios niveles de profundidad en la jerarquía .NET, haciendo así que su código sea un poco más fácil de escribir y de mantener.

Esqueleto de código

Vamos a ver rápidamente lo que se puede considerar como esqueleto de código para la mayoría de aplicaciones C#, código que ilustra un esquema básico para una aplicación C# sencilla, sin florituras. Puede que quiera escribirlo en un archivo y guardarlo para su uso como plantilla en un futuro. Observe que los paréntesis de ángulo indican dónde es necesario proporcionar información.

```
using <espacio de nombres>
namespace <su espacio de nombres opcional>
class <su clase>
{
    public static void Main()
    {
    }
}
```

AMBIGÜEDAD DE CLASES

En el caso de un tipo que se defina en más de un espacio de nombres referenciado, el compilador generará un error indicando la ambigüedad. Por lo tanto, el siguiente código no se compilará, porque la *clase C* se define en dos espacios de nombres y ambos se hacen referencia con la directiva *using*:

```
using A;
using B;

namespace A
{
    class C
    {
        public static void foo()
        {
            System.Console.WriteLine("A.C.foo");
        }
    }
}

namespace B
{
    class C
    {
        public static void foo()
        {
            System.Console.WriteLine("B.C.foo");
        }
    }
}

class MultiplyDefinedClassesApp
{
    public static void Main()
    {
        C.foo();
    }
}
```

Para evitar este tipo de error, asegúrese de que da a sus clases y métodos nombres únicos y descriptivos.

¡ALGO HA IDO MAL!

Entonces, ¿qué sabemos hasta ahora? Sabemos cómo escribir, compilar y ejecutar aplicaciones de C# y tenemos un esquema básico a partir del cual construir una aplicación

C#. ¿Y qué pasa cuando le pasan cosas malas a buenas aplicaciones C#? Primero, vamos a definir «cosa mala»: cualquier cosa que sucede que no esperaba que ocurriera. Por lo que se refiere a la programación, las cosas malas son de dos tipos: errores en tiempo de compilación y errores en tiempo de ejecución. Echemos un vistazo a un par de ejemplos de cada uno y a cómo se corrigen.

Errores en tiempo de compilación

Cuando un compilador, incluso el compilador C#, no puede interpretar lo que está intentando transmitir, imprimirá un mensaje de error y su aplicación no se construirá. Escriba el siguiente código en un archivo llamado `HelloErrors.cs` y compílelo:

```
using Syste;

class HelloErrors
{
    public static void Main()
    {
        xConsole.WriteLine("Hola, mundo");
        Console.WriteLinex("Hola, mundo");
    }
}
```

Al ejecutar el compilador producirá esta salida:

```
HelloErrors.cs(1,7): error CS0234: The type or namespace name
'Syste' does not exist in the class or namespace''
```

Si tenemos en cuenta que siempre hay espacio de nombres predeterminado, significa que el compilador no podría localizar nada llamado `Syste`, por razones obvias. Sin embargo, lo que queremos ilustrar aquí es qué podemos esperar cuando el compilador encuentra errores de sintaxis en el código. Primero verá el nombre del archivo actual que está siendo compilado, seguido por el número de línea y posición de columna del error. Lo siguiente que verá será el código error según lo define el compilador —en este caso, `CS0234`.

Finalmente, después del código error, verá una breve descripción del error. Muchas veces, esta descripción le dará la información suficiente para aclararlo. Sin embargo, si no es así, puede buscar el código de error en la documentación del .NET Framework SDK, para una descripción más detallada. La ayuda en pantalla asociada con el código de error `CS0234` se muestra en la Figura 3.3.

Observe que aunque presentamos tres errores —el espacio de nombre `System` está mal escrito, la clase `Console` está mal escrita y la invocación al método `WriteLine` está mal escrita—, el compilador informará sólo de un error. Esto se debe a que ciertos errores, una vez que se han encontrado, hacen que el compilador anule el proceso de compilación e imprima los errores acumulados hasta ese punto. En este ejemplo, el compilador paró su proceso de compilación una vez que no pudo resolver la directiva `using` porque ese

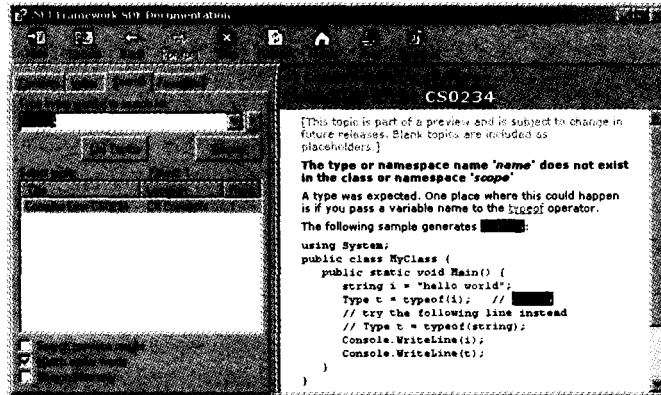


Figura 3.3. Puede utilizar el código de error proporcionado por el compilador para encontrar una descripción del error en la documentación de ayuda en pantalla.

error podría ser la causa de muchos más errores. Una vez que ha escrito correctamente el espacio de nombre *System* en la directiva *using*, el compilador informará de los números de línea y las posiciones de las columnas de los dos errores restantes.

INVESTIGANDO A FONDO ILDASM

Como leyó en el Capítulo 2, cuando crea un EXE o una DLL utilizando un compilador .NET, el archivo no es su archivo ejecutable normal. En su lugar, comprende un manifiesto, que lista los tipos y clases incluidos en el archivo y los códigos de operación MSIL (Microsoft intermediate language) que se compilarán y ejecutarán más tarde mediante la aplicación de instalación o mediante el entorno de ejecución .NET utilizando un compilador just-in-time (JITter).

Una gran ventaja es que el MSIL generado parece lenguaje ensamblador y se puede utilizar como una herramienta increíblemente didáctica para ilustrar lo que el compilador ha hecho con nuestro código. Por esta razón, en este libro muchas veces «bajaremos» hasta la salida MSIL del compilador de C# para ilustrar cómo funciona por dentro algo o para explicar por qué debería utilizar una característica particular del lenguaje de un modo específico. Para ver la salida MSIL que generan los compiladores de .NET, Microsoft ha incluido un desensamblador llamado Microsoft .NET Framework IL Disassembler (ILDASM) para permitirle que abra un archivo ejecutable (EXE o DLL) y examine con detenimiento sus espacios de nombres, clases, tipos y código. Empezaremos a sentirnos cómodos con ILDASM en la siguiente sección.

«Hola, mundo» en MSIL

En el cuadro de diálogo Ejecutar del menú Inicio, escriba **ildasm** y haga clic en Aceptar. Verá una aplicación difícil de describir con unas pocas opciones de menú. En este punto,

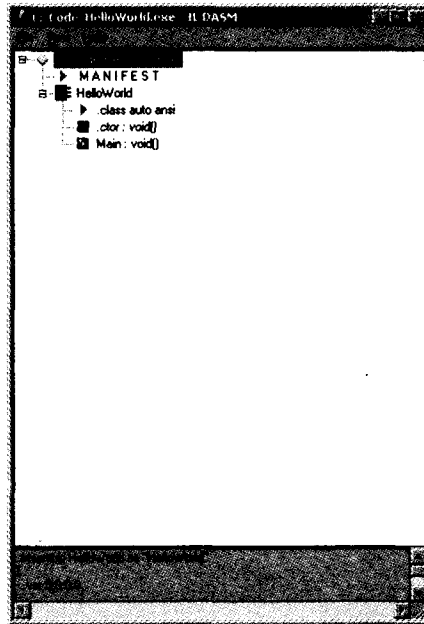


Figura 3.4. ILDASM le deja investigar a fondo el contenido del manifiesto y los códigos de operación IL que componen su aplicación .NET.

desde el menú Archivo, haga clic en Abrir. Cuando aparece el cuadro de diálogo Abrir archivo, desplácese a la carpeta que contiene la aplicación HelloWorld.exe que creó anteriormente (pág. 37) y selecciónela. Como se muestra en la Figura 3.4, las cosas empiezan a parecer algo más prometedoras.

Observe el árbol que utiliza ILDASM para mostrar los contenidos de un binario gestionado. La Figura 3.5 muestra los diferentes iconos utilizados en el árbol que utiliza ILDASM para describir las partes de una aplicación .NET. Como puede ver, si asocia los iconos que se muestran en la Figura 3.5 y el programa en ILDASM «Hola, mundo», HelloWorld.exe consiste en un manifiesto, una clase (*HelloWorld*), dos métodos (un constructor de clase y el método estático *Main*) y algo de información de clase.

La parte más interesante de «Hola, mundo» está en el método *Main*. Haga un doble clic en el método *Main* en el árbol de ILDASM y éste presentará una ventana que muestra el MSIL para el método *Main*, como se muestra en la Figura 3.6.

«Hola, mundo», incluso en MSIL, no es demasiado excitante, pero puede aprender unas cuantas cosas sobre el MSIL generado para trasladarlo a cualquier aplicación .NET. Echemos un vistazo a este método línea por línea para ver lo que queremos decir.

```
.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size      11 (0xb)
    .maxstack 8
}
```

Espacio de nombres:	 (Escudo azul)
Clase:	 (Rectángulo azul con tres salidas)
Interfaz:	 (Rectángulo azul con tres salidas marcado con 'I')
Clase valor:	 (Rectángulo marrón con tres salidas)
Enumerado:	 (Rectángulo marrón con tres salidas marcado con 'E')
Método:	 (Rectángulo magenta)
Método estático:	 (Rectángulo magenta marcado con 'S')
Campo:	 (Diamante cyan)
Campo estático:	 (Diamante cyan marcado con 'S')
Evento:	 (Triángulo verde que apunta hacia abajo)
Propiedad:	 (Triángulo rojo que apunta hacia arriba)
Manifiesto o elementos de información de una clase:	 (Triángulo rojo que apunta a la derecha)

Figura 3.5. Los diferentes iconos utilizados para denotar las partes de una aplicación .NET en ILDASM.

```
IL_0000: ldstr "Hola, mundo"
IL_0005: void [mscorlib]System.Console::WriteLine(string)
IL_000a: ret
} //end of method HelloWorld::Main
```

La primera línea define el método *Main* utilizando la palabra reservada de MSIL *.method*. También podemos ver que el método se define como *public* y *static*, que son los modificadores predeterminados para el método *Main*. Además, sin embargo, vemos

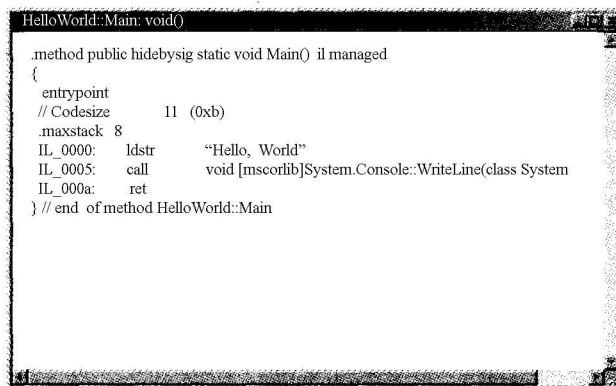


Figura 3.6. Buscar un método en el MSIL generado, abrir el binario en ILDASM y hacer doble clic en el método.

que este método se define como *gestionado*. Esta es una distinción importante porque usted también puede escribir código C# «sin gestionar» o «no seguro». El Capítulo 17, «Cómo interoperar con código no gestionado», discute el código C# no gestionado.

La siguiente línea de código utiliza la palabra reservada *MSIL .entrypoint* para designar este método particular como el punto de entrada a la aplicación. Cuando en entorno de ejecución .NET ejecuta esta aplicación, aquí es donde se pasará el control al programa.

Los siguientes elementos interesantes son los códigos de operación MSIL de las líneas *IL_0000* y *IL_0005*. El primero utiliza el código de operación *ldstr* (Load String) para cargar un literal dado («Hola, mundo») en la pila. La siguiente línea de código invoca al método *System.Console.WriteLine*. Observe que el MSIL prefija el nombre del método con el nombre del ensamblaje que define al método. Lo bonito de este nivel de detalle en el MSIL es que implica que puede escribir más fácilmente herramientas tipo análisis de dependencias que pueden recorrer una aplicación para determinar qué archivos se necesitan para que se ejecute correctamente. Además, puede ver el número de parámetros (y sus tipos) que el método espera. En este caso, el método *System.Console.WriteLine* esperará que un objeto *System.String* esté en la pila cuando se le invoque. Finalmente, la línea *IL_000a* es un sencillo código de operación *ret* de MSIL para salir del método.

ILDASM es una herramienta eficaz. Cuando nos referimos al MSIL generado por el compilador C#, puede arrancar ILDASM y seguir adelante.

NOTA Para decir si un EXE o un DLL está gestionado, intente abrirlo con ILDASM. Si el archivo es un archivo gestionado válido que contiene MSIL y un manifiesto, se abrirá. Si no lo es, recibirá un mensaje de error indicando que *<su archivo> tiene una cabecera no válida y no puede ser desensamblado*.

GUÍAS DE PROGRAMACIÓN DE C#

Cerraremos este capítulo con algunas guías sobre la escritura de aplicaciones en C#.

Cuándo definir sus propios espacios de nombres

En la aplicación «Hola, mundo», utilizamos el método *Console.WriteLine* que se define en el espacio de nombre *System*. De hecho, todos los tipos y clases .NET se definen en espacios de nombres. Sin embargo, no creamos un espacio de nombres para nuestra aplicación, así que vamos a solucionar ese tema.

Los espacios de nombres son una buena manera de categorizar sus tipos y clases para evitar colisiones de nombres. Microsoft sitúa todas las definiciones de clases y tipos .NET en espacios de nombres específicos, porque quiere asegurarse de que sus nombres no tienen conflictos con los nombres de alguien que esté utilizando sus compiladores. Sin

embargo, saber si debe utilizar espacios de nombres se resume en una cuestión: ¿Se utilizarán los tipos y clases que ha creado en un entorno no controlado por usted? En otras palabras, si su código sólo lo utilizan los miembros de su propio equipo, puede crear fácilmente reglas de nomenclatura para que no haya colisiones de nombres. Sin embargo, si está escribiendo clases que utilizarán desarrolladores de terceros, en cuyo caso no tendría ningún control sobre las prácticas de nomenclatura, debería utilizar espacios de nombres sin duda alguna. Además, ya que Microsoft recomienda utilizar el nombre de su compañía como espacio de nombres de alto nivel, recomendamos utilizar espacios de nombres siempre que alguien más pueda ver su código. Llámelo publicidad gratuita.

Guías de nomenclatura

Estadísticamente, el coste mayor asociado al desarrollo de una aplicación siempre ha sido el mantenimiento. Antes de continuar, debemos emplear algunos minutos para hablar sobre las convenciones de nomenclaturas, porque elegir una convención de nomenclatura estable y que se entienda fácilmente le permitirá escribir código que sea fácil de leer y, por lo tanto, de mantener.

Como la mayoría de nosotros sabemos, las convenciones de nomenclatura son un asunto delicado. El tema era más fácil antes, cuando Visual C++ y MFC aparecieron inicialmente. Recuerdo que me enfrenté a esta cuestión cuando era desarrollador jefe de un equipo en Peachtree Software que tenía como tarea hacer la primera aplicación de contabilidad de la compañía en MFC. Era una de esas reuniones de arranque de proyecto con todo el mundo deseando irse y preparados para luchar con uñas y dientes por cualquier punto filosófico, de manera opuesta a lo que sucedió más tarde en el proyecto cuando la gente sólo quería entregar el producto a medio hacer. Según entraban en fila los desarrolladores, podría sentir sólo por el brillo de sus ojos y los textos bajo sus brazos que estaban preparados para luchar. De cara a lo que seguramente iba a ser una sangría gratuita, ¿qué hice? Aposté, ¡por supuesto! Afirmé que como gran parte del desarrollo basado en MFC se pasaba buceando en el código de Microsoft, deberíamos utilizar las convenciones de nomenclatura que Microsoft había utilizado al escribir MFC. Después de todo, sería contraproducente tener dos sistemas de nomenclatura en el código fuente: uno para MFC y otro para el nuestro. Naturalmente, el hecho de que me *guste* la notación húngara no influyó exactamente de manera negativa.

Sin embargo, es un nuevo día, y tenemos en C# un nuevo lenguaje con una serie de nuevos retos. En este entorno, no vemos el código de Microsoft. Incluso a pesar de esto, después de muchas conversaciones con el equipo de diseño de C# de Microsoft, he descubierto que está evolucionando un estándar. Podría terminar de manera ligeramente distinta de lo que presentamos aquí, pero al menos esto le dará un lugar desde el que empezar.

Convenciones de nomenclatura estándar

Antes de explicar cuándo y cómo nombrar los diferentes elementos de su aplicación, echemos un breve vistazo a los distintos estándares utilizados hoy.

Notación húngara

La notación húngara es el sistema utilizado por la mayoría de los desarrolladores de C y C++ (incluidos los de Microsoft). Es un sistema de nomenclatura completo creado por el distinguido ingeniero de Microsoft Charles Simonyi. A principios de los años ochenta, Microsoft adoptó este famoso —o infame, dependiendo de su punto de vista— sistema de nomenclatura basado en las ideas de la tesis doctoral de Simonyi, «Meta-Programming: A Software Production Method».

La notación húngara especifica que se añada un prefijo a cada variable para indicar su tipo. Sin embargo, no a todos los tipos se les daba un prefijo estándar. Además, como se presentaron otros lenguajes y se crearon nuevos tipos, se tuvieron que crear nuevos prefijos. Por eso, incluso si analiza algún código que utiliza notación húngara, podría ver algunos prefijos que no está acostumbrado a ver. (Por cierto, el término «Notación húngara» viene del hecho de que los prefijos que forman parte de las variables parecen escritos en otro idioma que no es inglés; además, el señor Simonyi es húngaro).

Quizá la publicación más importante que alentó la utilización de la notación húngara fue el primer libro leído por casi todos los desarrolladores de Windows y OS/2: *Programming Windows* (Microsoft Press), de Charles Petzold, que utilizó un dialecto de la notación húngara a través de sus ejemplos de aplicaciones. Además, Microsoft empleó la notación en su propio desarrollo. Cuando salió MFC, aparecieron muchos prefijos nuevos específicos del desarrollo de C++ con su código fuente, garantizando así la utilización continuada de la notación húngara.

Así que, ¿por qué no seguimos utilizando la notación húngara? Porque la notación húngara es útil en situaciones donde es beneficioso saber el tipo o ámbito de una variable que se está utilizando. Sin embargo, como verá con más detalle en el último capítulo, todos los tipos en C# son objetos y están basados en la clase *.NET System.Object*. Por lo tanto, todas las variables tienen un conjunto de funcionalidad y características básicas de comportamiento. Debido a esto, la necesidad de la notación húngara disminuye en el entorno .NET.

NOTA Para los curiosos, o para los que sufren de insomnio, el artículo original que presentó la notación húngara se puede encontrar en <http://msdn.microsoft.com/library/techart/hunganotat.htm>.

Notación estilo Pascal y notación estilo «camello»

Aunque no se ha podido identificar al equipo C# con un estándar «rígido», es obvio por sus escritos que están siguiendo la notación publicada por el empleado y compañero de Microsoft Rob Caron, que sugiere utilizar una mezcla de notación estilo Pascal y «camello»*

* El nombre «camello» se debe a la apariencia característica de los nombres que siguen esta notación, ya que el perfil superior simula jorobas: `variableEnNotaciónEstiloCamello` (N. de los t.).

cuando haya que nombrar variables. En su artículo «Coding Techniques and Programming Practices», disponible en MSDN (<http://msdn.microsoft.com/library/techart/cfr.htm>), sugiere utilizar notación estilo Pascal en nombres de métodos, donde el primer carácter se escribe en mayúscula, y notación estilo «camello» en nombres de variables. Estando de acuerdo, hemos escogido utilizar la misma convención de nombres en los ejemplos de aplicaciones de este libro. Sin embargo, como C# contiene más elementos además de los métodos y las variables, en las siguientes secciones hemos listado los diferentes elementos y las convenciones de nomenclatura que hemos visto que Microsoft utiliza internamente y que hemos escogido para utilizarlas también.

NOTA Para más información sobre este tema, puede dirigirse a las guías del Framework .NET incluidas en la documentación del .NET Framework SDK, en .NET Framework Developer Specifications\NET Framework Design Guidelines\Naming Guidelines.

Espacios de nombres

Utilice el nombre de su producto o compañía, y emplee mayúsculas y minúsculas con una letra mayúscula inicial; por ejemplo, *Microsoft*. Si está en el negocio de la venta de componentes software, cree un espacio de nombre de alto nivel con el nombre de su compañía, y luego por cada producto cree un espacio de nombres anidado con sus tipos internos, lo que prevendrá una colisión de nombres con otros productos. Se puede encontrar un ejemplo en el .NET Framework SDK: *Microsoft.Win32*. Esta estrategia podría funcionar para nombres demasiado largos, pero recuerde que los usuarios de su código sólo necesitan especificar la directiva *using* para ahorrarse el escribir. Si su compañía se llama Trey Research y vende dos productos —una cuadrícula y una base de datos—, llame a su espacio de nombre *TreyResearch.Grid* y *TreyResearch.Database*.

Clases

Como se supone que los objetos son entidades vivas, que respiran y que tienen habilidades, nombre a las clases utilizando sustantivos que describan el dominio del problema de la clase. En casos en que la clase sea más genérica (esto es, menos específica al dominio del problema) —por ejemplo, un tipo que representa una cadena SQL—, utilice notación estilo Pascal.

Métodos

Utilice notación estilo Pascal en todos los métodos. Se espera que los métodos actúen; llevan a cabo un trabajo. Por lo tanto, haga que los nombres de sus métodos describan qué hacen. Ejemplos de esto son *PrintInvoice* y *OpenDatabase*.

En el caso de métodos que se utilizarán en expresiones booleanas, prefije el método con un verbo que indique lo que hará el método. Por ejemplo, si tiene un método que volverá un valor booleano basado en si está bloqueada una estación de trabajo, llame al método algo así como *IsWorkStationLocked*. De esta manera, si el método se utiliza en una instrucción *if*, su significado será mucho más claro, como se muestra aquí:

```
if (IsWorkStationLocked)...
```

Parámetros de métodos

Utilice notación estilo Pascal en todos los argumentos. Ponga nombres significativos a los argumentos, para que cuando se utilice IntelliSense, el usuario pueda ver inmediatamente para qué se utiliza cada argumento.

Interfaces

Utilice notación estilo Pascal en todas las interfaces. Es común prefijar los nombres de interfaces con una «I» mayúscula; por ejemplo, *Comparable*. (Esta es la única convención parecida a la notación húngara de la que tengo constancia en C#).

Muchos desarrolladores utilizan las mismas reglas cuando nombran interfaces que cuando nombran clases. Sin embargo, hay una diferencia fundamental filosófica entre las dos. Las clases representan una encapsulación de datos y de funciones que funcionan sobre esos datos. Las interfaces, por otra parte, representan el comportamiento. Implementando una interfaz, está diciendo que una clase puede exhibir ese comportamiento. Por lo tanto, es una técnica común nombrar interfaces con adjetivos. Por ejemplo, una interfaz que declara métodos para serializar datos, podría llamarse *ISerializable* para denotar fácilmente la capacidad que se está adquiriendo al implementar la clase.

Miembros de clase

Este es probablemente el asunto más complicado con relación a los desarrolladores de C#. Aquellos de nosotros que venimos de C++ y MFC estamos acostumbrados a prefijar los nombres de miembros con *m_*. Sin embargo, recomendaría utilizar notación estilo «camello», en el que la primera letra no se pone en mayúscula. De esa manera, si tiene un método que toma como argumento algo llamado *Foo*, puede diferenciarlo de la representación interna de esa variable creando el miembro interno llamado *foo*.

Es redundante prefijar un nombre de miembro con el nombre de la clase. Tome, por ejemplo, un nombre de clase de *Author*. En lugar de crear un miembro llamado *AuthorName*, que se accedería como *Author.AuthorName*, utilice *Name* como nombre del miembro.

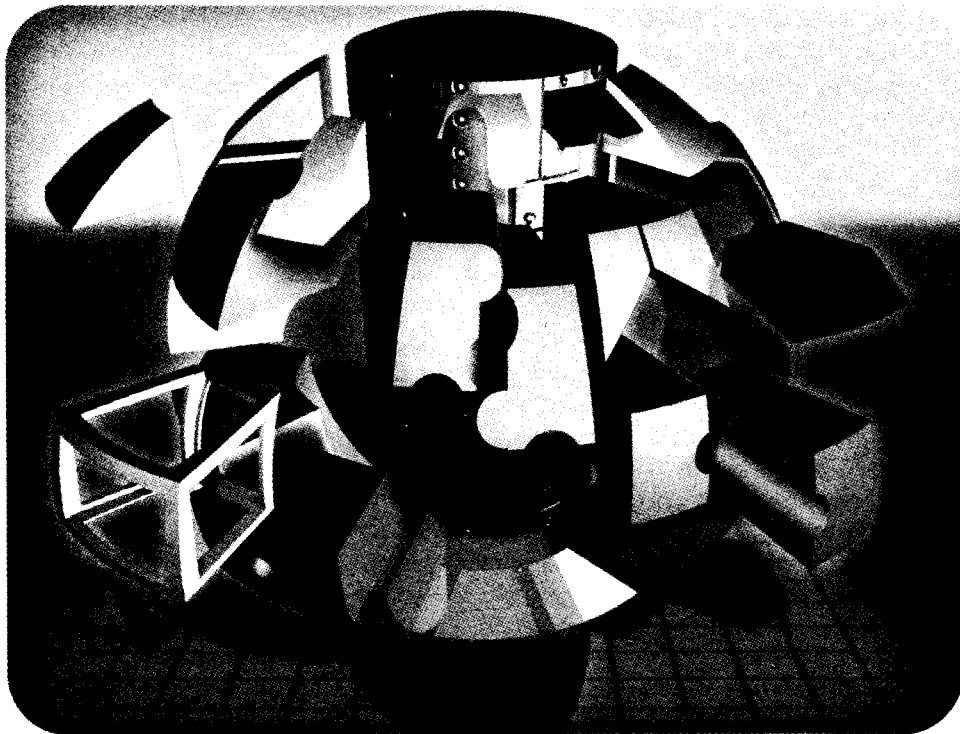
RESUMEN

Escribir, compilar y ejecutar un programa escrito en C# es un primer paso importante en la exploración del lenguaje. Aunque en general no importa qué programa utilice para editar sus archivos fuente C#, hay beneficios al utilizar editores y entornos más eficaces diseñados para el desarrollo de C#. Conocer las opciones y parámetros del compilador C# le permitirá tomar el control de cómo genera código MSIL el compilador. Puede explorar ese código utilizando herramientas como ILDASM, incluido en el Microsoft .NET Framework SDK.

La estructura de los programas C# proporciona un gran número de características diseñadas para hacer los programas más seguros, fáciles de escribir y menos propensos a tener defectos o errores. Están incluidos en este conjunto de características los espacios de nombres y la directiva *using*. Finalmente, las convenciones de nomenclatura, que incluyen convenciones de notación específicas, pueden hacer los programas más legibles y fáciles de mantener.

Parte II

FUNDAMENTOS DE C#



Capítulo 4

El sistema de tipos

En el centro de Microsoft .NET Framework hay un sistema de tipos universal llamado .NET Common Type System (CTS). Además de definir todos los tipos, el CTS estipula también las reglas que sigue el Common Language Runtime (CLR) en relación con las aplicaciones que declaran y utilizan estos tipos. En este capítulo veremos el nuevo sistema de tipos para que pueda aprender qué tipos están disponibles para los desarrolladores de C# y entender las distintas variantes de su utilización en programas C#. Empezaremos explorando el concepto de que todo elemento de programación es un objeto en .NET. Después echaremos un vistazo a cómo .NET divide los tipos en dos categorías —tipos valor y tipos referencia— y descubriremos cómo el empaquetamiento (*boxing*) permite que un sistema de tipos completamente orientado a objetos funcione eficientemente. Finalmente, analizaremos cómo funciona la conversión de tipos (*casting*) en C# y echaremos un vistazo a los espacios de nombres.

TODO ES UN OBJETO

La mayoría de los lenguajes orientados a objetos tienen dos tipos distintos: aquellos tipos que son intrínsecos al lenguaje (*tipos primitivos*) y los tipos que pueden crear los usuarios del lenguaje (*clases*). Como puede adivinar, los tipos primitivos son normalmente tipos sencillos, como caracteres, cadenas y números, y las clases tienden a ser tipos más elaborados.

El tener dos grupos discretos de tipos causa muchos problemas. Un problema se relaciona con la compatibilidad. Por ejemplo, digamos que usted quería tener una colección de *ints* en un sistema tradicional con estos dos grupos de tipos. Necesitaría crear una clase específicamente para mantener valores de tipo *int*. Y si quisiera una clase para mantener una colección de *doubles*, tendría que hacer lo mismo por ese tipo. La razón de estas dos clases distintas es que estos tipos primitivos normalmente no tienen nada en común. No son objetos reales, por eso no derivan de una clase base común. Son más como tipos mágicos que tienen que tratarse individualmente en sus propios términos. Aparece un problema parecido en estos sistemas tradicionales cuando quiere especificar

que un método puede tomar un parámetro de *cualquier* tipo admitido por el lenguaje. Como éstos tipos primitivos son incompatibles, no puede especificar un parámetro como éste, a menos que escriba clases que recubran cada tipo primitivo.

Afortunadamente, esto ya no es un problema en el mundo de .NET y C#, porque todo es un objeto en el CTS. De hecho, no todo es un objeto, sino que sucede algo incluso más importante: todos los objetos derivan implícitamente de una sola clase base definida como parte del CTS. Esta clase base, llamada *System.Object*, la trataremos en la sección «Empaquetado y desempaquetado».

TIPOS VALOR Y TIPOS REFERENCIA

Crear un lenguaje donde todo es un objeto no es un concepto nuevo. Otros lenguajes lo han intentado, siendo el más famoso SmallTalk. La mayor desventaja para hacer que todo sea un objeto siempre ha sido que el rendimiento es pobre. Por ejemplo, si intenta sumar dos valores de tipo *double* en SmallTalk, se reserva memoria para un objeto en el montón (*heap*). No es preciso decir que la reserva de memoria para un objeto es extremadamente ineficiente cuando todo lo que quería hacer era sumar dos números.

Los diseñadores del CTS se enfrentaron a la tarea de crear un sistema de tipos en el que todo fuera un objeto pero en el que el sistema de tipos funcionara de manera eficiente, cuando fuera posible. Su solución fue separar los tipos CTS en dos categorías: *tipos valor* y *tipos referencia*. Estos términos, como verá pronto, reflejan cómo se reserva memoria para las variables y cómo funcionan internamente.

Tipos valor

Cuando tiene una variable que es un tipo de valor, tiene una variable que contiene datos reales. Así, la primera regla de tipos valor es que no pueden ser *nulos* (*null*). Por ejemplo, más abajo hemos reservado memoria creando una variable de tipo CTS *System.Int32* en C#. En esta declaración, lo que sucede es que se ha reservado un espacio de 32 bits en la pila.

```
int i = 32;
```

Además, la asignación de un valor a *i* hace que se mueva un valor de 32 bits al espacio de memoria reservado.

Hay varios tipos valor definidos en C#, incluyendo los enumeradores, las estructuras y los tipos primitivos. En cualquier momento en que declare una variable de uno de esos tipos, se reserva memoria en la pila para almacenar el número de bytes asociado con ese tipo y se trabaja directamente con ese array de bits reservado. Además, cuando se pasa una variable de tipo valor, se está pasando el valor de esa variable y no una referencia al objeto subyacente.

Tipos referencia

Los tipos referencia son parecidos a las referencias de C++ en que son punteros de tipo seguro. El ser de tipo seguro significa que en lugar de ser meramente una dirección, que pudiera o no apuntar a lo que cree que apunta, para una referencia (cuando no es *null*) siempre se garantiza que va a apuntar a un objeto que es del tipo especificado y cuya memoria ha sido reservada en el montón. Tome nota también del hecho de que una referencia puede ser nula (*null*).

En el siguiente ejemplo se va a reservar memoria para un tipo referencia (*string*). Sin embargo, lo que pasa internamente es que el valor se ha reservado en el montón y se ha devuelto una referencia a ese valor.

```
string s = "Hola, mundo";
```

Como sucede con los tipos valor, hay varios tipos definidos como tipos referencia en C#: clases, arrays, delegados e interfaces. En cualquier momento que declare una variable de uno de esos tipos, se reserva en memoria el número de bytes asociados con ese tipo en el montón, y está trabajando con una referencia a ese objeto en lugar de trabajar directamente con los bits (que es lo que sucede con los tipos valor).

EMPAQUETADO Y DESEMPAQUETADO

Y aparece la pregunta, «¿Cómo hacen el sistema más eficiente estas diferentes categorías de tipos?». Se hace a través de la magia del *empaquetado*. En su acepción más simple, es la conversión de un tipo de valor en un tipo de referencia. El caso recíproco se da cuando un tipo de referencia se *desempaqueta* de vuelta a un tipo valor.

Lo que hace a esta técnica tan importante es que un objeto es sólo un objeto cuando necesita serlo. Por ejemplo, digamos que declara un tipo valor como *System.Int32*. La memoria para esta variable se reserva en el montón. Puede pasar esta variable a cualquier método definido que acepta un tipo *System.Object* y acceder a cualquiera de los miembros para los que tenga acceso. Por lo tanto, a usted le parece y lo maneja como un objeto. Sin embargo, en realidad, son sólo 4 bytes de la pila.

Sólo cuando intenta utilizar esa variable de manera consistente con la interfaz de clase base *System.Object* implicada es cuando el sistema empaqueta automáticamente la variable para que llegue a ser un tipo referencia y pueda utilizarse como un objeto. Mediante el empaquetado, C# hace posible que todo parezca ser un objeto, evitando así la sobrecarga requerida en caso de que todo fuera realmente un objeto. Echemos un vistazo a algunos ejemplos para centrarnos en esto.

```
int foo = 42;           //Tipo valor.
object bar = foo;       //foo se empaqueta en bar.
```

En la primera línea del código, estamos creando una variable (*foo*) de tipo *int*. Como ya sabe, el tipo *int* es un tipo valor (porque es un tipo primitivo). En la segunda línea,

el compilador ve que la variable *foo* se está copiando a un tipo referencia, que se representa por la variable *bar*. El compilador genera entonces el código MSIL necesario para empaquetar este valor.

Ahora, para volver a convertir *bar* en un tipo valor, puede realizar una conversión explícita de tipo:

```
int foo = 42;           //Tipo valor.
object bar = foo;       //foo se empaqueta en bar.
int foo2 = (int)bar;     //Se desempaqueta a entero.
```

Observe que cuando se empaqueta —esto es, una vez más, cuando pasa de un tipo valor a un tipo de referencia— no se necesita una conversión explícita de tipo. Sin embargo, cuando se desempaqueta —pasando de un tipo referencia a un tipo valor— se necesita la conversión de tipo. Esto es porque en caso de desempaquetar, un objeto podría transformarse a cualquier tipo. Por lo tanto, la conversión de tipo es necesaria para que el compilador pueda comprobar que ésta es válida para el tipo de variable especificado. Como la conversión de tipo implica unas reglas estrictas y como estas reglas están gobernadas por el CTS, más adelante echaremos un vistazo con más detalle a este tema en «Conversión entre tipos», en este mismo capítulo.

LA RAÍZ DE TODOS LOS TIPOS: *SYSTEM.OBJECT*

Como ya hemos mencionado, todos los tipos derivan al final del tipo *System.Object*, garantizando así que cada tipo del sistema tiene un conjunto mínimo de habilidades. La Tabla 4.1 describe los cuatro métodos públicos que tienen todos los tipos por esta razón.

Tabla 4.1. Métodos públicos del tipo *System.Object*

Nombre del método	Descripción
<i>bool Equals()</i>	Este método compara dos referencias a objeto en tiempo de ejecución para determinar si son exactamente el mismo objeto. Si las dos variables se refieren al mismo objeto, el valor devuelto es cierto. En el caso de los tipos valor, que se tratarán en la próxima sección, este método devuelve cierto si los dos tipos son idénticos y tienen el mismo valor.
<i>int GetHashCode()</i>	Recupera el código de dispersión (<i>hash</i>) especificado para un objeto. Las funciones de dispersión se utilizan cuando quien implementa una clase quiere poner el código de dispersión de un objeto en una tabla de dispersión por motivos de rendimiento.
<i>Type GetType()</i>	Utilizado con los métodos de reflexión (analizados en el Capítulo 16, «Cómo obtener información sobre metadatos con Reflection») para recuperar la información del tipo para un objeto dado.
<i>string ToString</i>	Por defecto, este método se utiliza para recuperar el nombre del objeto. Puede ser redefinido por clases derivadas para que se devuelva una cadena que represente el objeto de forma más amigable para el usuario.

La Tabla 4.2 describe los métodos protegidos del *System.Object*.

Tabla 4.2. *Métodos protegidos del tipo System.Object*

Nombre del método	Descripción
<i>void Finalize()</i>	Este método lo invoca el entorno de ejecución para permitir que se liberen recursos antes de que entre el recolector de basura. Observe que este método puede no invocarse. Por lo tanto, no ponga código que deba ejecutarse en este método. Esta regla forma parte de algo llamado <i>finalización determinista</i> , que trataremos en detalle en el Capítulo 5, «Clases».
<i>Object MemberwiseClone</i>	Este miembro representa una <i>copia superficial</i> del objeto. Con esto queremos decir una copia de un objeto que contiene referencias a otros objetos donde no se incluyen copias de los objetos referenciados. Si necesita que su clase soporte una <i>copia profunda</i> , que incluye copia de los objetos referenciados, debe implementar la interfaz <i>ICloneable</i> y clonarlo manualmente, o copiarlo, usted mismo.

TIPOS Y ALIAS

Mientras el CTS es responsable de definir tipos que se pueden utilizar en lenguajes .NET, la mayoría de los lenguajes escogen implementar alias a esos tipos. Por ejemplo, un valor entero de 4 bytes se representa por el tipo CTS *System.Int32*. C# entonces define para él un alias llamado *int*. No hay ventajas por utilizar una técnica en lugar de la otra. La Tabla 4.3 lista los diferentes tipos CTS y sus alias de C#.

CONVERSIÓN ENTRE TIPOS

En este punto, vamos a ver uno de los aspectos más importantes de los tipos: la conversión de tipos. Si asumimos una clase base llamada *Employee* y una clase derivada llamada *ContractEmployee*, el siguiente código funciona porque siempre hay una conversión de tipo asociada desde una clase derivada a su clase base.

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample1
{
    public static void Main()
    {
        Employee e = new ContractEmployee();
    }
}
```

Tabla 4.3. Tipos y Alias CTS

Nombre de tipo CTS	Alias C#	Descripción
<i>System.Object</i>	<i>object</i>	Clase base para todos los tipos CTS.
<i>System.String</i>	<i>string</i>	Cadena.
<i>System.SByte</i>	<i>sbyte</i>	Byte con signo de 8 bits.
<i>System.Byte</i>	<i>byte</i>	Byte sin signo de 8 bits.
<i>System.Int16</i>	<i>short</i>	Valor de 16 bits con signo.
<i>System.UInt16</i>	<i>ushort</i>	Valor de 16 bits sin signo.
<i>System.Int32</i>	<i>int</i>	Valor de 32 bits con signo.
<i>System.UInt32</i>	<i>uint</i>	Valor de 32 bits sin signo.
<i>System.Int64</i>	<i>Long</i>	Valor de 64 bits con signo.
<i>System.UInt64</i>	<i>ulong</i>	Valor de 64 bits sin signo.
<i>System.Char</i>	<i>char</i>	Carácter unicode de 16 bits.
<i>System.Single</i>	<i>float</i>	Valor en coma flotante del IEEE de 32 bits.
<i>System.Double</i>	<i>double</i>	Valor en coma flotante del IEEE de 64 bits.
<i>System.Boolean</i>	<i>bool</i>	Valor booleano (<i>true/false</i>).
<i>System.Decimal</i>	<i>decimal</i>	Tipo de datos de 128-bit igual a 28 o 29 dígitos —principalmente utilizado para aplicaciones financieras, donde se requiere un alto grado de precisión.

Sin embargo, lo siguiente es ilegal, porque el compilador no puede proporcionar una conversión de clase base a derivada de manera implícita:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample2
{
    public static void Main()
    {
        ContractEmployee ce = new Employee(); //No compilará.
    }
}
```

La razón del comportamiento diferente se remite al Capítulo 1, «Fundamentos de la programación orientada a objetos», y el concepto de sustitutibilidad. Recuerde que las reglas de sustitutibilidad afirman que una clase derivada se puede utilizar en lugar de su clase de base. Por lo tanto, un objeto de tipo *ContractEmployee* debería poder utilizarse siempre en lugar de o como si fuera un objeto *Employee*. Por eso compila el primer ejemplo.

Sin embargo, usted no puede hacer una conversión desde un objeto de tipo *Employee* a un objeto de tipo *ContractEmployee* porque no hay garantía de que el objeto vaya a ser compatible con la interfaz definida por la clase *ContractEmployee*. Por lo tanto, en caso de una conversión al tipo de una clase derivada, se utiliza una conversión de tipo explícita de la siguiente forma:

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample3
{
    public static void Main()
    {
        //La conversión al tipo de la clase derivada fallará.
        ContractEmployee ce = (ContractEmployee) new Employee();
    }
}
```

Pero ¿qué pasa si mentimos e intentamos engañar al CTS haciendo una conversión explícita de tipo de una clase base a una clase derivada como sigue?

```
class Employee { }

class ContractEmployee : Employee { }

class CastExample4
{
    public static void Main()
    {
        Employee e = new Employee();
        ContractEmployee c = (ContractEmployee) e;
    }
}
```

El programa compila, pero al ejecutar el programa se genera una excepción en tiempo de ejecución. Hay dos cosas que observar aquí. Primero, el resultado no es un error de tiempo de compilación, porque *e* podía realmente haber sido un objeto cuyo tipo se hubiera convertido a *ContractEmployee*. Por lo tanto, la verdadera naturaleza del objeto no puede conocerse hasta el momento en que se ejecute. Segundo, el CLR determina los tipos de los objetos en tiempo de ejecución. Cuando el CLR reconoce una conversión de tipo inválida, lanza una excepción *System.InvalidCastException*.

Hay otra manera de hacer conversiones de tipo con objetos: utilizando la palabra reservada *as*. La ventaja de utilizar esta palabra reservada en lugar de una conversión de tipo es que si la conversión es inválida, usted no tiene que preocuparse por que se lance una excepción. Lo que pasará, en cambio, es que el resultado será *null*. Aquí tenemos un ejemplo:

```
using System;

class Employee { }
```

```

class ContractEmployee : Employee { }

class CastExample5
{
    public static void Main()
    {
        Employee e = new Employee();
        Console.WriteLine("e = {0}",
            e == null ? "nulo" : e.ToString());

        ContractEmployee c = e as ContractEmployee;
        Console.WriteLine("c = {0}",
            c == null ? "nulo" : c.ToString());
    }
}

```

Si ejecuta este ejemplo, verá el siguiente resultado:

```

c:>CastExample5
e = Employee
c = nulo

```

Observe que la capacidad de comparar un objeto con *null* implica que no tiene que correr el riesgo de utilizar un objeto nulo. De hecho, si el ejemplo hubiera intentado invocar un método de *System.Object* en el objeto *c*, el CTS habría lanzado una excepción *System.NullReferenceException*.

ESPACIOS DE NOMBRES

Los *espacios de nombres* se utilizan para definir el ámbito en las aplicaciones C#. Declarando un espacio de nombres, el desarrollador puede dar a una aplicación C# una estructura jerárquica basada en grupos de tipos relacionados semánticamente y de otros espacios de nombres (anidados). Muchos archivos de código fuente pueden contribuir al mismo espacio de nombres. Hasta ese punto, si está agrupando varias clases en un determinado espacio de nombres, puede definir cada una de esas clases en su propio archivo de código fuente. Un programador que emplee sus clases puede tener acceso a todas las clases de un mismo espacio de nombres a través de la palabra reservada *using*.

NOTA Se recomienda, donde se pueda aplicar, un nombre de compañía como nombre del espacio de nombres raíz para ayudar a asegurar la exclusividad. Vea el Capítulo 3, «Hola, C#», para más información sobre guías de nomenclatura.

La palabra reservada *using*

A veces querrá utilizar el nombre entero calificado para un determinado tipo con la forma *espacionombres.tipo*. Sin embargo, esto puede ser bastante tedioso y a veces no es ne-

cesario. En el siguiente ejemplo, estamos utilizando el objeto *Console* que existe en el espacio de nombre *System*.

```
class Using1
{
    public static void Main()
    {
        System.Console.WriteLine("prueba");
    }
}
```

Sin embargo, ¿qué pasa si sabemos que el objeto *Console* existe sólo en el espacio de nombre *System*? La palabra reservada *using* nos permite especificar un orden de búsqueda de espacios de nombres para que cuando el compilador encuentre un tipo no calificado, pueda mirar en los espacios de nombres listados para buscar el tipo. En el siguiente ejemplo, el compilador localiza el objeto *Console* en el espacio de nombre *System* sin que el desarrollador tenga que especificarlo siempre.

```
using System;

class Using2
{
    public static void Main()
    {
        Console.WriteLine("prueba");
    }
}
```

Cuando empiece a construir aplicaciones reales con varios cientos de invocaciones a distintos objetos de *System*, verá rápidamente la ventaja de no tener que anteponer a cada una de esas referencias de objetos el nombre del espacio de nombres.

No puede especificar un nombre de clase con la palabra reservada *using*. Por lo tanto, lo siguiente no sería válido:

```
using System.Console; //no válido

class Using3
{
    public static void Main()
    {
        WriteLine("prueba");
    }
}
```

Lo que puede hacer, sin embargo, es utilizar una variante de la palabra reservada *using* para crear un alias *using*:

```
using console = System.Console;

class Using4
{
}
```

```
public static void Main()  
{  
    console.WriteLine("prueba");  
}
```

Esto es especialmente ventajoso en casos en los que los espacios de nombres anidados combinados con nombres largos de clases hacen tedioso escribir código.

BENEFICIOS DE CTS

Una de las características clave de cualquier lenguaje o entorno de ejecución es su soporte para los tipos. Un lenguaje que dispone de un número limitado de tipos o que limita la capacidad del programador para extender sus tipos integrados no es un lenguaje con gran esperanza de vida. En cambio, el tener un sistema de tipos unificado tiene además muchos beneficios.

Interoperabilidad del lenguaje

El CTS juega un papel esencial a la hora de permitir la interoperabilidad del lenguaje, porque define el conjunto de tipos que debe admitir un compilador .NET para interoperar con otros lenguajes. El propio CTS se define en la Common Language Specification (CLS). El CLS define un conjunto único de reglas para cada compilador .NET, asegurándose que cada compilador genere código que interactúe consistentemente con el CLR. Uno de los requisitos del CLS es que el compilador debe admitir ciertos tipos definidos en el CTS. El beneficio en esto es que al utilizar todos los lenguajes .NET un sistema de tipos único, se le asegura que los objetos y los tipos creados en diferentes lenguajes pueden interactuar con cualquier otro de forma directa. Es esta combinación CTS/CLS la que ayuda a hacer la interoperabilidad del lenguaje algo más que sólo un sueño de programador.

Jerarquía de objetos de raíz única

Como mencionamos anteriormente, una característica importante del CTS es la jerarquía de objetos de una única raíz. En el .NET Framework, todos los tipos del sistema derivan de la clase base *System.Object*. Una diferencia importante con el lenguaje C++, en el que no hay clases base definidas para todas las clases, es esta aproximación de una única clase base, que está respaldada por los teóricos de la POO y se implementa en la mayoría de los principales lenguajes orientados a objetos. Los beneficios de una jerarquía de raíz única no son aparentes inmediatamente, pero con el tiempo se preguntará cuántos lenguajes fueron diseñados antes de que se adoptara este tipo de jerarquía.

Una jerarquía de objetos de una única raíz es la clave para un sistema de tipos unificado, porque garantiza que todos los objetos de la jerarquía tengan una interfaz

común, y por lo tanto, todo en la jerarquía, será al final del mismo tipo base. Uno de los principales inconvenientes de C++ es la falta de soporte para una jerarquía de ese tipo. Consideremos un ejemplo sencillo para ver lo que queremos decir.

Digamos que ha construido una jerarquía de objetos en C++ basada en una clase de base llamada *CFoo*. Ahora, supongamos que quiere integrar con otra jerarquía de objetos cuyos objetos derivan todos de una clase base llamada *CBar*. En este ejemplo, las jerarquías de objeto tienen interfaces incompatibles y será preciso mucho esfuerzo para integrar estas dos jerarquías. Tendría que utilizar algún tipo de clase adaptadora con agregación o utilizar herencia múltiple para hacer este trabajo. Con una jerarquía de una única raíz, la compatibilidad no es un problema, porque todos los objetos tienen la misma interfaz (heredada de *System.Object*). Como resultado, sabe que todos y cada uno de los objetos de su jerarquía —y sobre todo en las jerarquías de código .NET de terceros— tienen un conjunto mínimo de funcionalidad.

Seguridad de tipo

El último beneficio de CTS que mencionaremos es la seguridad de tipo. La seguridad de tipo garantiza que los tipos sean lo que dicen que son y que sólo se puedan realizar operaciones apropiadas en un tipo particular. La seguridad de tipo proporciona una serie de ventajas y capacidades —como se describe a continuación—, la mayoría de las cuales tiene su origen en la jerarquía de objetos de raíz única.

- Cada referencia a un objeto tiene tipo, y el objeto al que se hace referencia también lo tiene. El CTS garantiza que una referencia siempre apunta a lo que indica que apunta.
- Como el CTS sigue la huella de todos los tipos del sistema, no hay manera de engañarle para que piense que un tipo es realmente otro. Esto es, obviamente, una preocupación importante para las aplicaciones distribuidas en las que la seguridad es una prioridad.
- Cada tipo es responsable de definir la accesibilidad de sus miembros especificando un modificador de acceso. Esto se hace miembro a miembro, e incluye permitir cualquier acceso (declarando el miembro *public*); limitar la visibilidad sólo a clases heredadas (declarando el miembro *protected*); no permitir ningún acceso (declarando el miembro *private*), y permitir el acceso sólo a otros tipos en la unidad de compilación actual (declarando el miembro *internal*). Trataremos los modificadores de acceso con más detalle en el próximo capítulo.

RESUMEN

El Common Type System es una característica importante del .NET Framework. El CTS define las reglas del sistema de tipos que las aplicaciones deben seguir para ejecutarse en el CLR. Los tipos CTS se dividen en dos categorías: tipos referencia y

tipos valor. Los espacios de nombres se pueden utilizar para definir el ámbito en una aplicación. Los beneficios de un sistema de tipos común incluyen la interoperabilidad del lenguaje, una jerarquía de objetos de raíz única y seguridad de tipos. Los tipos se pueden convertir en C# mediante empaquetado y desempaquetado y se pueden crear tipos compatibles que compartan características y funcionalidad mediante las conversiones de tipo.

Capítulo 5

Clases

Las clases están en el centro de todo lenguaje orientado a objetos. Como se expuso en el Capítulo 1, «Fundamentos de la programación orientada a objetos», una clase es la encapsulación de datos y métodos que operan sobre esos datos. Eso es cierto en cualquier lenguaje orientado a objetos. Lo que diferencia a los lenguajes a partir de ahí son los tipos de datos que puede almacenar como miembros y las capacidades de cada tipo de clase. Con las clases, como con muchas de las características del lenguaje, C# toma prestado un poco de C++ y Java y añade algo de ingenuidad para crear soluciones elegantes para viejos problemas.

En este capítulo, primero describiremos lo más básico de la definición de clases en C#, incluyendo ejemplos de miembros de instancias, modificadores de acceso, constructores y listas de inicialización, y después veremos la definición de miembros estáticos y la diferencia entre campos constantes y de sólo lectura. Después de eso, trataremos los destructores y la finalización determinista. Finalmente, el capítulo concluirá con un tratamiento rápido de la herencia y las clases de C#.

CÓMO DEFINIR CLASES

La sintaxis utilizada para definir clases en C# es sencilla, especialmente si programa habitualmente en C++ o Java. Coloque la palabra reservada *class* delante del nombre de su clase y después inserte los miembros de la clase entre las «llaves», así:

```
class Employee
{
    private long employeeId;
}
```

Como puede ver, esta clase es tan básica como parece. Tenemos una clase llamada *Employee* que contiene un único miembro llamado *employeeId*. Observe la palabra reservada *private* que precede a nuestro miembro. Es un *modificador de acceso*. Se definen cuatro modificadores de acceso válidos en C#, y los trataremos en breve.

MIEMBROS DE CLASE

En el Capítulo 4, «El sistema de tipos», se trataron los diferentes tipos definidos por el Common Type System (CTS). Estos tipos se admiten como miembros de clases de C# e incluyen lo siguiente:

- **Campos.** Un campo es una variable miembro utilizada para almacenar un valor. En la jerga de la POO, a veces se hace referencia a los campos como los datos del objeto. Puede aplicar varios modificadores a un campo, dependiendo de cómo quiera utilizarlo. Estos modificadores incluyen *static*, *readonly* y *const*. En el apartado siguiente veremos lo que significan esos modificadores y cómo se usan.
- **Métodos.** Un método es el código real que opera sobre los datos (o campos) del objeto. En este capítulo nos centraremos en la definición de los datos de clase; el Capítulo 6, «Métodos», tratará los métodos con mucho más detalle.
- **Propiedades.** Las propiedades a veces se llaman *smart fields* (campos inteligentes), porque en realidad son métodos que parecen campos de cara a los clientes de la clase. Esto permite un mayor grado de abstracción para el cliente, ya que no tiene que saber si está accediendo al campo directamente o se está invocando un método de acceso. El Capítulo 7, «Propiedades, arrays e indizadores», cubre en detalle las propiedades.
- **Constantes.** Como el nombre sugiere, una constante es un campo con un valor que no se puede cambiar. Más tarde, en este capítulo, trataremos las constantes y las compararemos con los campos *de sólo lectura*.
- **Indizadores.** De la misma forma que una propiedad es un campo inteligente, un indizador es un array inteligente; esto es, un miembro que permite que un objeto sea indizado a través de métodos de acceso *get* y *set*. Un indizador le permite indizar fácilmente un objeto para propósitos de asignación o recuperación de valores. Tanto los indizadores como las propiedades se tratan en el Capítulo 7.
- **Eventos.** Un evento es algo que hace que se ejecute un trozo de código. Los eventos forman parte integral de la programación de Microsoft Windows. Por ejemplo, se lanza un evento cuando se mueve el ratón o se hace clic en una ventana o se cambia su tamaño. Los eventos de C# utilizan el patrón de diseño estándar Publicación/Suscripción visto en Microsoft Message Queuing (MSMQ) y en el modelo de eventos asíncronos de COM+ —que le da a una aplicación capacidades de manejo de eventos asíncronos—, pero en C# este patrón de diseño es un concepto de «primera clase» integrado en el lenguaje. El Capítulo 14, «Delegados y manejadores de eventos», describe cómo utilizar los eventos.
- **Operadores.** C# le da la posibilidad, por medio de la sobrecarga de operadores, de añadir los operadores matemáticos estándares a una clase para que, utilizándolos, pueda escribir código más intuitivo. La sobrecarga del operador se trata en detalle en el Capítulo 13, «Sobrecarga de operadores y conversiones definidas por el usuario».

MODIFICADORES DE ACCESO

Ahora que hemos observado los diferentes tipos que se pueden definir como miembros de una clase C#, echemos un vistazo a unos importantes modificadores utilizados para especificar lo visible o accesible que es un determinado miembro para el código externo a su propia clase. Estos modificadores se llaman *modificadores de acceso* y se listan en la Tabla 5.1.

Tabla 5.1. *Modificadores de Acceso de C#*

Modificador de acceso	Descripción
<i>public</i>	Significa que el miembro es accesible desde fuera de la clase que lo define y de la jerarquía de las clases derivadas.
<i>protected</i>	El miembro no es visible desde fuera de la clase y se puede acceder a él sólo desde clases derivadas.
<i>private</i>	No se puede acceder al miembro desde fuera del ámbito de la clase que lo define. Por lo tanto, ni siquiera las clases derivadas tienen acceso a estos miembros.
<i>internal</i>	El miembro es visible sólo desde la unidad de compilación actual. El modificador de acceso <i>internal</i> crea un híbrido de acceso <i>public</i> y <i>protected</i> dependiendo de donde resida el código.

Observe que, a menos que quiera que un miembro tenga por defecto el modificador de acceso *private*, debe especificar un modificador de acceso. Esto contrasta con C++, donde un miembro que no se ha decorado explícitamente con un modificador de acceso toma las características de visibilidad del modificador de acceso anteriormente establecido. Por ejemplo, en el siguiente código C++, los miembros *a*, *b* y *c* se definen con visibilidad *public*, y los miembros *d* y *e* se definen como miembros *protected*:

```
class CAccessModsInCpp
{
    public:
        int a;
        int b;
        int c;
    protected:
        int d;
        int e;
}
```

Para conseguir el mismo objetivo en C#, este código tendría que cambiarse de esta manera:

```
class AccessModsInCSharp
{
    public int a;
```

```

    public int a;
    public int a;
    protected int d;
    protected int d;
}

```

El siguiente código C# hace que el miembro *b* sea declarado como *private*:

```

public MoreAccessModsInCSharp
{
    public int a;
    int b;
}

```

EL MÉTODO *MAIN*

Toda aplicación C# debe tener un método *Main* definido en una de sus clases. Además, este método debe definirse como *public* y *static*. (Entraremos en lo que significa *static* en breve). Al compilador C# no le importa qué clase tiene el método *Main* definido, ni si la clase que se elige afecta al orden de compilación. Esto es diferente en C++, donde las dependencias deben analizarse muy de cerca cuando se construye una aplicación. El compilador C# es lo bastante inteligente para recorrer los archivos fuente y localizar el método *Main* por sí solo. Así, este método tan importante es el punto de entrada a todas las aplicaciones C#.

Aunque pueda colocarse el método *Main* en cualquier clase, es recomendable crear una clase específica para alojar el método *Main*. He aquí un ejemplo de esa utilización con nuestra —hasta ahora— sencilla clase *Employee*:

```

class Employee
{
    private int employeeId;
}
class AppClass
{
    static public void Main()
    {
        Employee emp = new Employee();
    }
}

```

Como puede verse, el ejemplo tiene dos clases. Esta es una aproximación común a la programación C# incluso en las aplicaciones más sencillas. La primera clase (*Employee*) es una clase del dominio del problema, y la segunda clase (*AppClass*) contiene el necesario punto de entrada (*Main*) para la aplicación. En este caso, el método *Main* instancia el objeto *Employee*, y si fuera una aplicación real, utilizaría los miembros del objeto *Employee*.

Argumentos de línea de comandos

Se puede acceder a los argumentos de línea de comandos de una aplicación declarando el método *Main* de forma que tome un tipo array de cadenas como único argumento. Llegados a ese punto, el argumento puede procesarse como haríamos con cualquier array. Aunque los arrays no se tratarán hasta el Capítulo 7, he aquí algo de código genérico para iterar por los argumentos de la línea de comandos de una aplicación y para escribirlos en el dispositivo estándar de salida:

```
using System;
class CommandLineApp
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            Console.WriteLine("Argumento: {0}", arg);
        }
    }
}
```

Y he aquí un ejemplo de invocación a esta aplicación con un par de valores seleccionados aleatoriamente:

```
e:>CommandLineApp 5 42
Argumento: 5
Argumento: 42
```

Los parámetros de línea de comandos se dan en un array de cadenas. Si quisiera procesar los parámetros como si fueran opciones o indicadores, tendrá que programar esa capacidad usted mismo.

NOTA Los desarrolladores de Microsoft Visual C++ ya están acostumbrados a iterar sobre un array que representa los argumentos de la línea de comandos de una aplicación. Sin embargo, a diferencia de Visual C++, el array creado en C# a partir de la línea de comandos no contiene el nombre de la aplicación como primera entrada del array.

Valores de retorno

La mayoría de los ejemplos de este libro define *Main* de la siguiente manera:

```
class SomeClass
{
    public static void Main()
    {
```

```

    :
}
    :
}

```

Sin embargo, también puede definirse *Main* de forma que devuelva un valor de tipo *int*. Aunque no es común en las aplicaciones de GUI (Interfaz gráfica de usuario=*Graphical User Interface*), puede ser útil cuando se escriben aplicaciones de consola que se diseñan para ejecutarse por lotes. La instrucción *return* termina la ejecución del método y el valor devuelto se utiliza como nivel de error para la aplicación o el archivo de lotes que lo invocaron, para indicar el éxito o el fracaso definidos por el usuario. Para hacer esto, utilice el siguiente prototipo:

```

public static int Main()
{
    :
    //Devolver algún valor de tipo int
    //que representa el éxito o un valor.
    return 0;
}

```

Múltiples métodos *Main*

Los diseñadores de C# incluyeron un mecanismo por el cual se puede definir más de una clase con un método *Main*. ¿Por qué querría hacer eso? Una razón podría ser el colocar código de pruebas en sus clases. En ese caso puede utilizar la opción */main:* <nombre-Clase> con el compilador para especificar qué método *Main* de qué clase tiene que utilizarse. A continuación tenemos un ejemplo en el que hay dos clases que contienen métodos *Main*:

```

using System;

class Main1
{
    public static void Main()
    {
        Console.WriteLine("Main1");
    }
}

class Main2
{
    public static void Main()
    {
        Console.WriteLine("Main2");
    }
}

```

Para compilar esta aplicación de forma que se utilice el método *Main1.Main* como punto de entrada, debería utilizar esta opción:

```
csc MultipleMain.cs /main:Main1
```

Si cambiamos la opción a `/main:Main2`, entonces utilizaría el método `Main2.Main`.

Obviamente, dado que C# distingue entre mayúsculas y minúsculas, hay que tener cuidado y utilizar también el nombre correcto de la clase en la opción. Además, intentar compilar una aplicación que consiste en múltiples clases con varios métodos `Main` definidos y sin especificar la opción `/main` producirá un error de compilación.

CONSTRUCTORES

Una de las ventajas más grandes de un lenguaje de POO como C# es que se pueden definir métodos especiales que se invocan siempre cada vez que se crea una instancia de clase. Estos métodos se llaman *constructores*. C# presenta un nuevo tipo de constructor llamado *constructor estático*, que veremos en la próxima sección, «Miembros estáticos y miembros de instancia».

Un beneficio fundamental al utilizar un constructor es que garantiza que el objeto se inicialice adecuadamente antes de ser utilizado. Cuando un usuario instancia un objeto, se invoca el constructor de ese objeto, y éste debe terminar antes de que el usuario pueda llevar a cabo cualquier otro trabajo con ese objeto. Esta garantía es la que ayuda a asegurar la integridad del objeto y a hacer que las aplicaciones escritas con lenguajes orientados a objetos sean mucho más fiables.

Pero ¿cómo nombrar a un constructor para que el compilador sepa invocarlo cuando se instancia el objeto? Los diseñadores de C# siguieron las reglas de Stroustrup y dictaminaron que los constructores en C# deben tener el mismo nombre que la propia clase. He aquí una clase sencilla con un constructor igualmente sencillo:

```
using System;

class Constructor1App
{
    Constructor1App()
    {
        Console.WriteLine("Estamos en el constructor");
    }

    public static void Main()
    {
        Constructor1App app = new Constructor1App();
    }
}
```

Los constructores no devuelven valores. Si se intenta prefijar el constructor con un tipo, el compilador emitirá un error indicando que no pueden definirse miembros con los mismos nombres que el tipo en el que se incluyen.

Debería tener en cuenta también la manera en que los objetos se instancian en C#. Esto se hace utilizando la palabra reservada `new` con la siguiente sintaxis:

```
<clase> <objeto> = new <clase>(parámetros del constructor)
```

Si usted viene de un entorno C++, ponga especial atención a esto. En C++ puede instanciar un objeto de dos maneras. Puede declararlo en la pila, de la siguiente forma:

```
//Código C++. Esto crea una instancia de MyClass en la pila.
MyClass myClass;
```

O puede instanciar el objeto en la memoria libre (o montón) utilizando la palabra reservada de C++ *new*:

```
//Código C++. Crea una instancia de MyClass en el montón.
MyClass myClass = new MyClass();
```

La instanciación de objetos en C# es diferente, y esto es una causa de confusión para los nuevos desarrolladores de C#. La confusión surge del hecho de que ambos lenguajes comparten una palabra reservada común para crear objetos. Aunque utilizar la palabra reservada *new* en C++ le permite decidir dónde se crea un objeto, en C# depende del tipo que se está instanciando. Como vimos en el Capítulo 4, los tipos referencia se crean en el montón y los tipos valor se crean en la pila. Por lo tanto, la palabra reservada *new* le permite crear una nueva instancia de una clase, pero no determina dónde se crea el objeto.

Dicho esto, el siguiente código es válido en C#, pero si usted es desarrollador C++, no hace lo que podría pensar:

```
MyClass myClass;
```

En C++, esto crearía una instancia de *MyClass* en la pila. Como ya hemos mencionado, sólo puede crear objetos en C# utilizando la palabra reservada *new*. Por lo tanto, esta línea de código en C# simplemente declara que *myClass* es una variable del tipo *MyClass*, pero no instancia el objeto.

Como ejemplo de esto, si compila el siguiente programa, el compilador C# le advertirá de que la variable se ha declarado pero nunca se utiliza en la aplicación:

```
using System;

class Constructor2App
{
    Constructor2App()
    {
        Console.WriteLine("Estamos en el constructor");
    }

    public static void Main()
    {
        Constructor2App app;
    }
}
```

Por lo tanto, si declara algún tipo de objeto, necesita instanciarlo en algún lugar del código utilizando la palabra reservada *new*:

```
Constructor2App app;
app = new Constructor2App();
```

¿Por qué debería declarar un objeto sin instanciarlo? Declarar objetos antes de utilizarlos —o utilizar *new*— se hace en casos en los que se está declarando una clase dentro de otra. Este anidamiento de clases se llama *contención* o *agregación*.

Miembros estáticos y miembros de instancia

Como sucede en C++, se puede definir un miembro de clase como *miembro estático* o *miembro de instancia*. Por defecto, cada miembro se define como miembro de instancia, lo que significa que se hace una copia de ese miembro para cada instancia de la clase. Cuando un miembro se declara miembro estático, sólo hay una copia del miembro. Un miembro estático se crea cuando se carga la aplicación que contiene la clase, y existe durante toda la vida de la aplicación. Por lo tanto, se puede acceder al miembro incluso antes de que la clase se haya instanciado. Pero ¿por qué se querría hacer esto?

Un ejemplo sería el método *Main*. El Common Language Runtime (CLR) necesita tener un punto de entrada común a la aplicación. Para que el CLR no tenga que instanciar uno de sus objetos, la regla es definir un método estático llamado *Main* en una de las clases. También utilizará miembros estáticos cuando tenga un método que, desde una perspectiva orientada a objetos, pertenezca a una clase en términos semánticos pero no necesite un objeto real; por ejemplo, si quiere estar al tanto de cuántas instancias de un objeto dado se crean durante el tiempo de vida de una aplicación. Como los miembros estáticos son comunes a todas las instancias de los objetos, el siguiente ejemplo resolvería este problema:

```
using System;

class InstCount
{
    public InstCount()
    {
        instanceCount++;
    }

    static public int instanceCount = 0;
}

class AppClass
{
    public static void Main()
    {
        Console.WriteLine(InstCount.instanceCount);

        InstCount ic1 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);

        InstCount ic2 = new InstCount();
        Console.WriteLine(InstCount.instanceCount);
    }
}
```

En este ejemplo, la salida sería la siguiente:

```
0
1
2
```

Un último apunte sobre los miembros estáticos: un miembro estático debe tener un valor válido. Se puede especificar este valor cuando se define el miembro, como sigue:

```
static public int instanceCount1 = 10;
```

Si no inicializa la variable, el CLR lo hará por usted al iniciarse la aplicación utilizando por defecto el valor 0. Por lo tanto, las siguientes dos líneas son equivalentes:

```
static public int instanceCount2;
static public int instanceCount2 = 0;
```

Inicializadores de constructor

Todos los constructores de objetos de C# —con la excepción de los constructores de *System.Object*— incluyen una invocación al constructor de la clase base inmediatamente antes de la ejecución de la primera línea del constructor. Estos inicializadores de constructor le permiten especificar exactamente qué clase y qué constructor invocar. Esto se hace de dos formas:

- Un inicializador de la forma *base(...)* permite que se invoque al constructor de la clase base de la clase actual —esto es, el constructor cuya signatura coincide con la del constructor invocado.
- Un inicializador que toma la forma *this(...)* permite a la clase actual invocar otro constructor definido en la misma clase. Esto es útil cuando tiene múltiples constructores sobrecargados y quiere asegurarse de que siempre se invoca a un constructor predeterminado. Los métodos sobrecargados se tratarán en el Capítulo 6, pero damos ahora una rápida y sencilla definición: los métodos sobrecargados son dos o más métodos con el mismo nombre pero con diferentes listas de parámetros.

Para ver el orden de invocación en acción, observe que el siguiente código ejecutará el constructor para la clase A primero y después el constructor para la clase B:

```
using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}
```

```

class B : A
{
    public B()
    {
        Console.WriteLine("B");
    }
}

class DefaultInitializerApp
{
    public static void Main()
    {
        B b = new B();
    }
}

```

Este código es el equivalente funcional al siguiente código en el que el constructor de la clase base se invoca explícitamente:

```

using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }
}

class B : A
{
    public B() : base()
    {
        Console.WriteLine("B");
    }
}

class BaseDefaultInitializerApp
{
    public static void Main()
    {
        B b = new B();
    }
}

```

Echemos un vistazo a un ejemplo aún mejor que explica cuándo son útiles los inicializadores de constructor. Una vez más, tenemos dos clases: *A* y *B*. Esta vez, la clase *A* tiene dos constructores, uno que no tiene parámetros y otro que toma un *int*. La clase *B* tiene un constructor que toma un *int*. El problema viene en la construcción de la clase *B*. Si se ejecuta el siguiente código, se invocará el constructor de la clase *A* que no tiene argumentos:

```

using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }

    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

class DerivedInitializer1App
{
    public static void Main()
    {
        B b = new B(42);
    }
}

```

Por eso, ¿cómo me puedo asegurar que se invocará el constructor deseado de la clase A? Diciendo explícitamente al compilador qué constructor queremos invocar mediante el inicializador; así:

```

using System;

class A
{
    public A()
    {
        Console.WriteLine("A");
    }

    public A(int foo)
    {
        Console.WriteLine("A = {0}", foo);
    }
}

class B : A
{
    public B(int foo) : base(foo)
    {
        Console.WriteLine("B = {0}", foo);
    }
}

```

```
class DerivedInitializer2App
{
    public static void Main()
    {
        B b = new B(42);
    }
}
```

NOTA Al contrario que en Visual C++, no puede utilizar inicializadores de constructor para acceder a miembros de instancia, aparte de los propios constructores de la clase actual.

CONSTANTES FRENTE A CAMPOS DE SÓLO LECTURA

Seguramente, habrá veces en las que se tengan campos que no se quieran alterar durante la ejecución de la aplicación; por ejemplo, archivos de datos de los que depende una aplicación, el valor de *pi* en una clase de funciones matemáticas, o cualquier valor que se utilice en una aplicación, del que se sepa que no va a cambiar. Para solucionar estas necesidades, C# permite por definición dos tipos de miembros estrechamente relacionados: constantes y campos de sólo lectura.

Constantes

Como se puede adivinar por el nombre, las *constantes* —representadas por la palabra reservada *const*— son campos que permanecen constantes durante la vida de la aplicación. Sólo hay dos reglas en cuenta cuando se define que algo va a ser *const*. Primero, una constante es un miembro cuyo valor se establece en tiempo de compilación, tanto por el programador o en su defecto por el compilador. Segundo, el valor de un miembro constante se debe escribir en forma de literal.

Para definir un campo como constante, hay que especificar la palabra reservada *const* antes de definir el miembro, de la siguiente forma:

```
using System;

class MagicNumbers
{
    public const double pi = 3.1415;
    public const int answerToAllLifesQuestions = 42;
}

class ConstApp
{
    public static void Main()
    {
        Console.WriteLine("pi = {0}, cualquier otra cosa = {1}",
            MagicNumbers.pi, MagicNumbers.answerToAllLifesQuestions);
    }
}
```

Observe un punto clave en este código. No hace falta que el cliente instancie la clase *MagicNumbers*, ya que por defecto los miembros *const* son estáticos. Para aclararlo, eche un vistazo al MSIL que se generó para esos dos miembros:

```
answerToAllLifesQuestions : public static literal int32 = int32(0x0000002A)
pi : public static literal float64 = float64(3.1415000000000002)
```

Campos de sólo lectura

Un campo definido como *const* es útil porque documenta claramente la intención del programador de que el campo contenga un valor inmutable. Sin embargo, esto funciona solamente si conoce el valor en tiempo de compilación. Por eso, ¿qué hace un programador cuando surge la necesidad de un campo con un valor que no se conocerá hasta que estemos el tiempo de ejecución y que no debería cambiarse una vez se haya inicializado? Este asunto —normalmente sin resolver en otros lenguajes— fue resuelto por los diseñadores del lenguaje C# con *campos de sólo lectura*.

Cuando se define un campo con la palabra reservada *readonly*, se tiene la habilidad de establecer el valor de ese campo en un lugar: el constructor. Después de ese punto, el campo no se puede cambiar ni por la propia clase ni por los clientes de la clase. Digamos que quiere conocer en todo momento la resolución de pantalla de una aplicación gráfica. No puede solucionar este problema con un *const*, porque la aplicación no puede determinar la resolución de pantalla del usuario hasta que comience la ejecución; así que utilice código como el del siguiente ejemplo.

```
using System;

class GraphicsPackage
{
    public readonly int ScreenWidth;
    public readonly int ScreenHeight;

    public GraphicsPackage()
    {
        this.ScreenWidth = 1024;
        this.ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        GraphicsPackage graphics = new GraphicsPackage();
        Console.WriteLine("Anchura = {0}, Altura = {1}",
            graphics.ScreenWidth,
            graphics.ScreenHeight);
    }
}
```

A primera vista, este código parece exactamente lo que necesitamos. Sin embargo, hay un pequeño problema: los campos de sólo lectura que definimos son campos de instancia, lo que significa que el usuario tendría que instanciar la clase para utilizarlos. Esto podría no ser un problema, y podría ser incluso lo que quiere en casos en los que la forma de instanciar la clase determine el valor del campo de sólo lectura. Pero, ¿y si lo que se quiere es una constante, que es estática por definición, que pueda inicializarse en tiempo de ejecución? En ese caso se definiría el campo con los modificadores *static* y *readonly*. Después se crearía un tipo especial de constructor llamado *constructor estático*. Los constructores estáticos son constructores que se utilizan para inicializar campos estáticos, de sólo lectura o de otro tipo. Hemos modificado el ejemplo anterior para hacer los campos de resolución de la pantalla estáticos y de sólo lectura, y se ha añadido un constructor estático. Observe que se ha antepuesto la palabra reservada *static* a la definición del constructor:

```
using System;
class GraphicsPackage
{
    public static readonly int ScreenWidth;
    public static readonly int ScreenHeight;

    static GraphicsPackage()
    {
        //El código para calcular
        //la resolución estaría aquí.
        ScreenWidth = 1024;
        ScreenHeight = 768;
    }
}

class ReadOnlyApp
{
    public static void Main()
    {
        Console.WriteLine("Anchura = {0}, Altura = {1}",
            GraphicsPackage.ScreenWidth,
            GraphicsPackage.ScreenHeight);
    }
}
```

LIMPIEZA DE OBJETOS Y GESTIÓN DE RECURSOS

Una de las características más importantes de un sistema basado en componentes es su capacidad para proporcionar limpieza y liberación de recursos basada en la terminación de componentes. Por «limpieza y liberación de recursos» queremos decir la liberación puntual de referencias a otros componentes, así como a la liberación de recursos escasos o limitados sobre los cuales podría haber conflictos para su consecución, como conexiones a base de datos, manejadores de archivos y puertos de comunicaciones. Por «terminación» nos referimos al momento en el que el objeto ya no se utiliza.

En C++, la limpieza es directa porque se hace en el *destructor* del objeto, una función definida por cada objeto de C++ que se ejecuta automáticamente cuando el objeto sale de ámbito. En el mundo de Microsoft .NET, la limpieza de objetos se gestiona automáticamente mediante el recolector de basura de .NET (Garbage Collector=GC). Esta estrategia es algo polémica, ya que, en contraste con el control existente en C++, donde se conoce cuando se ejecuta el código de terminación de un objeto, la solución .NET está basada en un modelo «perezoso». El GC utiliza hilos de ejecución o subprocesos que operan en segundo plano para determinar cuándo se ha dejado de hacer referencia a un objeto. Entonces, otros hilos del GC se responsabilizan de ejecutar el código de terminación de ese objeto. Esto funciona bien en la mayoría de las situaciones, pero es una solución poco óptima cuando se trata de recursos que necesitan liberarse de forma puntual y en un orden predecible. Como ya se habrá dado cuenta, no es un problema fácil de resolver. En esta sección, resolveremos los problemas de terminación de objetos y de gestión de recursos y su papel en la creación de objetos con tiempos de vida predecibles. (Observe que esta sección está totalmente basada en una excelente explicación de la gestión de recursos de C# presentada por Brian Harry, miembro del equipo de Microsoft's .NET, en un foro público en línea. Agradezco a Brian el haberme permitido utilizar su explicación).

Un poquito de historia

Hace algunos años, cuando empezó el proyecto de .NET, se desató un debate masivo sobre el tema de la gestión de recursos. Los primeros contribuyentes de .NET vinieron de COM y de los equipos de Microsoft Visual Basic, así como de otros muchos equipos de Microsoft. Uno de los grandes problemas a los que estos equipos plantaron cara fue a cómo manejar los problemas relativos al conteo de referencias, incluyendo ciclos y los errores debidos a una utilización incorrecta. Un ejemplo de estos problema es la referencia circular: un objeto contiene una referencia a otro objeto, que a su vez contiene una referencia al primer objeto. Cuándo y cómo se libera una referencia circular puede causar problemas. No liberar una o ambas referencias da como resultado un fallo de memoria (*memory leak*), que es extremadamente difícil de localizar. Y cualquiera que haya trabajado bastante con COM puede contar historias sobre el tiempo perdido para localizar errores que resultan de los problemas del conteo de referencias. Microsoft, dándose cuenta de que los problemas de dicho conteo son bastante comunes en aplicaciones basadas en componentes (como las aplicaciones COM), empezó a proporcionar una solución uniforme en .NET.

La solución inicial al problema estaba basada en el conteo automático de referencias —para que no tuviera importancia el hecho de que el programador hubiera olvidado contar las referencias—, junto con métodos para detectar y manejar ciclos automáticamente. Además, el equipo de .NET pensó en añadir recolección de referencias conservativas, recolección basada en trazas y algoritmos de GC que pudieran recolectar un solo objeto sin tener que hacer la traza completa del grafo. Sin embargo, por varias razones —trataremos esos diferentes enfoques en breve—, se concluyó que estas soluciones no iban a funcionar en el caso típico.

Un obstáculo importante fue que en los primeros días del desarrollo de .NET se buscaba como objetivo primario el mantener un alto grado de compatibilidad con Visual Basic. Por lo tanto, la solución tenía que ser completa y transparente, sin cambios semánticos al propio lenguaje Visual Basic. Después de muchos debates, la solución final fue un modelo basado en contexto, donde todo lo que existiera en un contexto determinista utilizaría un conteo de referencias montado sobre el GC, y todo lo que no estuviera en este contexto utilizaría sólo el GC. Esto ayudó a evitar algunos de los problemas de bifurcación (descritos más adelante), pero no daba una solución buena para una mezcla con un nivel de detalle mayor de código entre lenguajes. Por eso, la decisión fue hacer una serie de cambios en el lenguaje Visual Basic para «modernizarlo» y hacerlo más eficaz. Parte de esa decisión fue el abandono de los requisitos de tiempo de vida de Visual Basic. Esta decisión también acabó en términos generales con la investigación de los problemas de tiempo de vida determinista.

Finalización determinista

Después de salirnos del camino con un poco de historia, vamos a definir la finalización determinista. Una vez que se determina que un objeto no se va a utilizar más, se ejecuta su código de terminación y se liberan las referencias que tiene hacia otros objetos. Entonces, este proceso de terminación recorre en cascada el grafo de objetos, empezando por objeto superior del grafo. Esto funcionaría, idealmente, tanto para objetos cuyo uso es compartido como para los que se utilizan de forma exclusiva.

Observe que aquí no hay promesas relacionadas con el tiempo. Una vez que un hilo del GC descubre que una referencia ya no se utiliza más, ese hilo particular no hará nada más hasta que se ejecute el código de terminación del objeto. Sin embargo, se puede producir un cambio de contexto en la UCP durante el proceso en cualquier momento, lo que significa que podría transcurrir una cantidad arbitraria de tiempo —desde la perspectiva de la aplicación— antes de que se complete este paso. Como ya se ha mencionado, existen muchas situaciones en las que una aplicación se puede preocupar por la temporalidad u orden de ejecución con relación al código de terminación. Estas situaciones van ligadas, generalmente, a los recursos para los que hay un alto grado de solicitudes de cara a conseguirlos. He aquí algunos ejemplos de recursos que un objeto podría tener que liberar tan pronto como dejara de estar activo o en uso:

- **Memoria.** Liberar la memoria del grafo de un objeto permite que ésta pueda reutilizarse rápidamente.
- **Manejadores de ventana.** La memoria consumida por un objeto de ventana en el GC no refleja el coste real. Hay memoria reservada dentro del sistema operativo para representar la ventana, y podría incluso haber un límite, además de la memoria disponible, con relación al número total de manejadores de ventana para los que se puede reservar memoria.
- **Conexiones de datos.** Las conexiones de datos concurrentes a menudo tienen licencia, y por lo tanto podría haber un número limitado de ellas disponible. Es

importante que se devuelvan inmediatamente a un conjunto de conexiones libres para que se puedan reutilizar.

- **Archivos.** Como hay una sola instancia de un archivo dado y se requiere un acceso exclusivo para muchas operaciones, es importante que los manejadores del archivo se cierren cuando no se utilicen.

Recolección basada en conteo de referencias

El conteo de referencias hace un trabajo razonable al proporcionar finalización determinista en muchos casos. Sin embargo, merece la pena observar que en algunos casos no ocurre así, siendo los ciclos el ejemplo más comúnmente citado. De hecho, el conteo de referencias directo *nunca recolecta objetos que participan en ciclos*. La mayoría de nosotros hemos aprendido de forma dolorosa técnicas manuales para gestionarlo, pero utilizarlas es una fuente indeseable de errores. También, si comienza a crear objetos remotos mediante apartamentos, se puede conseguir código reentrante al apartamento y así introducir mucho indeterminismo en su programa. Algunos argumentarían que en el momento en que manejemos una referencia a un objeto fuera del control inmediato de un programa fuertemente acoplado, hemos perdido la finalización determinista, porque no tenemos ni idea cuándo o si ese código «extranjero» liberará la referencia. Otros creen que un sistema complejo dependiente del orden de terminación de un grafo de objetos complejo es un diseño inherentemente frágil, con posibilidad de crear problemas de mantenimiento significativos según evolucione el código.

Recolección basada en trazas

Un recolector basado en trazas hace algunas promesas menos importantes que las que hace el conteo de referencias. Es un sistema algo más perezoso con respecto al código de terminación que se ejecuta. Los objetos tienen *finalizadores*, métodos que son ejecutados cuando el programa ya no puede acceder al objeto. Las trazas tienen la ventaja de que los ciclos no son un problema, y sobre todo que el asignar una referencia es una sencilla operación de movimiento; veremos más de esto en breve. El precio que se paga es que no hay promesas asociadas a que se ejecute «inmediatamente» el código de terminación después de que una referencia deje de utilizarse. Entonces, ¿qué se promete? La verdad es que para los programas que se comportan adecuadamente —un programa que no se comporta adecuadamente es uno que falla o que pone al hilo de ejecución finalizador en un bucle infinito—, se invocan los finalizadores para los objetos. La documentación en línea tiende a ser demasiado cauta sobre las promesas a este respecto, pero si un objeto tiene un método *Finalize*, el sistema lo invocará. Esto no soluciona el problema de la finalización determinista, pero es importante entender que los recursos se recuperarán y que los finalizadores son una manera efectiva de prevenir el quedarnos sin recursos por una mala gestión en un programa.

Rendimiento

El rendimiento, al estar relacionado con *el problema de finalización*, es un problema importante. El equipo de .NET cree que debe existir algún tipo de recolector basado en trazas para manejar los ciclos, lo que demanda gran parte del coste de un recolector de trazas. También se cree que el rendimiento de la ejecución del código se puede ver sustancialmente afectado por el coste del conteo de referencias. La buena noticia es que en el contexto de todos los objetos reservados en la ejecución de un programa, el número de esos objetos que necesitan realmente finalización determinista es pequeño. Sin embargo, es difícil aislar normalmente el coste del rendimiento para sólo esos objetos. Echemos un vistazo al pseudocódigo de una asignación de referencias simple cuando se utiliza un recolector basado en trazas y después al pseudocódigo cuando se utiliza la cuenta de referencia:

```
//Trazas.  
a = b;
```

Eso es. El compilador lo convierte en una instrucción de movimiento simple y podría incluso optimizar todo en algunas circunstancias.

```
//Cuenta de referencias.  
if (a != null)  
    if (InterlockedDecrement(ref a.m_ref) == 0)  
        a.FinalRelease();  
  
if (b != null)  
    InterlockedIncrement(ref b.m_ref);  
  
a = b;
```

El incremento en este código es muy alto; el conjunto sobre el que trabajamos (*working set*) es mayor y el rendimiento de ejecución es obscuramente más alto, especialmente cuando se dan las dos instrucciones interbloqueadas. Se puede limitar el incremento del código poniéndolo en un método «auxiliar», y más aún, incrementando la longitud de la ruta de acceso del código. Además, la generación del código sufrirá finalmente cuando se pongan todos los bloques *try* necesarios, ya que el optimizador tiene en sus manos algo entrelazado en presencia del código de manejo de excepciones; esto es cierto incluso en C++ sin gestionar. También merece la pena observar que todos los objetos son 4 bytes mayores a causa del campo extra del conteo de referencias, incrementando otra vez el uso de la memoria y el tamaño del conjunto sobre el que trabajamos.

Ahora siguen dos ejemplos de aplicaciones que demuestran el coste de esto (son cortesía del equipo .NET). Esta prueba particular se repite, reservando memoria para los objetos, haciendo dos asignaciones y saliendo del ámbito de una referencia. Como ocurre con cualquier prueba, está abierta a algunas interpretaciones subjetivas. Uno podría incluso argumentar que en el contexto de esta rutina, la mayor parte del conteo de referencias se puede optimizar. Esto probablemente sea verdad; sin embargo, aquí se intenta demostrar el efecto. En un programa real, esos tipos de optimización son difíciles de

realizar, si no imposible. De hecho, los programadores de C++ hacen esas optimizaciones manualmente, lo que lleva a errores del conteo de referencias. Por lo tanto, observe que en un programa real la proporción de asignación frente a la reserva de memoria es mucho más alta que aquí.

He aquí el primer ejemplo, `ref_gc.cs` —esta versión se basa en el GC basado en trazas:

```
using System;

public class Foo
{
    private static Foo m_f;
    private int m_member;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;
        for (int i = 0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            //Asignar la estática a f2.
            f2 = m_f;

            //Asignar f a la estática.
            m_f = f;

            //f sale de ámbito.
        }

        //Asignar f2 a la estática.
        m_f = f2;

        //f2 sale de ámbito.

        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Marcas = {0}", ticks);
    }

    public Foo()
    {
    }
}
```

Y aquí está el segundo, `ref_rm.cs` —una versión de conteo de referencias que utiliza operaciones de interbloqueo para mantener la seguridad del hilo de ejecución:

```
using System;
using System.Threading;

public class Foo
```

```

{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();

            //Asignar estática a f2.
            if (f2 != null)
            {
                if (Interlocked.Decrement(ref f2.m_ref) == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                Interlocked.Increment(ref m_f.m_ref);
            f2 = m_f;

            //Asignar f a la estática.
            if (m_f != null)
            {
                if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                    m_f.Dispose();
            }
            if (f != null)
                Interlocked.Increment(ref f.m_ref);
            m_f = f;

            //f sale de ámbito.
            if (Interlocked.Decrement(ref f.m_ref) == 0)
                f.Dispose();
        }

        //Asignar f2 a la estática.
        if (m_f != null)
        {
            if (Interlocked.Decrement(ref m_f.m_ref) == 0)
                m_f.Dispose();
        }
        if (f2 != null)
            Interlocked.Increment(ref f2.m_ref);
        m_f = f2;

        //f2 sale de ámbito.
        if (Interlocked.Decrement(ref f2.m_ref) == 0)
            f2.Dispose();
        ticks = Environment.TickCount - ticks;
        Console.WriteLine("Marcas = {0}", ticks);
    }
}

```

```

    public Foo()
    {
        m_ref = 1;
    }

    public virtual void Dispose()
    {
    }
}

```

Observe que sólo hay un hilo de ejecución y por lo tanto no hay conflictos por conseguir el bus, haciendo de éste el caso «ideal». Probablemente se podría poner un poco a punto, pero no demasiado. También merece la pena observar que Visual Basic históricamente no ha tenido que preocuparse por utilizar operaciones interbloqueadas para su conteo de referencias (aunque Visual C++ sí). En versiones anteriores de Visual Basic, un componente se ejecutaba en un apartamento monoproceso y se garantizaba que en él, en cada momento, sólo se ejecutaría simultáneamente un hilo de ejecución. Una de las metas para Visual Basic.NET es la programación multihilo y otra es deshacerse de la complejidad de los modelos multihilo de COM. Sin embargo, para los que quieran ver una versión que no utilice prefijos de bloqueo, sigue otro ejemplo, otra vez cortesía del equipo .NET: `ref_rs.cs`, una versión de conteo de referencias que asume que se está ejecutando en un entorno de un único hilo de ejecución. Este programa no es tan lento como la versión multihilo, pero es todavía un poquito más lento que la versión GC.

```

using System;

public class Foo
{
    private static Foo m_f;
    private int m_member;
    private int m_ref;

    public static void Main(String[] args)
    {
        int ticks = Environment.TickCount;

        Foo f2 = null;

        for (int i=0; i < 10000000; ++i)
        {
            Foo f = new Foo();
            //Asignar estática a f2.
            if (f2 != null)
            {
                if (--f2.m_ref == 0)
                    f2.Dispose();
            }
            if (m_f != null)
                ++m_f.m_ref;
            f2 = m_f;

            //Asignar f a la estática.
            if (m_f != null)

```

```

        {
            if (--m_f.m_ref == 0)
                m_f.Dispose();
        }
        if (f != null)
            ++f.m_ref;
        m_f = f;

        //f sale de ámbito.
        if (--f.m_ref == 0)
            f.Dispose();
    }

    //Asignar f2 a la estática.
    if (m_f != null)
    {
        if (--m_f.m_ref == 0)
            m_f.Dispose();
    }
    if (f2 != null)
        ++f2.m_ref;
    m_f = f2;

    //f2 sale de ámbito.
    if (--f2.m_ref == 0)
        f2.Dispose();

    ticks = Environment.TickCount - ticks;
    Console.WriteLine("Marcas = {0}", ticks);
}

public Foo()
{
    m_ref = 1;
}

public virtual void Dispose()
{
}
}

```

Obviamente, aquí hay muchas variables. Aun así, ejecutando las tres aplicaciones, resulta que la versión GC se ejecuta casi dos veces más rápido que el ejemplo de conteo de referencias de un solo hilo de ejecución y cuatro veces más rápido que el ejemplo de conteo de referencias con prefijos de bloqueo. Mi experiencia personal fue como sigue —observe que estos números representan la media de ejecución en un IBM ThinkPad 570 ejecutando el compilador de .NET Framework SDK Beta 1:

GC Version (ref_gc)	1162 ms
Ref Counting (ref_rm) (multi-threaded)	4757 ms
Ref Counting (ref_rs) (single-threaded)	1913 ms

La solución perfecta

La mayoría estaría de acuerdo en que la solución perfecta a este problema sería un sistema en el que cada objeto se alojara en memoria, se utilizara y se liberara con un coste barato. En ese sistema ideal todos los objetos se liberarían de forma determinista y ordenada en el instante en que el programador creyera que el objeto ya no se fuese a necesitar más (sin tener en cuenta si existen ciclos). Después de haber invertido incontables horas en resolver este problema, el equipo .NET cree que la única forma de cumplir esto es combinar un GC basado en trazas con cuenta de referencias. Los datos de las pruebas reflejan que la cuenta de referencias es demasiado cara para utilizarse de modo general con todos los objetos del entorno de programación. Las rutas de acceso al código son más largas y el código y los datos del conjunto con que trabajamos son mayores. Entonces, si se combina este ya de por sí alto precio con el coste adicional de implementar un recolector basado en trazas para liberar ciclos, se está pagando un precio desorbitante por la gestión de la memoria.

Se debería observar que ya en los primeros días del desarrollo .NET se investigaron varias técnicas para encontrar una manera de mejorar el rendimiento de la cuenta de referencias. Anteriormente han existido sistemas con un rendimiento razonablemente alto contruidos en base al conteo de referencias. Desgraciadamente, después de examinar la literatura, se determinó que tales sistemas obtenían un rendimiento mejor si se dejaba de lado algo de determinismo.

Como cosa aparte, merece la pena observar que los programas C++/COM no sufren este problema. La mayoría de los programas C++ de alto rendimiento que utilizan COM usan internamente clases de C++ para las que el programador ha de gestionar explícitamente la memoria. Los programadores de C++ generalmente utilizan COM sólo para crear las interfaces para sus clientes. Esta es una característica fundamental que permite a esos programas ejecutarse con un rendimiento adecuado. Sin embargo, obviamente, esta no es la meta de .NET, donde se intenta que los problemas de gestión de memoria los controle el GC y no el programador.

La solución (casi) perfecta

Así, como con la mayoría de las cosas en la vida, no se puede tener todo lo relacionado con la solución de gestión de recursos. Pero, ¿y si se pudiera tener finalización determinista sólo en los objetos en los que se necesitara? El equipo .NET pasó mucho tiempo pensándolo. Hay que recordar que esto estaba en el contexto de duplicar exactamente la semántica de Visual Basic 6 en el nuevo sistema. La mayoría de ese análisis todavía tiene sentido, pero algunas ideas que se descartaron hace mucho tiempo, ahora parecen más atractivas como técnicas de gestión de recursos que como semántica transparente del tiempo de vida de Visual Basic 6.

El primer intento fue simplemente marcar una clase de forma que requiriese finalización determinista, bien mediante atributo bien mediante herencia de una clase «especial». Esto haría que se contaran las referencias al objeto. Se investigaron muchos diseños diferentes, que incluían tanto subclases de *System.Object* como cambiar la raíz de la

jerarquía clases por alguna otra clase que enlazara el mundo del conteo de referencias con el mundo de no realizar dicho conteo. Desgraciadamente, había una serie de problemas que no se podían evitar, como se describe en las siguientes secciones.

Composición

En cualquier momento en que se tome un objeto que requiera finalización determinista y se almacene en un objeto que no la requiera, se ha perdido el determinismo (ya que el determinismo no es transitivo). El problema es que esto llega al corazón de la jerarquía de clases. Por ejemplo, ¿qué pasa con los arrays? Si se quieren arrays de objetos deterministas, los arrays necesitan ser deterministas. Y ¿qué pasa con las colecciones y las tablas de dispersión (*hash*)? La lista continúa —antes de que nos demos cuenta, se contarán las referencias de toda la biblioteca de clases para conseguir así nuestro propósito.

Otra alternativa era bifurcar (o dividir en dos ramas) la biblioteca de clases de .NET Framework y tener dos versiones de muchos tipos de clases; por ejemplo, los arrays deterministas y los no deterministas. Esto se consideró seriamente. Sin embargo, la decisión final fue que tener dos copias del framework entero sería confuso, su rendimiento sería muy malo —se cargarían dos copias de cada clase— y al final no sería práctico.

Se examinaron soluciones específicas para clases específicas, pero ninguna de éstas se acercó a ser una solución general viable para todo el framework. Al ser esto el problema fundamental —si un objeto no determinista contiene una referencia a un objeto determinista, el sistema es no determinista—, se consideró también que fuera un error, pero se concluyó con que esta restricción haría difícil escribir programas reales.

Conversión de tipo

De alguna forma, la conversión de tipo es un problema. Hay que considerar estas preguntas: ¿se puede convertir el tipo de un objeto determinista a un *System.Object*? Si es así, ¿hay que contar la referencia? Si la respuesta es sí, hay que contar la referencia de todo. Si la respuesta es no, el objeto pierde el determinismo. Si la respuesta es «error», se ha violado la premisa fundamental de que *System.Object* es la raíz de la jerarquía de objetos.

Interfaces

A estas alturas, se debería dar cuenta de lo complejo que es este asunto. Si un objeto determinista implementa interfaces, ¿el tipo de la referencia se cuenta como referencia a interfaz? Si la respuesta es sí, la referencia cuenta todos los objetos que implementan las interfaces. (Observe que *System.Int32* implementa interfaces). Si la respuesta es no, otra vez el objeto pierde el determinismo. Si la respuesta es «error», los objetos deterministas no pueden implementar las interfaces. Si la respuesta es «depende de si la interfaz está marcada como determinista», hay otro problema de bifurcación. Se supone que las inter-

faces no dictan la semántica del tiempo de vida de los objetos. ¿Qué pasaría si alguien implementase una API que tiene una interfaz *ICollection*, y el objeto que la implementa necesitase determinismo, pero la interfaz no se definió de esa manera? No tenemos suerte. En este escenario, necesitaríamos definir dos interfaces —una determinista y otra no determinista— implementando cada método dos veces. Lo crea o no, también se consideraron mecanismos para generar automáticamente las dos versiones de los métodos. Esa línea de pensamiento se hundió profundamente en una complejidad inmensa y se abandonó la idea.

El patrón de diseño Liberar

¿Dónde nos deja todo esto? Actualmente, parecería que se nos deja con la finalización determinista que funciona de la siguiente manera:

- El GC sigue la pista de a qué objetos se hace referencia.
- El GC siempre tiene un hilo de ejecución de baja prioridad analizando los objetos para determinar cuándo no se hace referencia a alguno.
- Un segundo hilo de ejecución de baja prioridad es responsable de la limpieza. Este hilo de ejecución invoca al método *Finalize* del objeto.

Este enfoque resuelve el problema de referencia circular, pero causa otros problemas. Por ejemplo, ¡no hay garantía de que el entorno de ejecución invoque el método *Finalize*! Esto es lo que se conoce por *determinista* en la finalización determinista. Así surge la pregunta: «¿Cómo puedo hacer la limpieza necesaria y saber que siempre se invoca al código de limpieza?».

En el caso de la liberación de recursos escasos, Microsoft propone una solución basada en el patrón de diseño Liberar. Este patrón de diseño recomienda que el objeto posea un método público, con un nombre genérico como *Cleanup* (Limpieza) o *Dispose* (Liberación), y que el usuario sepa cómo invocarlo cuando termine de utilizar el objeto. Entonces está en manos del diseñador de la clase hacer cualquier limpieza necesaria en ese método. De hecho, se puede ver que muchas de las clases de la biblioteca de clases de .NET Framework implementan un método *Dispose* para este propósito. Como ejemplo, la documentación para la clase *System.WinForms.TrayIcon* dice: «Invoque *Dispose* cuando haya terminado de utilizar *TrayIcon*».

HERENCIA

Como se mencionaba en el Capítulo 1, la herencia se utiliza cuando se construye una clase a partir de otra —en términos de datos o de comportamiento— y se adhiere a las reglas de sustitutibilidad—, es decir, que la clase derivada puede sustituirse por la clase base. Un ejemplo sería el caso en que estuviera escribiendo una jerarquía de clases de bases de datos. Digamos que quiere tener una clase para manejar las bases de datos Microsoft SQL Server y Oracle. Como estas bases de datos difieren en algunos aspectos,

querría tener una clase para cada base de datos. Sin embargo, las dos bases de datos comparten suficiente funcionalidad como para colocar la funcionalidad común en una clase base, derivar las otras dos clases de la clase base y redefinir o modificar el comportamiento heredado.

Para heredar una clase de otra, habría que utilizar la siguiente sintaxis:

```
clase <claseDerivada> : <claseBase>
```

El ejemplo de base de datos tendría la siguiente apariencia:

```
using System;

class Database
{
    public Database()
    {
        CommonField = 42;
    }

    public int CommonField;

    public void CommonMethod()
    {
        Console.WriteLine("Método Database.Common");
    }
}

class SQLServer : Database
{
    public void SomeMethodSpecificToSQLServer()
    {
        Console.WriteLine("SQLServer.SomeMethodSpecificToSQLServer");
    }
}

class Oracle : Database
{
    public void SomeMethodSpecificToOracle()
    {
        Console.WriteLine("Oracle.SomeMethodSpecificToOracle");
    }
}

class InheritanceApp
{
    public static void Main()
    {
        SQLServer sqlserver = new SQLServer();

        sqlserver.SomeMethodSpecificToSQLServer();
        sqlserver.CommonMethod();
        Console.WriteLine("Campo común heredado = {0}",
            sqlserver.CommonField);
    }
}
```

Si compilamos y ejecutamos esta aplicación, obtenemos el siguiente resultado:

```
SQLServer.SomeMethodSpecificToSQLServer
Método Database.Common
Campo común heredado = 42
```

Observe que los métodos *Database.CommonMethod* y *Database.CommonField* son ahora parte de la definición de la clase *SQLServer*. Como las clases *SQLServer* y la *Oracle* se derivan de la clase base *Database*, las dos heredan casi todos los miembros que se definen como *public*, *protected* o *internal*. La única excepción a esto es el constructor, que no puede heredarse. Cada clase debe implementar su propio constructor independientemente de su clase base.

La redefinición de métodos se trata en el Capítulo 6. Sin embargo, para que esta sección sea completa, mencionaremos que la redefinición de métodos le permite heredar un método de una clase base y cambiar después la implementación de éste. Las clases abstractas están muy relacionadas con la redefinición de métodos —éstas también se tratarán en el Capítulo 6.

Interfaces múltiples

Aclaremos esto, porque la herencia múltiple ha llegado a ser objeto de controversia en muchos grupos de noticias nuevos y listas de correo: *C# no* permite la herencia múltiple a través de la derivación. Sin embargo, se pueden agregar las características de comportamiento de múltiples entidades de programación implementando múltiples interfaces. En el Capítulo 9, «Interfaces», trataremos las interfaces y cómo trabajar con ellas. Por ahora, piense en las interfaces de C# como lo haría con una interfaz de COM.

Dicho esto, el siguiente programa no es válido:

```
class Foo
{
}

class Bar
{
}

class MITest : Foo, Bar
{
    public static void Main()
    {
    }
}
```

El error que se produce en este ejemplo tiene que ver con cómo se implementan las interfaces. Las interfaces que decide implementar se listan a continuación de la clase base de la clase. Por lo tanto, en este ejemplo, el compilador de C# piensa que *Bar* debería ser de tipo interfaz. Es por esto por lo que el compilador de C# dará el siguiente mensaje de error:

```
'Bar' : type in interface list is not an interface
```

El siguiente ejemplo, más realista, es perfectamente válido, porque la clase *MyFancyGrid* se deriva de *Control* e implementa las interfaces *ISerializable* e *IDataBound*:

```
class Control
{
}

interface ISerializable
{
}

interface IDataBound
{
}

class MyFancyGrid : Control, ISerializable, IDataBound
{
}
```

La clave es que la única manera de poder implementar algo como herencia múltiple en C# es a través de la utilización de interfaces.

Clases selladas

Si queremos asegurarnos de que una clase nunca se pueda utilizar como clase base, hay que utilizar el modificador *sealed* cuando se defina ésta. La única restricción es que una clase abstracta no puede utilizarse como clase sellada, porque, por su naturaleza, las clases abstractas se han de utilizar como clases base. Otro punto a resaltar es que aunque el propósito de una clase sellada es evitar que se derive de ella de forma involuntaria, se les permiten ciertas optimizaciones en tiempo de ejecución. Específicamente, dado que el compilador garantiza que la clase nunca puede tener clases derivadas, es posible transformar las invocaciones a funciones miembro virtuales para las instancias de clases selladas en invocaciones no virtuales. He aquí un ejemplo de cómo crear una clase sellada:

```
using System;

sealed class MyPoint
{
    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    private int X;
    public int x
    {
        get
```

```

        {
            return this.X;
        }
        set
        {
            this.X = value;
        }
    }
    private int Y;
    public int y
    {
        get
        {
            return this.Y;
        }
        set
        {
            this.Y = value;
        }
    }
}

class SealedApp
{
    public static void Main()
    {
        MyPoint pt = new MyPoint(6,16);
        Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
    }
}

```

Observe que se utilizó el modificador de acceso *private* en los miembros de clase interna X e Y. Si utilizamos el modificador *protected* se generaría un aviso del compilador, debido a que los miembros protegidos son visibles desde las clases derivadas, y, como ya sabemos, las clases selladas no tienen ninguna clase derivada.

RESUMEN

El concepto de clases y sus relaciones con los objetos son las bases para la programación basada en objetos. Las características orientadas a objetos de C# conforman una herencia transmitida desde C++ y que se ha moldeado y mejorado por las características de .NET Framework. En sistemas gestionados como el Common Language Runtime, la gestión de recursos es un asunto que preocupa mucho a los desarrolladores. El CLR se esfuerza por liberar a los programadores del trabajo arduo del conteo de referencias mediante la recolección de basura basada en la finalización determinista. También, la herencia se trata de manera diferente en C# que en C++. Aunque sólo se permite la herencia simple, los desarrolladores pueden aprovecharse aún de algunos de los beneficios de la herencia múltiple mediante la implementación de interfaces múltiples.

Capítulo 6

Métodos

Como aprendimos en el Capítulo 1, «Fundamentos de la programación orientada a objetos», las clases son agrupaciones de datos encapsulados y los métodos que trabajan sobre esos datos. Dicho de otra manera, los métodos dan a las clases sus características de comportamiento, y nombramos a los métodos basándonos en las acciones que queremos que lleven a cabo las clases en representación nuestra. Hasta ahora, no hemos entrado en asuntos demasiado específicos en cuanto a definir e invocar métodos en C#. De eso se ocupa este capítulo: descubriremos las palabras reservadas de los parámetros de los métodos *ref* y *out* y cómo no permiten definir un método de tal manera que se pueda devolver más de un único valor al invocador. También aprenderemos a definir métodos sobrecargados para que múltiples métodos con el mismo nombre puedan funcionar de forma distinta, dependiendo de los tipos y/o número de argumentos que se les han pasado. Después aprenderemos cómo manejar situaciones en las que no sabemos el número exacto de argumentos que tendrá un método hasta el tiempo de ejecución. Finalmente, acabaremos el capítulo tratando los métodos virtuales —basándonos en el tratamiento de la herencia del Capítulo 5, «Clases»— y cómo definir métodos estáticos.

PARÁMETROS *REF* Y *OUT* DE LOS MÉTODOS

Cuando intentamos recuperar información utilizando un método en C#, recibimos un único valor de retorno. Por lo tanto, a primera vista podría parecer que sólo se puede recuperar un valor de cada invocación a método. Obviamente, el invocar un método para cada dato necesario sería en muchas ocasiones engorroso. Por ejemplo, digamos que tenemos una clase *Color* que representa un determinado color con tres valores utilizando el modelo estándar RGB (rojo-verde-azul; red-green-blue en inglés) para describir los colores. Si utilizamos sólo valores de retorno, nos veríamos forzados a escribir código parecido al del siguiente ejemplo para recuperar los tres valores.

```
//Asumimos que color es una instancia de una clase Color.  
int red = color.GetRed();
```

```
int green = color.GetGreen();
int blue = color.GetBlue();
```

Pero lo que queremos es algo parecido a lo siguiente:

```
int red;
int green;
int blue;
color.GetRGB(red, green, blue);
```

Sin embargo, tenemos un problema. Cuando se invoca el método *color.GetRGB*, los valores para los argumentos *red*, *green* y *blue* se copian en la pila local del método y cualquier cambio que haga el método no se hará en las variables del invocador.

En C++, este problema se soluciona haciendo que el método invocado pase punteros o referencias a las variables para que el método funcione sobre los datos del invocador. La solución en C# es similar. Realmente, C# ofrece dos soluciones parecidas. La primera tiene que ver con la palabra reservada *ref*. Esta palabra reservada le dice al compilador de C# que los argumentos que se están pasando apuntan a la misma memoria que las variables en el código que invoca. De esa manera, si el método invocado modifica estos valores y a continuación vuelve, las variables del código que invoca se habrán modificado. El siguiente código ilustra cómo utilizar la palabra reservada *ref* con el ejemplo de la clase *Color*:

```
using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(ref int red, ref int green, ref int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class RefTest1App
{
    public static void Main()
    {
        Color color = new Color();
        int red;
        int green;
```

```

        int blue;
        color.GetColors(ref red, ref green, ref blue);
        Console.WriteLine("rojo = {0}, verde = {1}, azul = {2}",
                           red, green, blue);
    }
}

```

El hecho de utilizar la palabra reservada *ref* tiene una desventaja, y de hecho, a causa de esta limitación, el código anterior no se compilará. Cuando se utilice la palabra reservada *ref*, antes de invocar el método se deben inicializar los argumentos pasados. Por lo tanto, para que este código funcione, debe modificarse de esta manera:

```

using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(ref int red, ref int green, ref int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class RefTest2App
{
    public static void Main()
    {
        Color color = new Color();
        int red = 0;
        int green = 0;
        int blue = 0;
        color.GetColors(ref red, ref green, ref blue);
        Console.WriteLine("rojo = {0}, verde = {1}, azul = {2}",
                           red, green, blue);
    }
}

```

En este ejemplo, parece no tener sentido el inicializar las variables que se van a sobrescribir, ¿verdad? Por lo tanto, C# proporciona un mecanismo alternativo para pasar un argumento cuyo valor cambiado necesita ser visto por el código invocador: la palabra reservada *out*. Aquí tenemos el mismo ejemplo de la clase *Color* utilizando la palabra reservada *out*:

```

using System;

class Color
{
    public Color()
    {
        this.red = 255;
        this.green = 0;
        this.blue = 125;
    }

    protected int red;
    protected int green;
    protected int blue;

    public void GetColors(out int red, out int green, out int blue)
    {
        red = this.red;
        green = this.green;
        blue = this.blue;
    }
}

class OutTest1App
{
    public static void Main()
    {
        Color color = new Color();
        int red;
        int green;
        int blue;
        color.GetColors(out red, out green, out blue);
        Console.WriteLine("rojo = {0}, verde = {1}, azul = {2}",
            red, green, blue);
    }
}

```

La única diferencia entre la palabra reservada *ref* y la palabra reservada *out* es que la palabra reservada *out* no necesita que el código que invoca inicialice previamente los argumentos que se pasen. Entonces, ¿cuándo debe utilizarse la palabra reservada *ref*? Debería utilizarse cuando necesitemos asegurarnos de que el método que invoca ha inicializado el argumento. En los ejemplos anteriores podría utilizarse *out*, porque el método que se invoca no dependía del valor de la variable que se le pasaba. Pero ¿qué pasaría si el método invocado utilizara un valor del parámetro? Echemos un vistazo a este código:

```

using System;

class Window
{
    public Window(int x, int y)
    {
        this.x = x;
    }
}

```

```

        this.y = y;
    }

    protected int x;
    protected int y;

    public void Move(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void ChangePos(ref int x, ref int y)
    {
        this.x += x;
        this.y += y;

        x = this.x;
        y = this.y;
    }
}

class OutTest2App
{
    public static void Main()
    {
        Window wnd = new Window(5, 5);

        int x = 5;
        int y = 5;

        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);

        x = -1;
        y = -1;
        wnd.ChangePos(ref x, ref y);
        Console.WriteLine("{0}, {1}", x, y);
    }
}

```

Como puede verse, el método que se invoca —*Window.ChangePos*— basa su trabajo en los valores que se le están pasando. En este caso, la palabra reservada *ref* fuerza al invocador a inicializar el valor para que el método funcione correctamente.

SOBRECARGA DE MÉTODOS

La sobrecarga de métodos permite que el programador de C# utilice el mismo nombre de método múltiples veces mientras que los argumentos pasados sean diferentes. Esto es extremadamente útil en al menos dos escenarios. El primero implica situaciones en las que se quiere exponer un único nombre de método donde el comportamiento del método

es ligeramente distinto dependiendo de los tipos de los valores que han pasado. Por ejemplo, digamos que tenemos una clase de registro de actividad (*logging*) que permite que la aplicación escriba información de diagnóstico en el disco. Para hacer la clase un poco más flexible, se podrían tener varias formas del método *Write*, utilizado para especificar la información que se tiene que escribir. Además de aceptar la cadena que tiene que escribirse, el método podría también aceptar una cadena de identificación de recurso. Sin la capacidad de sobrecargar métodos, se tendría que implementar un método para una de estas situaciones, algo como *WriteString* y *WriteFromResourceId*. Sin embargo, utilizando la sobrecarga de métodos, se podrían implementar los siguientes métodos—ambos llamados *WriteEntry*— con cada método diferenciándose sólo en el tipo del parámetro:

```
using System;

class Log
{
    public Log(string fileName)
    {
        //Abrir el archivo fileName y desplazarse hasta el final.
    }

    public void WriteEntry(string entry)
    {
        Console.WriteLine(entry);
    }

    public void WriteEntry(int resourceId)
    {
        Console.WriteLine
            ("Recuperar cadena utilizando el id de recurso y escribir en el
             registro de actividad");
    }
}

class Overloading1App
{
    public static void Main()
    {
        Log log = new Log("MiArchivo");
        log.WriteEntry("EntradaUno");
        log.WriteEntry(42);
    }
}
```

Un segundo escenario en el que es útil la sobrecarga de métodos es cuando se utilizan constructores, que son esencialmente métodos invocados al instanciar un objeto. Digamos que queremos crear una clase que pueda construirse de más de una forma; por ejemplo, tomando tanto un manejador de archivos [un valor entero (*int*)] o un nombre de archivo [una cadena (*string*)] para abrir un archivo. Como las reglas de C# dictan que el constructor de una clase debe tener el mismo nombre que la misma clase, no sirve crear simplemente nombres de métodos distintos para cada uno de los diferentes tipos de variables. En cambio, se necesita sobrecargar el constructor:

```

using System;

class File
{
}

class CommaDelimitedFile
{
    public CommaDelimitedFile(String fileName)
    {
        Console.WriteLine("Construido mediante un nombre de archivo");
    }
    public CommaDelimitedFile(File file)
    {
        Console.WriteLine("Construido con un objeto archivo(File)");
    }
}

class Overloading2App
{
    public static void Main()
    {
        File file = new File();
        CommaDelimitedFile file2 = new CommaDelimitedFile(file);
        CommaDelimitedFile file3 =
            new CommaDelimitedFile("Un nombre de archivo");
    }
}

```

Un punto importante que debemos recordar con relación a la sobrecarga es que la lista de argumentos de cada método debe ser diferente. Por lo tanto, el siguiente código no se compilará porque la única diferencia entre las dos versiones del método *Overloading3App.Foo* es el tipo del valor que se devuelve:

```

using System;

class Overloading3App
{
    void Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double)");
    }

    //ERROR: Sólo difiere en el valor de retorno. No se compilará.
    double Foo(double input)
    {
        Console.WriteLine("Overloading3App.Foo(double) (Segunda versión)");
    }
    public static void Main()
    {
        Overloading3App app = new Overloading3App();

        double i = 5;
        app.Foo(i);
    }
}

```

PARÁMETROS DE MÉTODO VARIABLES

A veces, el número de argumentos que se van a pasar a un método no se conocerán hasta que estemos en tiempo de ejecución. Por ejemplo, digamos que queremos una clase que trace una línea de un grafo de acuerdo con una serie de coordenadas x e y . Podríamos exponer un método de la clase que tomara como único argumento un objeto *Point* que represente tanto a un valor x como a un valor y . Este método almacenaría cada objeto *Point* en una lista enlazada de un array miembro hasta que el invocador quisiera imprimir la secuencia entera de puntos. Sin embargo, esta es una decisión de diseño pobre por un par de razones. Primero requiere que el usuario realice el trabajo innecesario de invocar un método para cada punto de la línea que se tiene que dibujar —bastante tedioso si la línea es larga— y después se ha de invocar otro método para que se dibuje la línea. La segunda desventaja es que se requiere que la clase almacene estos puntos de alguna manera, cuando la única razón por la que se necesitan es para la utilización de un único método, el método *DrawLine*. Los argumentos variables son la solución adecuada para resolver este tipo de problemas.

Se puede especificar un número variable de parámetros de método utilizando la palabra reservada *params* y especificando un array en la lista de argumentos del método. He aquí un ejemplo de la clase *Draw* en C# que permite al usuario hacer una única invocación que tome un número variable de objetos *Point* e imprimir cada uno de ellos:

```
using System;

class Point
{
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int x;
    public int y;
}

class Chart
{
    public void DrawLine(params Point[] p)
    {
        Console.WriteLine("\nEste método imprimiría una línea utilizando" +
                           "los siguientes puntos:");
        for (int i = 0; i < p.GetLength(0); i++)
        {
            Console.WriteLine("{0}, {1}", p[i].x, p[i].y);
        }
    }
}

class VarArgsApp
{
    public static void Main()
```

```

{
    Point p1 = new Point(5, 10);
    Point p2 = new Point(5, 15);
    Point p3 = new Point(5, 20);

    Chart chart = new Chart();
    chart.DrawLine(p1, p2, p3);
}

```

El método *DrawLine* le dice al compilador que puede tomar un número variable de objetos *Point*. En tiempo de ejecución, el método utiliza un bucle *for* sencillo para recorrer los objetos *Point* que se le pasan, imprimiendo cada uno.

Observe que en una aplicación real sería mucho mejor utilizar propiedades para tener acceso a los miembros *x* e *y* de los objetos *Point*, en lugar de hacer los miembros *public*. Además, también sería mejor utilizar la instrucción *foreach* en el método *DrawLine* en lugar de un bucle *for*. Sin embargo, por razones de continuidad en este libro, todavía no se presentan estas características del lenguaje. Las propiedades se tratarán en el Capítulo 7, «Propiedades, arrays e indizadores», y la instrucción *foreach* en el Capítulo 11, «Control del flujo de programa».

MÉTODOS VIRTUALES

Como vimos en el Capítulo 5, se puede derivar una clase de otra para que esta clase pueda heredar y construir a partir de las capacidades de una clase existente. Como aún no se habían tratado los métodos, esa discusión tocaba sólo la herencia de campos y de métodos. En otras palabras, no vimos la capacidad de modificar el comportamiento de la clase base en las clases derivadas. Esto se hace utilizando métodos virtuales, y ese es el tema de esta sección.

Redefinición de métodos

Echemos primero un vistazo a cómo redefinir la funcionalidad de un método heredado de la clase base. Empezaremos con una clase base que representa a un empleado. Para hacer el ejemplo lo más sencillo posible, le daremos a esta clase base un único método llamado *CalculatePay*, donde no hacemos más que mostrar el nombre del método invocado. Esto nos ayudará a determinar más tarde qué métodos del árbol de herencia se invocarán.

```

class Employee
{
    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay()");
    }
}

```

Ahora, digamos que queremos derivar una clase a partir de *Employee* y queremos redefinir el método *CalculatePay* para hacer algo específico en la clase derivada. Para ello necesitamos utilizar la palabra reservada *new* con la definición del método de la clase derivada. He aquí el código que muestra lo fácil que es:

```
using System;

class Employee
{
    public void CalculatePay()
    {
        Console.WriteLine("Employee.CalculatePay()");
    }
}

class SalariedEmployee : Employee
{
    //La palabra reservada new le permite redefinir la
    //implementación de la clase base.
    new public void CalculatePay()
    {
        Console.WriteLine("SalariedEmployee.CalculatePay()");
    }
}

class Poly1App
{
    public static void Main()
    {
        Poly1App poly1 = new Poly1App();

        Employee baseE = new Employee();
        baseE.CalculatePay();
        SalariedEmployee s = new SalariedEmployee();
        s.CalculatePay();
    }
}
```

Si compilamos y ejecutamos esta aplicación, obtenemos el siguiente resultado:

```
c:\>Poly1App
Employee.CalculatePay()
Salaried.CalculatePay()
```

Polimorfismo

La redefinición de métodos con la palabra reservada *new* funciona bien si se tiene una referencia al objeto derivado. Sin embargo, ¿qué pasa si se tiene la referencia con una conversión de tipo a su clase base y queremos que el compilador invoque la implementación de un método de la clase derivada? Por esta razón entra en escena el polimorfismo. El polimorfismo posibilita que definamos un método múltiples veces a través de la

jerarquía de clases de forma que el entorno de ejecución invoque la versión apropiada de ese método dependiendo del objeto exacto que se utilice.

Echemos un vistazo a nuestro ejemplo de empleados para ver lo que queremos decir. La aplicación Poly1App se ejecuta correctamente porque tenemos dos objetos: un objeto *Employee* y un objeto *SalariedEmployee*. En una aplicación más elaborada, probablemente leeríamos todos los registros de empleados de una base de datos y los volcaríamos en un array. Aunque algunos de estos empleados serían contratantes y algunos serían empleados asalariados, necesitaríamos colocarlos en nuestro array como si fueran del mismo tipo —el tipo de la clase base, *Employee*. Sin embargo, cuando recorremos el array, recuperando e invocando el método *CalculatePay* de cada objeto, querríamos que el compilador invocara a la implementación correcta del método *CalculatePay* del objeto.

En el siguiente ejemplo hemos añadido una nueva clase, *ContractEmployee*. La clase principal de la aplicación contiene ahora un array de tipo *Employee* y dos métodos adicionales: *LoadEmployees*, que carga los objetos empleados en el array, y *DoPayroll*, que recorre el array, invocando el método *CalculatePay* de cada objeto.

```
using System;

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
    }

    public void CalculatePay()
    {
        Console.WriteLine("Se invoca Employee.CalculatePay para {0}",
                           name);
    }
}

class ContractEmployee : Employee
{
    public ContractEmployee(string name)
    : base(name)
    {
    }

    public new void CalculatePay()
    {
        Console.WriteLine("Se invoca ContractEmployee.CalculatePay para {0}",
                           name);
    }
}
```

```

    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee (string name)
        : base(name)
    {
    }

    public new void CalculatePay()
    {
        Console.WriteLine("Se invoca SalariedEmployee.CalculatePay para {0}",
            name);
    }
}

class Poly2App
{
    protected Employee[] employees;
    public void LoadEmployees()
    {
        //Simulando la carga de la base de datos.
        employees = new Employee[2];
        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly2App poly2 = new Poly2App();
        poly2.LoadEmployees();
        poly2.DoPayroll();
    }
}

```

Si ejecutamos esta aplicación, como resultado obtenemos lo siguiente:

```

c:\>Poly2App
Se invoca Employee.CalculatePay para Kate Dresen
Se invoca Employee.CalculatePay para Megan Sherman

```

Obviamente, esto no es lo que queríamos —la implementación de *CalculatePay* de la clase base se invoca para cada objeto. Lo que ha sucedido es un ejemplo de un fenómeno llamado *enlace en tiempo de compilación* (*early binding*). Cuando se compiló el código, el compilador de C# miró la invocación de *emp.CalculatePay* y determinó la dirección

de memoria a la que saltaría cuando se hiciera la invocación. En este caso, sería la situación en memoria del método *Employee.CalculatePay*.

Echemos un vistazo al siguiente MSIL que se generó a partir de la aplicación Poly2App, y tomemos nota en concreto de la línea *IL_0014* y del hecho de que invoca explícitamente al método *Employee.CalculatePay*:

```
.method public hidebysig instance void DoPayroll() il managed
{
    //Code size          34 (0x22)
    .maxstack 2
    .locals (class Employee V_0,
             class Employee[] V_1,
             int32 V_2,
             int32 V_3)
    IL_0000:    ldarg.0
    IL_0001:    ldflld      class Employee[] Poly2App::employees
    IL_0006:    stloc.1
    IL_0007:    ldloc.1
    IL_0008:    ldlen
    IL_0009:    conv.i4
    IL_000a:    stloc.2
    IL_000b:    ldc.i4.0
    IL_000c:    stloc.3
    IL_000d:    br.s          IL_001d
    IL_000f:    ldloc.1
    IL_0010:    ldloc.3
    IL_0011:    ldelem.ref
    IL_0012:    stloc.0
    IL_0013:    ldloc.0
    IL_0014:    call          instance void Employee::CalculatePay()
    IL_0019:    ldloc.3
    IL_001a:    ldc.i4.1
    IL_001b:    add
    IL_001c:    stloc.3
    IL_001d:    ldloc.3
    IL_001e:    ldloc.2
    IL_001f:    blt.s        IL_000f
    IL_0021:    ret
} //end of method Poly2App::DoPayroll
```

Esa invocación al método *Employee.CalculatePay* es el problema. Lo que queremos que se produzca en su lugar es *enlace en tiempo de ejecución (late binding)*. El enlace en tiempo de ejecución significa que el compilador no selecciona el método a ejecutar hasta el que se esté ejecutando la aplicación. Para forzar al compilador a invocar la versión correcta del método de un objeto cuyo tipo ha sido convertido al de una clase anterior en la jerarquía, utilizamos dos nuevas palabras reservadas: *virtual* y *override*. La palabra reservada *virtual* debe utilizarse con el método de la clase base, y la palabra reservada *override* se utiliza con la implementación del método de la clase derivada. Aquí tenemos el ejemplo otra vez —¡esta vez funcionando correctamente!:

```
using System;
```

```

class Employee
{
    public Employee(string name)
    {
        this.Name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
    }

    virtual public void CalculatePay()
    {
        Console.WriteLine("Se invoca Employee.CalculatePay para {0}",
                           name);
    }
}

class ContractEmployee : Employee
{
    public ContractEmployee(string name)
    : base(name)
    {
    }

    override public void CalculatePay()
    {
        Console.WriteLine("Se invoca ContractEmployee.CalculatePay para {0}",
                           name);
    }
}

class SalariedEmployee : Employee
{
    public SalariedEmployee (string name)
    : base(name)
    {
    }

    override public void CalculatePay()
    {
        Console.WriteLine("Se invoca SalariedEmployee.CalculatePay para {0}",
                           name);
    }
}

class Poly3App
{
    protected Employee[] employees;
    public void LoadEmployees()
    {
        //Simulando la carga de la base de datos.
        employees = new Employee[2];
    }
}

```

```

        employees[0] = new ContractEmployee("Kate Dresen");
        employees[1] = new SalariedEmployee("Megan Sherman");
    }

    public void DoPayroll()
    {
        foreach(Employee emp in employees)
        {
            emp.CalculatePay();
        }
    }

    public static void Main()
    {
        Poly3App poly3 = new Poly3App();
        poly3.LoadEmployees();
        poly3.DoPayroll();
    }
}

```

Antes de ejecutar esta aplicación, vamos a echar un vistazo al código IL que se ha generado, esta vez observando que la línea *IL_0010* utiliza el código de operación de MSIL *callvirt*, que le dice al compilador que el método exacto que se va a invocar no se sabrá hasta el momento de la ejecución porque depende de qué objeto derivado se esté utilizando:

```

.method public hidebysig instance void DoPayroll() cil managed
{
    //Code size          32 (0x20)
    .maxstack 2
    .locals (class Employee V_0,
             class Employee[] V_1,
    int32 V_2)
    IL_0000: ldarg.0
    IL_0001: ldfld          class Employee[] Poly3App::employees
    IL_0006: stloc.1
    IL_0007: ldc.i4.0
    IL_0008: stloc.2
    IL_0009: br.s          IL_0019
    IL_000b: ldloc.1
    IL_000c: ldloc.2
    IL_000d: ldelem.ref
    IL_000e: stloc.0
    IL_000f: ldloc.0
    IL_0010: callvirt      instance void Employee::CalculatePay()
    IL_0015: ldloc.2
    IL_0016: ldc.i4.1
    IL_0017: add
    IL_0018: stloc.2
    IL_0019: ldloc.2
    IL_001a: ldloc.1
    IL_001b: ldlen
    IL_001c: conv.i4
    IL_001d: blt.s          IL_000b
    IL_001f: ret
} //end of method Poly3App::DoPayroll

```

Si ejecutamos el código ahora, se deberían generar los siguientes resultados:

```
c:\>Poly3App
Se invoca ContractEmployee.CalculatePay para Kate Dresen
Se invoca SalariedEmployee.CalculatePay para Megan Sherman
```

NOTA Los métodos virtuales no se pueden declarar como *private* porque, por definición, no serían visibles en las clases derivadas.

MÉTODOS ESTÁTICOS

Un método estático es un método que existe en una clase como un todo más que en una instancia específica de la clase. Como sucede con otros miembros estáticos, el principal beneficio de los métodos estáticos es que residen fuera de una instancia particular de clase sin contaminar el espacio global de la aplicación y sin ir contra la granularidad de la orientación a objetos, por no estar asociados con una clase. Un ejemplo de esto es una API de base de datos que escribimos en C#. En nuestra jerarquía de clases, teníamos una clase llamada *SQLServerDb*. Junto con las capacidades básicas de CMLB (creación, modificación, lectura y borrado), la clase también tenía un método para reparar la base de datos. En el método de clase *Repair* no era necesario abrir la base de datos. En realidad, la función ODBC que se utilizó (*SQLConfigDataSource*) se encargaba de que la base de datos se cerrara durante la operación. Sin embargo, el constructor de *SQLServerDb* abría una base de datos especificada en un nombre que se le pasaba a tal efecto. Por lo tanto era perfecto un método estático. Nos permitía colocar un método en la clase *SQLServerDb* y así no tener que invocar el constructor de nuestra clase. Obviamente, por parte del cliente, el beneficio era que no tenía que instanciar tampoco la clase *SQLServerDb*. En el siguiente ejemplo se puede ver un método estático (*RepairDatabase*) que se invoca desde el método *Main*. Observe que no creamos una instancia de *SQLServerDB* para hacerlo:

```
using System;

class SQLServerDb
{
    //Otros miembros sin importancia.
    public static void RepairDatabase()
    {
        Console.WriteLine("reparando base de datos...");
    }
}

class StaticMethodApp
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

El definir un método como estático implica utilizar la palabra reservada *static*. El usuario emplea entonces la sintaxis *Clase.Método* para invocarlo. Observe que esta sintaxis es necesaria incluso si el usuario tiene una referencia a una instancia de la clase. Para ilustrar este punto, el siguiente código fallaría al compilar:

```
//Este código fallará al compilar.
using System;

class SQLServerDb
{
    //Otros miembros sin importancia.
    public static void RepairDatabase()
    {
        Console.WriteLine("reparando base de datos...");
    }
}

class StaticMethod2App
{
    public static void Main()
    {
        SQLServerDb db = new SQLServerDb();
        db.RepairDatabase();
    }
}
```

Acceso a los miembros de clase

El último punto que trataremos sobre los métodos estáticos es la regla que controla qué miembros de clase se pueden acceder desde un método estático. Como podría imaginar, un método estático puede acceder a cualquier miembro estático en la clase, pero no puede acceder a un miembro de instancia. Esto queda ilustrado en el siguiente código:

```
using System;

class SQLServerDb
{
    static string progressString1 = "reparando base de datos...";
    string progressString2 = "reparando base de datos...";

    public static void RepairDatabase()
    {
        Console.WriteLine(progressString1);    //Esto funcionará.
        Console.WriteLine(progressString2);    //Falla al compilar.
    }
}

class StaticMethod3App
{
    public static void Main()
    {
        SQLServerDb.RepairDatabase();
    }
}
```

RESUMEN

Los métodos dan a las clases sus características de comportamiento y llevan a cabo acciones en nuestro lugar. Los métodos en C# son flexibles, permitiendo que se devuelvan múltiples valores, sobrecarga y parámetros variables. Las palabras reservadas *ref* y *out* permiten que un método devuelva más de un único valor al invocador. La sobrecarga permite que varios métodos con el mismo nombre funcionen de manera diferente, dependiendo de los tipos y/o el número de argumentos que se les ha pasado. Los métodos pueden tener parámetros variables. La palabra reservada *params* permite tratar con métodos donde no se conoce el número de argumentos hasta el momento de la ejecución. Los métodos virtuales permiten controlar cómo se modifican los métodos heredados en clases derivadas. Finalmente, la palabra reservada *static* permite a los métodos que existan como parte de una clase más que como parte de un objeto.

Capítulo 7

Propiedades, arrays e indizadores

Hasta ahora hemos descrito los tipos básicos que permite C# y cómo declararlos y utilizarlos en las clases y aplicaciones. Este capítulo romperá el patrón de presentar una característica importante del lenguaje por capítulo. En este capítulo aprenderemos sobre propiedades, arrays e indizadores, porque estas características del lenguaje comparten un punto en común. Permiten al desarrollador de clases de C# extender la estructura básica de clases/campos/métodos de clase al exponer una interfaz más intuitiva y natural para los miembros de la clase.

PROPIEDADES COMO CAMPOS INTELIGENTES

Siempre ha sido un buen objetivo diseñar clases que no sólo escondan la implementación de los métodos de clase, sino que también hagan que cualquier miembro no pueda acceder directamente a los campos de clases. Proporcionando *métodos de acceso*, cuyo trabajo es recuperar y asignar los valores a esos campos, puede asegurarse el que un campo se trate correctamente —esto es, de acuerdo con las reglas del dominio del problema específico— y que se lleve a cabo cualquier proceso adicional que se necesite.

Como ejemplo, digamos que tenemos una clase dirección con un campo código postal (*ZIP code*) y un campo ciudad. Cuando el cliente modifica el campo *Address.ZipCode*, se quiere validar el código postal contra una base de datos y asignar automáticamente el campo *Address.City* basándonos en ese código. Si el cliente tuviera acceso directo a un miembro *public Address.ZipCode*, no habría una manera fácil de hacerlo, porque cambiar directamente la variable de un miembro no requiere un método. Por lo tanto, en lugar de garantizar acceso directo al campo *Address.ZipCode*, un mejor diseño sería definir los campos *Address.ZipCode* y *Address.City* como *protected* y proporcionar unos métodos de acceso para recuperar y asignar el campo *Address.ZipCode*. De esta manera se puede enlazar código con el cambio que realice cualquier trabajo adicional que se tenga que hacer.

Este ejemplo de código postal se programaría en C# como sigue. Observe que el campo real *ZipCode* se define como *protected* y por lo tanto no es accesible desde el cliente, y los métodos de acceso *GetZipCode* y *SetZipCode* se definen como *public*.

```
class Address
{
    protected string ZipCode;
    protected string City;
    public string GetZipCode()
    {
        return this.ZipCode;
    }

    public void SetZipCode(string ZipCode)
    {
        //Validar ZipCode contra algún repositorio de datos.
        this.ZipCode = ZipCode;
        //Actualizar this.City basándonos en el código validado.
    }
}
```

El cliente accedería a un valor *Address.ZipCode* de la siguiente forma:

```
Address addr = new Address();
addr.SetZipCode("55555");
string zip = addr.GetZipCode();
```

Cómo definir y utilizar propiedades

Utilizar métodos de acceso funciona bien y es una técnica utilizada por los programadores de varios lenguajes orientados a objetos, incluidos C++ y Java. Sin embargo, C# proporciona un mecanismo más rico incluso —las propiedades— que tienen las mismas capacidades como métodos de acceso y son mucho más elegantes para el cliente. Mediante las propiedades, un programador puede escribir un cliente que pueda acceder a los campos de una clase como si fueran públicos sin saber si existe un método de acceso.

Una propiedad de C# consiste en una declaración de campo y métodos de acceso que se utilizan para modificar el valor del campo. Estos métodos de acceso se llaman métodos de *obtención o lectura (getter)* y métodos de *establecimiento o asignación (setter)*. Los métodos de obtención o lectura se utilizan para recuperar los valores del campo y los métodos de establecimiento o asignación se utilizan para modificar el valor del campo. A continuación mostramos el ejemplo anterior reescrito utilizando propiedades de C#:

```
class Address
{
    protected string city;
    protected string zipCode;

    public string ZipCode
    {
        get
        {
```

```

        return zipCode;
    }

    set
    {
        //Validar value contra algún repositorio de datos.
        zipCode = value;
        //Actualizar city basándonos en el código validado.
    }
}

```

Observe que se ha creado un campo llamado *Address.zipCode* y una propiedad llamada *Address.ZipCode*. Esto puede confundir un poco al principio, porque se puede pensar que *Address.zipCode* es el campo y nos podemos preguntar por qué necesita definirse dos veces. Pero no es el campo. Es la propiedad, que es, simplemente, un medio genérico de definir los accesos a miembros de clase para que se pueda utilizar la sintaxis más intuitiva de *objeto.campo*. Si en este ejemplo se fuera a omitir el campo *Address.zipCode* y a cambiar la instrucción *zipCode=value* en el método de asignación por *ZipCode=value*, provocaríamos que se invocase infinitamente al método de asignación. Observe también que el método de asignación no tiene ningún parámetro. El valor que se pasa es inmediatamente colocado en una variable llamada *value* que es accesible dentro del método de asignación. (Pronto veremos cómo se produce la magia en MSIL).

Ahora que hemos escrito la propiedad *Address.zipCode*, echemos un vistazo a los cambios que se necesitan para el código del cliente:

```

Address addr = new Address();
addr.ZipCode = "55555";
string zip = addr.ZipCode;

```

Como puede verse, la forma como accede un cliente a los campos es intuitiva: ya no hay que adivinar o examinar la documentación (también llamada código fuente) para determinar si un campo es *public*, y si no, cuál es el nombre del método de acceso.

Lo que está haciendo realmente el compilador

Entonces, ¿cómo nos permite invocar un método mediante la sintaxis estándar *objeto.campo* el compilador? Y también, ¿de dónde viene esa variable *value*? Para contestar a estas preguntas, tenemos que mirar en el MSIL producido por el compilador. Vamos a considerar primero el método de lectura de la propiedad.

En el ejemplo siguiente, tenemos el siguiente método de lectura definido:

```

class Address
{
    protected string city;
    protected string zipCode;
    public string ZipCode
    {

```

```

        get
        {
            return zipCode;
        }
        :
    }
    :
}

```

Si se mira el MSIL resultante de este método, verá que el compilador ha creado un método de acceso llamado *get_ZipCode*, como se muestra a continuación:

```

.method public hidebysig specialname instance string
    get_ZipCode() cil managed
{
    //Code size          11 (0xb)
    .maxstack 1
    .locals ([0] string _Vb_t_$00000003$00000000)
    IL_0000: ldarg.0
    IL_0001: ldfld      string Address::zipCode
    IL_0006: stloc.0
    IL_0007: br.s      IL_0009
    IL_0009: ldloc.0
    IL_000a: ret
} //end of method Address::get_ZipCode

```

Se puede utilizar el nombre del método de acceso porque el compilador prefija el nombre de la propiedad con *get_* (para un método de lectura) o *set_* (para un método de asignación). Como resultado, el siguiente código se convierte en una invocación a *get_ZipCode*:

```
String str = addr.ZipCode; //esto invoca Address::get_ZipCode
```

Por lo tanto, podríamos vernos tentados a probar la siguiente invocación explícita al método de acceso:

```
String str = addr.get_ZipCode; /**ERROR -No compilará
```

Sin embargo, en este caso, el código no compilará porque es ilegal invocar explícitamente un método MSIL interno.

La respuesta a nuestra pregunta —¿cómo nos permite un compilador utilizar la sintaxis *object.field* e invocar un método?— es que el compilador realmente genera los métodos de lectura y asignación apropiados cuando analiza la sintaxis de la propiedad C#. Por lo tanto, en el caso de la propiedad *Address.zipCode*, el compilador genera MSIL que contiene los métodos *get_ZipCode* y el *set_ZipCode*.

Ahora echemos un vistazo al método de asignación generado. En la clase *Address* vimos lo siguiente:

```

public string ZipCode
{

```

```

:
set
{
    //Validar value contra algún repositorio de datos.
    zipCode = value;
    //Actualizar city basándonos en el código validado.
}
}

```

Observe que en este código no se declara una variable llamada *value*, aunque podemos utilizarla para almacenar el valor que nos pasa el invocador y para asignar al campo miembro protegido *zipCode*. Cuando el compilador C# genera el MSIL para un método de asignación, añade esta variable como argumento en un método llamado *set_ZipCode*.

En el MSIL generado, este método tiene como argumento una variable de tipo cadena (string):

```

.method public hidebysig specialname instance void
    set_ZipCode(string 'value') cil managed
{
    //Code size 8          (0x8)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: ldarg.1
    IL_0002: stfld      string Address::zipCode
    IL_0007: ret
} //end of method Address::set_ZipCode

```

Incluso aunque no se pueda ver este método en el código de fuente C#, cuando le asigna un valor a la propiedad *ZipCode* con algo como *addr.ZipCode*=«12345», se transforma en una invocación MSIL a *Address::set_ZipCode*(«12345»). Como sucede con el método *get_ZipCode*, si intentamos invocar este método directamente en C#, se genera un error.

Propiedades de sólo lectura

En el ejemplo que hemos estado utilizando, la propiedad *Address.zipCode* se considera de lectura/escritura, porque se definen tanto un método de lectura como un método de asignación. A veces no queremos que el cliente pueda establecer el valor de un campo dado, en cuyo caso haremos el campo de sólo lectura. Logramos esto omitiendo el método de asignación. Para ilustrar una propiedad de sólo lectura, tenemos que evitar que el cliente asigne un valor al campo *Address.city*, haciendo que la propiedad *Address.zipCode* sea la única que puede cambiar el valor de este campo:

```

class Address
{
    protected string city;
    public string City
    {
        get
        {

```

```

        return city;
    }
}

protected string zipCode;
public string ZipCode
{
    get
    {
        return zipCode;
    }
    set
    {
        //Validar value contra algún repositorio de datos.
        zipCode = value;
        //Actualizar city basándonos en el código validado.
    }
}
}

```

Cómo heredar propiedades

Al igual que sucede con los métodos, las propiedades se pueden decorar con modificadores *virtual*, *override* o *abstract*, que ya se trataron en el Capítulo 6, «Métodos». Esto permite que una clase derivada herede o redefina propiedades como si fueran cualquier otro miembro de la clase base. El elemento clave aquí es que podemos especificar estos modificadores sólo al nivel de propiedad. En otras palabras, en casos en los que tenemos tanto un método de lectura como uno de asignación, si se anula uno, debemos anular los dos.

Utilización avanzada de propiedades

Hasta ahora hemos hablado de que las propiedades son útiles por las siguientes razones:

- Proporcionan un nivel de abstracción a los clientes.
- Proporcionan medios genéricos de acceso a los miembros de las clases utilizando la sintaxis *objeto.campo*.
- Permiten que una clase garantice que se pueda hacer cualquier procesamiento adicional cuando un campo particular sea accedido o se modifique.

La tercera razón nos conduce a otra utilización de las propiedades: la implementación de algo llamado *inicialización perezosa*. Esto es una optimización técnica por la que los miembros de una clase no se inicializan hasta que no se necesitan.

La inicialización perezosa es beneficiosa cuando se tiene una clase que contiene miembros a los que rara vez se hace referencia cuya inicialización tarda mucho tiempo o consume muchos recursos. Un ejemplo serían aquellas situaciones en las que los datos tienen que leerse desde una base de datos o a través de una red congestionada. Como

sabemos que a estos miembros no se les hace referencia muy a menudo y su inicialización es cara, podemos retrasar su inicialización hasta que se invoquen los métodos de lectura. Para ilustrar esto, digamos que tenemos una aplicación de inventario que ejecutan los agentes de ventas en sus portátiles para realizar los pedidos de clientes y que ocasionalmente utilizan para comprobar los niveles de inventario. Utilizando las propiedades, podríamos permitir que las clases relevantes se instanciaran sin tener que leer los registros del inventario, como se muestra en el código que sigue. Entonces, si un agente de ventas quisiera acceder al inventario para consultar un elemento, el método de lectura, en ese mismo momento, accedería a una base de datos remota.

```
class Sku
{
    protected double onHand;

    public string OnHand
    {
        get
        {
            //Leer de la BD Central y fijar el valor de onHand.
            return onHand;
        }
    }
}
```

Como ya hemos visto a lo largo de este capítulo, las propiedades nos permiten proporcionar métodos de acceso a los campos y una interfaz genérica e intuitiva para el cliente. A causa de esto, las propiedades se conocen a veces como *campos inteligentes*. Ahora, vamos a dar un paso más y veremos cómo se definen y se utilizan los arrays en C#. También veremos cómo se utilizan las propiedades con arrays en forma de *indizadores*.

ARRAYS

Hasta ahora, la mayoría de los ejemplos de este libro nos ha mostrado cómo definir un número finito y predeterminado de variables. Sin embargo, en aplicaciones más prácticas, no sabemos el número exacto de objetos que necesitamos hasta la ejecución. Por ejemplo, si estamos desarrollando un editor y queremos seguir el rastro de los controles que se añaden a una cuadro de diálogo, el número exacto de controles que mostrará el editor no se puede conocer hasta que lo ejecutemos. Sin embargo, podemos utilizar un array para almacenar y seguir el rastro a un grupo de objetos cuya memoria se reserva dinámicamente —los controles del editor en este caso.

En C#, los arrays son objetos cuya clase base es *System.Array*. Por lo tanto, aunque la sintaxis para definir un array parezca similar a la de C++ o Java, realmente estamos instanciando una clase .NET, lo que significa que todos los arrays declarados tienen los mismos miembros heredados de *System.Array*. En esta sección, estudiaremos cómo declarar e instanciar arrays, cómo trabajar con los diferentes tipos de arrays y cómo iterar a través de los elementos de un array. También veremos algunas de las propiedades y métodos de la clase *System.Array* más comúnmente utilizados.

Cómo declarar arrays

Para declarar un array en C# se colocan corchetes vacíos entre el tipo y el nombre variable; así:

```
int[] numbers;
```

Observe que esta sintaxis difiere ligeramente de C++, en la que los corchetes se colocan después del nombre de la variable. Como los arrays están basados en clases, muchas de las mismas reglas que se aplican para declarar una clase también se aplican a los arrays. Por ejemplo, cuando declaramos un array, realmente no estamos creando ese array. Igual que hacemos con una clase, debemos instanciar el array antes de que exista en términos de tener memoria reservada para sus elementos. En el siguiente ejemplo se declara y se instancia un array al mismo tiempo:

```
//Declara e instancia un array unidimensional de 6 enteros.
int[] numbers = new int[6];
```

Sin embargo, cuando declaramos el array como miembro de una clase, tenemos que declarar e instanciar el array en dos pasos distintos, porque no podemos instanciar un objeto hasta el momento de la ejecución:

```
class YourClass
{
    :
    int[] numbers;
    :

    void SomeInitMethod()
    {
        :
        numbers = new int[6];
        :
    }
}
```

Ejemplo de array unidimensional

A continuación tenemos un ejemplo de declaración de un array unidimensional como miembro de una clase, la instanciación y relleno del array en el constructor y después la utilización de un bucle *for* para recorrerlo, imprimiendo cada elemento:

```
using System;

class SingleDimArrayApp
{
    protected int[] numbers;
```

```

SingleDimArrayApp()
{
    numbers = new int[6];
    for (int i = 0; i < 6; i++)
    {
        numbers[i] = i * i;
    }
}

protected void PrintArray()
{
    for (int i = 0; i < numbers.Length; i++)
    {
        Console.WriteLine("números [{0}] = {1}", i, numbers[i]);
    }
}

public static void Main()
{
    SingleDimArrayApp app = new SingleDimArrayApp();
    app.PrintArray();
}
}

```

Si ejecutamos este ejemplo, se produce el siguiente resultado:

```

números[0] = 0
números[1] = 1
números[2] = 4
números[3] = 9
números[4] = 16
números[5] = 25

```

En este ejemplo, el método *SingleDimArray.PrintArray* utiliza la propiedad *Length* de *System.Array* para determinar el número de elementos en el array. Aunque no es obvio aquí, porque sólo tenemos un array de una dimensión, la propiedad *Length* devuelve realmente el número de todos los elementos en todas las dimensiones de un array. Por lo tanto, en el caso de un array bidimensional de 5 por 4, la propiedad *Length* devolvería 9. En la próxima sección veremos los arrays multidimensionales y cómo determinar el límite superior de una dimensión determinada del array.

Arrays multidimensionales

Además de los arrays unidimensionales, C# permite la declaración de arrays multidimensionales en los que cada dimensión del array se separa por una coma. Aquí se declara un array tridimensional de valores en coma flotante (*double*):

```
double[, ,] numbers;
```

Para determinar rápidamente el número de dimensiones en un array de C#, hay que contar el número de comas y sumar una al total.

En el siguiente ejemplo, tenemos un array bidimensional de cifras de ventas que representan las cifras hasta la fecha de este año y los totales del año pasado durante el mismo período de tiempo. Observe especialmente la sintaxis utilizada para instanciar el array (en el constructor *MultiDimArrayApp*).

```
using System;

class MultiDimArrayApp
{
    protected int currentMonth;
    protected double[,] sales;

    MultiDimArrayApp()
    {
        currentMonth = 10;

        sales = new double[2,currentMonth];
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j = 0; j < 10; j++)
            {
                sales[i,j] = (i * 100) + j;
            }
        }
    }

    protected void PrintSales()
    {
        for (int i = 0; i < sales.GetLength(0); i++)
        {
            for (int j = 0; j < sales.GetLength(1); j++)
            {
                Console.WriteLine("[{0}][{1}] = {2}", i, j, sales[i,j]);
            }
        }
    }

    public static void Main()
    {
        MultiDimArrayApp app = new MultiDimArrayApp();
        app.PrintSales();
    }
}
```

Al ejecutar el ejemplo *MultiDimArrayApp* obtenemos estos resultados:

```
[0] [0] = 0
[0] [1] = 1
[0] [2] = 2
[0] [3] = 3
[0] [4] = 4
[0] [5] = 5
[0] [6] = 6
[0] [7] = 7
[0] [8] = 8
[0] [9] = 9
```

```

[1][0] = 100
[1][1] = 101
[1][2] = 102
[1][3] = 103
[1][4] = 104
[1][5] = 105
[1][6] = 106
[1][7] = 107
[1][8] = 108
[1][9] = 109

```

Recuerde que en el ejemplo del array unidimensional se dijo que la propiedad *Length* devolvería el número total de elementos del array, por eso en este ejemplo el valor que devuelve sería 20. En el método *MultiDimArray.PrintSales* se utilizó el método *Array.GetLength* para determinar la longitud o el límite superior de cada dimensión del array. Entonces se pudo utilizar cada valor específico en el método *PrintSales*.

Cómo consultar el rango

Ahora que hemos visto qué fácil es recorrer dinámicamente un array unidimensional o multidimensional, podríamos preguntarnos cómo determinar el número de dimensiones en un array por programa. El número de dimensiones en un array se llama *rango* de un array, y el rango se recupera utilizando la propiedad *Array.Rank*. He aquí un ejemplo de cómo hacerlo con varios arrays:

```

using System;

class RankArrayApp
{
    int[] singleD;
    int[,] doubleD;
    int[,,] tripleD;

    protected RankArrayApp()
    {
        singleD = new int[6];
        doubleD = new int[6,7];
        tripleD = new int[6,7,8];
    }

    protected void PrintRanks()
    {
        Console.WriteLine("Rango de singleD = {0}", singleD.Rank);
        Console.WriteLine("Rango de doubleD = {0}", doubleD.Rank);
        Console.WriteLine("Rango de tripleD = {0}", tripleD.Rank);
    }

    public static void Main()
    {
        RankArrayApp app = new RankArrayApp();
        app.PrintRanks();
    }
}

```

Como era de esperar, la aplicación *RankArrayApp* genera lo siguiente:

```
singleD Rank = 1
doubleD Rank = 2
tripleD Rank = 3
```

Arrays irregulares

Lo último que veremos en relación con los arrays es el *array irregular*. Un array irregular es simplemente un array de arrays. He aquí un ejemplo de cómo definir un array que contiene arrays de enteros:

```
int[][] jaggedArray;
```

Se podría utilizar un array irregular si se estuviera desarrollando un editor. En este editor, podría querer almacenar el objeto que representa cada control creado por el usuario en un array. Digamos que tenemos un array de botones y cuadros combinados (*combo boxes*) (para mantener el ejemplo pequeño y manejable). Se podrían tener tres botones y dos cuadros combinados almacenados en sus respectivos arrays. Si declaramos un array irregular, podemos tener un array «padre» para esos arrays, de forma que podremos recorrer fácilmente mediante programa los controles cuando los necesitemos, como se muestra a continuación:

```
using System;

class Control
{
    virtual public void SayHi()
    {
        Console.WriteLine("clase de control básica");
    }
}

class Button : Control
{
    override public void SayHi()
    {
        Console.WriteLine("control botón");
    }
}

class Combo : Control
{
    override public void SayHi()
    {
        Console.WriteLine("control cuadro combinado");
    }
}

class JaggedArrayApp
{
```

```

public static void Main()
{
    Control[] [] controls;
    controls = new Control[2] [];

    controls[0] = new Control[3];
    for (int i = 0; i < controls[0].Length; i++)
    {
        controls[0][i] = new Button();
    }

    controls[1] = new Control[2];
    for (int i = 0; i < controls[1].Length; i++)
    {
        controls[1][i] = new Combo();
    }

    for (int i = 0; i < controls.Length; i++)
    {
        for (int j = 0; j < controls[i].Length; j++)
        {
            Control control = controls[i][j];
            control.SayHi();
        }
    }

    string str = Console.ReadLine();
}

```

Como puede verse, se han definido una clase base (*Control*) y dos clases derivadas (*Button* y *Combo*) y se ha declarado el array irregular como un array de arrays que contiene los objetos *Control*. De esa manera se pueden almacenar los tipos específicos del array, y a través de la magia del polimorfismo, saber que a la hora de extraer los objetos del array (mediante un objeto *cuyo tipo se ha convertido a la clase padre*) se obtendrá el comportamiento esperado.

CÓMO TRATAR OBJETOS COMO ARRAYS UTILIZANDO INDIZADORES

En la sección de «Arrays» aprendimos cómo declarar e instanciar arrays, cómo trabajar con los diferentes tipos de array y cómo recorrer sus elementos. También aprendimos cómo aprovecharnos de algunas de las propiedades y métodos de los tipos de array más comúnmente utilizados, subrayando el uso de la clase *System.Array*. Continuemos trabajando con arrays echando un vistazo a cómo una característica específica de C# llamada indizadores permite tratar por programa objetos como si fueran arrays.

Y ¿por qué queríamos tratar un objeto como un array? Como la mayoría de características de un lenguaje de programación, el beneficio de los indizadores reside en hacer una aplicación más intuitiva de escribir. En la primera sección de este capítulo, «Propiedades como campos inteligentes», vimos cómo las propiedades de C# nos daban la

posibilidad de hacer referencia a campos de clase utilizando la sintaxis estándar *clase.campo*, aunque en último término se resuelven a métodos de lectura y asignación. Esta abstracción libera al programador que escribe el cliente de una clase de tener que determinar si los métodos de lectura/asignación existen para el campo y de tener que saber el formato exacto de estos métodos. De manera similar, los indizadores permiten a un cliente de clase indizar un objeto como si éste, por sí mismo, fuera un array.

Considere el siguiente ejemplo. Tenemos una clase cuadro de lista (*list box*) que necesita exponer alguna forma en la que un usuario de esa clase pueda insertar cadenas. Si está familiarizado con el Win32 SDK, sabe que para insertar una cadena en una ventana de cuadro de lista hay que enviar un mensaje *LB_ADDSTRING* o *LB_INSERTSTRING*. Cuando este mecanismo apareció a finales de los ochenta pensábamos que éramos realmente programadores orientados a objetos. Después de todo, ¿no estábamos mandando mensajes a un objeto como nos decían esos maravillosos libros de diseño y análisis orientado a objetos? Sin embargo, a medida que empezaron a proliferar lenguajes orientados a objetos y basados en objetos como C++ y Object Pascal, aprendimos que los objetos podían utilizarse para crear interfaces de programación más intuitivas para tales tareas. Utilizando C++ y MFC (Microsoft Foundation Classes), se nos proporcionó un entramado completo de clases que nos posibilitaba tratar las ventanas (como los cuadros de lista) como objetos con estas clases que exponían las funciones de los miembros cuya función era proporcionar básicamente un pequeño adaptador para el envío y la recepción de mensajes desde y hacia el control subyacente de Microsoft Windows. En el caso de la clase *CListBox* (esto es, el adaptador MFC para el control cuadro de lista), nos daban una función de miembro *AddString* y otra *InsertString* para las tareas que anteriormente se realizaban al enviar los mensajes *LB_ADDSTRING* y *LB_INSERTSTRING*.

Sin embargo, para ayudar a desarrollar un lenguaje mejor y más intuitivo, el equipo de diseño del lenguaje C# observó esto y se preguntó: «¿Por qué no tener la habilidad de tratar un objeto que es en el fondo un array como un array?». Cuando se considera un cuadro de lista, ¿no es un array de cadenas con la funcionalidad adicional de mostrarse y ordenarse? De esta idea nació el concepto de indizadores.

Cómo definir indizadores

Dado que a las propiedades a veces se las conoce como «campos inteligentes» y a los indizadores se les llama «arrays inteligentes», tiene sentido que las propiedades y los indizadores compartan la misma sintaxis. De hecho, definir indizadores es muy parecido a definir propiedades, con dos diferencias importantes: la primera, los indizadores toman un argumento *índice*; la segunda, como la propia clase se está utilizando como un array, la palabra reservada *this* se utiliza como nombre del indizador. En breve veremos un ejemplo completo, pero echemos antes un vistazo a un ejemplo de indizador:

```
class MyClass
{
    public object this [int idx]
    {
        .
    }
}
```

```

        get
        {
            //Devolver el dato deseado.
        }
        set
        {
            //Asignar el dato deseado.
        }
    }
    ...
}

```

No se ha mostrado un ejemplo completo para ilustrar la sintaxis de los indizadores, porque la implementación real interna de cómo definir los datos y cómo leer y asignar esos datos es irrelevante para los indizadores. Tenga en cuenta que, con independencia de cómo se almacenen los datos internamente (esto es, como un array, una colección y demás), los indizadores son simplemente un medio para que el programador que instancie la clase escriba código como este:

```

MyClass cls = new MyClass();
cls[0] = someObject;
Console.WriteLine("{0}", cls[0]);

```

Lo que se haga en el indizador es cosa de cada uno, siempre y cuando el cliente de la clase obtenga los resultados esperados al acceder al objeto como un array.

Ejemplo de indizador

Vamos a echar un vistazo en algunos lugares donde los indizadores tienen mucho sentido. Empezaremos con el ejemplo de cuadro de lista que ya hemos utilizado. Como ya se mencionó, desde un punto de vista conceptual, un cuadro de lista es simplemente una lista o un array de cadenas que se ha de mostrar. En el siguiente ejemplo se ha declarado una clase llamada *MyListBox* que contiene un indizador para asignar y leer cadenas mediante un objeto *ArrayList*. (La clase *ArrayList* es una clase del .NET Framework utilizada para almacenar una colección de objetos).

```

using System;
using System.Collections;

class MyListBox
{
    protected ArrayList data = new ArrayList();

    public object this [int idx]
    {
        get
        {
            if (idx > -1 && idx < data.Count)
            {
                return (data[idx]);
            }
        }
    }
}

```

```

    }
    else
    {
        //Posiblemente se lance una excepción aquí.
        return null;
    }
}
set
{
    if (idx > -1 && idx < data.Count)
    {
        data[idx] = value;
    }
    else if (idx == data.Count)
    {
        data.Add(value);
    }
    else
    {
        //Posiblemente se lance una excepción aquí.
    }
}
}
}

class Indexers1App
{
    public static void Main()
    {
        MyListBox lbx = new MyListBox();
        lbx [0] = "foo";
        lbx [1] = "bar";
        lbx [2] = "baz";
        Console.WriteLine("{0}{1}{2}", lbx[0], lbx[1], lbx[2]);
    }
}

```

Observe en este ejemplo que se comprueban errores de salida de rango en la indización de los datos. Esta comprobación está asociada técnicamente con los indizadores, porque, como ya se mencionó, a los indizadores sólo les concierne cómo el cliente de la clase puede utilizar el objeto como un array y no tienen nada que ver con la representación interna de los datos. Así, cuando aprendemos una característica nueva de un lenguaje, nos ayuda más ver el uso práctico de ésta que sólo su sintaxis. Por eso, en los dos métodos de lectura y asignación del indizador se valida el valor del índice que se pasa frente a los datos que están almacenados en el miembro de la clase *ArrayList*. Como consideración personal, seguramente escogería lanzar excepciones en los casos en los que el valor del índice que se está pasando no se puede resolver. Sin embargo, no deja de ser una elección personal —el manejo de errores que haríamos cada uno podría diferir. La clave es la necesidad de indicar el error al cliente en los casos en los que se ha pasado un índice no válido.

Guías de diseño

Los indizadores son también otro ejemplo de cómo el equipo de diseño de C# ha añadido una sutil y eficaz característica al lenguaje para ayudarnos a ser más productivos en nuestros esfuerzos de desarrollo. Sin embargo, como cualquier característica de cualquier lenguaje, los indizadores tienen un lugar. Deberían utilizarse sólo donde fuera intuitivo tratar un objeto como un array. Tomemos como ejemplo el caso de la facturación. Es razonable que una aplicación de facturación tenga una clase *Invoice* que defina un miembro array de objetos *InvoiceDetail*. En tal caso, sería muy intuitivo para el usuario poder acceder a las líneas de detalle con la siguiente sintaxis:

```
InvoiceDetail detail = invoice[2]; //Recupera la tercera línea
                                //de detalle.
```

Sin embargo, no sería intuitivo si damos un paso más e intentamos convertir todos los miembros de *InvoiceDetail* en un array al que se accedería mediante un indizador. Como se puede ver a continuación, la primera línea es mucho más legible e inteligible que la segunda:

```
TermCode terms = invoice.Terms; //Propiedad de acceso al miembro Terms.
TermCode terms = invoice[3];    //Una solución buscando un problema.
```

En este caso, la máxima es cierta porque se hace algo que no tendríamos que hacer necesariamente. O en términos más concretos, pensemos en cómo va a afectar al cliente de la clase implementar cualquier característica nueva, y dejemos que ese pensamiento nos guíe cuando decidamos si implementarla o no hará más fácil el utilizar la clase.

RESUMEN

Las propiedades de C# consisten en la declaración de campos y métodos de acceso. Las propiedades permiten un acceso inteligente a los campos de clase para que un programador que escribe un cliente para la clase no tenga que intentar determinar si (y cómo) se crea un método de acceso para el campo. Los arrays en C# se declaran colocando un corchete vacío *entre* el tipo y el nombre de variable, una sintaxis ligeramente distinta de la utilizada en C++. Los arrays de C# pueden ser unidimensionales, multidimensionales o irregulares. Los objetos en C# se pueden tratar como arrays mediante la utilización de los indizadores. Los indizadores permiten a los programadores trabajar y seguir la pista fácilmente a muchos objetos del mismo tipo.

Capítulo 8

Atributos

La mayoría de los lenguajes de programación se diseñan con un conjunto determinado de capacidades en mente. Por ejemplo, cuando se empieza a diseñar un compilador, se piensa en cómo se estructurará una aplicación escrita en el nuevo lenguaje, cómo el código invocará a otro código, cómo se empaquetará la funcionalidad, y muchos otros aspectos que harán del lenguaje un medio productivo para el desarrollo del software. La mayoría de lo que sugiere un diseñador de compiladores es estático. Por ejemplo, en C# se define una clase colocando la palabra reservada *class* antes del nombre de la clase. Después se indica la herencia insertando dos puntos después del nombre de la clase, seguido del nombre de la clase base. Este es un ejemplo de decisión que, una vez tomada por el diseñador del lenguaje, no puede cambiarse.

En la actualidad, la gente que escribe compiladores es extremadamente buena en lo que hace. Sin embargo, ni siquiera ellos pueden anticipar todos los desarrollos futuros en nuestra industria y cómo esos desarrollos alterarán el modo en que los programadores quieren expresar sus tipos en un determinado lenguaje. Por ejemplo, ¿cómo se crea la relación entre una clase en C++ y la URL de documentación de esa clase? O ¿cómo se asocian los miembros específicos de una clase C++ con los campos XML para la nueva solución «business-to-business» de la compañía? Como C++ se diseñó muchos años antes de la llegada de Internet y de lenguajes como XML, no es fácil realizar ninguna de esas tareas.

Hasta ahora, las soluciones a los problemas de este tipo suponían almacenar información extra en un archivo separado (DEF, IDL, etc.), información que después estaba vagamente asociada con el tipo o miembro en cuestión. Como el compilador no tiene conocimiento del archivo separado o de la relación basada en código generado entre la clase y el archivo, este enfoque normalmente se llama «solución desconectada». El principal problema es que la clase ya no es «autodescriptiva»; esto es, un usuario ya no puede mirar solamente la definición de la clase y saber todo sobre ésta. Una ventaja de un componente autodescriptivo es que el compilador y el entorno de ejecución pueden asegurarse de que se cumplen las reglas asociadas con el componente. Además, un componente autodescriptivo es más fácil de mantener, porque el desarrollador puede ver toda la información relacionada con el componente en un único lugar.

Este ha sido el modo de hacerlo en todo el mundo durante las décadas de la evolución del compilador. Los diseñadores del lenguaje intentan determinar lo que se necesitará que haga el lenguaje, diseñan el compilador con esas capacidades, y para bien o para mal, esas son las capacidades que se tienen hasta que llega otro compilador. Eso es lo que tenemos hasta ahora. C# ofrece un paradigma distinto, que proviene de la introducción de una característica llamada *atributos*.

PRESENTACIÓN DE LOS ATRIBUTOS

Lo que los atributos permiten hacer es bastante innovador. Proporcionan mecanismos genéricos para asociar información (como anotaciones) a los tipos definidos en C#. Se pueden utilizar atributos para definir información durante el tiempo de diseño (como información de la documentación), información del entorno de ejecución (como el nombre de una columna de base de datos para un campo), o incluso características del comportamiento en tiempo de ejecución (como si un miembro determinado es «transaccional»; esto es, capaz de participar en una transacción). Las posibilidades son infinitas, que es precisamente lo más importante. Dado que se puede crear un atributo basado en cualquier información que le guste, existe un mecanismo estándar para definir los atributos y para consultar en tiempo de ejecución tanto el miembro y el tipo como sus atributos asociados.

Un ejemplo ilustrará mejor cómo utilizar esta eficaz característica. Digamos que tenemos una aplicación que almacena algo de información en el Registro de Windows (*Windows Registry*). Una consideración de diseño sería decidir dónde almacenar la información de claves del Registro. En la mayoría de los entornos de desarrollo, esta información se almacena, típicamente, en un archivo de recursos o en constantes, o incluso se codifica directamente en las invocaciones a las API del Registro. Sin embargo, otra vez, lo que tenemos es una situación en la que una parte integral de una clase se está almacenando aparte del resto de la definición de la clase. Utilizando atributos, podríamos adjuntar esta información a los miembros de las clases de tal manera que tengamos un componente completamente autodescriptivo. He aquí un ejemplo de cómo se vería, teniendo en cuenta que ya hemos definido el atributo *RegistryKey* en alguna otra parte:

```
class MyClass
{
    [RegistryKey(HKEY_CURRENT_USER, "foo")]
    public int Foo;
}
```

Para asociar un atributo determinado a un tipo o miembro de C#, simplemente hay que especificar los datos del atributo entre paréntesis antes del tipo o miembro objetivo. En el ejemplo, asociamos un atributo llamado *RegistryKey* al campo *MyClass.Foo*. En tiempo de ejecución —que veremos en breve—, todo lo que tendremos que hacer es preguntar al campo por su clave del Registro y después utilizar ese valor para almacenar los datos en éste.

CÓMO DEFINIR ATRIBUTOS

En el ejemplo anterior, observe que la sintaxis utilizada para adjuntar un atributo a un tipo o miembro se parece un poco a la instanciación de una clase. Esto ocurre porque un atributo es realmente una clase derivada de la clase base *System.Attribute*.

Ahora vamos a desarrollar un poco el atributo *RegistryKey*:

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get {return hive;}
        set {hive = value;}
    }

    protected String valueName;
    public String ValueName
    {
        get {return valueName;}
        set {valueName = value;}
    }
}
```

Lo que hemos hecho es añadir un *enum* para los diferentes tipos de registros, un constructor para la clase atributo (que toma un tipo de Registro y un nombre de valor) y dos propiedades para el subárbol del Registro y el nombre de valor. Se puede hacer mucho más cuando definamos atributos, pero en este punto, dado que sabemos cómo definir y asociar atributos, sigamos adelante y aprendamos cómo preguntar por los atributos en tiempo de ejecución. De esa manera, tendremos un ejemplo con el que jugar, completo y que funcione. Una vez que hayamos hecho esto, nos trasladaremos a algunos de los aspectos más avanzados de la definición y asociación de atributos.

NOTA Observe que en los ejemplos los nombres de las clases de atributos se anexan con la palabra *Attribute*. Sin embargo, cuando se adjunta el atributo a un tipo o miembro, no se incluye el sufijo *Attribute*. Esto es un mecanismo abreviado gratuito hecho por los diseñadores del lenguaje C#. Cuando el compilador ve que se está asociando un atributo con un tipo o un miembro, buscará una clase derivada de *System.Attribute* con el nombre del atributo especificado. Si no se puede localizar la clase, el compilador añadirá *Attribute* al nombre del atributo específico y lo buscará. Por lo tanto, es una práctica común definir los nombres de las clases de atributo terminándolos con *Attribute* y después omitir esa parte del nombre.

CÓMO PREGUNTAR SOBRE ATRIBUTOS

Sabemos cómo definir un atributo derivándolo de *System.Attribute* y cómo adjuntarlo a un tipo o miembro. Y ahora, ¿qué? ¿Cómo podemos utilizar atributos en código? En otras palabras, ¿cómo podemos preguntar a un tipo o miembro por los atributos (y sus parámetros) que se han adjuntado?

Para preguntar a un tipo o miembro por sus atributos adjuntos, debemos utilizar 'reflection'. Reflection es un tema avanzado que se trata en el Capítulo 16, «Cómo obtener información sobre metadatos con Reflection»; así que sólo discutiremos lo justo para ilustrar lo que se necesita para recuperar información del atributo en el entorno de ejecución. Si quiere aprender más sobre 'reflection', consulte el Capítulo 16.

Reflection es una característica que permite determinar dinámicamente en tiempo de ejecución las características de tipo para una aplicación. Por ejemplo, podemos utilizar las API 'Reflection' del .NET Framework para iterar a través de los metadatos para un ensamblaje completo y generar una lista de todas las clases, tipos y métodos que se han definido para ese ensamblaje. Echemos un vistazo a algunos ejemplos de atributos y cómo se les preguntaría utilizando 'reflection'.

Atributos de clase

Cómo recuperar un atributo depende del tipo de miembro al que se le pregunte. Digamos que queremos definir un atributo que indicará el servidor remoto en el que se va a crear un objeto. Sin atributos, se guardaría esta información en una constante o en el archivo de recursos de la aplicación. Utilizando los atributos, se puede comentar la clase con su servidor remoto de la siguiente manera:

```
using System;

public enum RemoteServers
{
    JEANVALJEAN,
    JAVERT,
```

```

        COSETTE
    }

    public class RemoteObjectAttribute : Attribute
    {
        public RemoteObjectAttribute(RemoteServers Server)
        {
            this.server = Server;
        }

        protected RemoteServers server;
        public string Server
        {
            get
            {
                return RemoteServers.GetName(typeof(RemoteServers),
                                                this.server);
            }
        }
    }

    [RemoteObject(RemoteServers.COSETTE)]
    class MyRemotableClass
    {
    }

```

Para determinar el servidor en el que vamos a crear el objeto, utilizamos código como el siguiente:

```

class ClassAttrApp
{
    public static void Main()
    {
        Type type = typeof(MyRemotableClass);
        foreach (Attribute attr in type.GetCustomAttributes(true))
        {
            RemoteObjectAttribute remoteAttr =
                attr as RemoteObjectAttribute;
            if (null != remoteAttr)
            {
                Console.WriteLine("Crear este objeto en {0}.",
                                remoteAttr.Server);
            }
        }
    }
}

```

Como cabría esperar, el resultado de esta aplicación es el siguiente:

Crear este objeto en COSETTE.

Como todas las variaciones de este ejemplo utilizarán algo de código común, examinemos lo que va a pasar con relación a la 'reflection' y cómo devuelve el valor del atributo en tiempo de ejecución.

En la primera línea del método *Main* se observará la utilización del operador *typeof*:

```
Type type = typeof(MyRemotableClass);
```

Este operador devuelve el objeto *System.Type* asociado con el tipo que se le pasa como único argumento. Una vez que tenemos ese objeto, podemos empezar a consultarlo.

Hay dos cosas que se deben explicar con relación a la siguiente línea de código:

```
foreach (Attribute attr in type.GetCustomAttributes(true))
```

La primera es la invocación al método *Type.GetCustomAttributes*. Este método devuelve un array de tipos *Attribute*, que en este caso contendrá todos los atributos asociados a la clase llamada *MyRemotableClass*. La segunda es la instrucción *foreach*, que recorre el array devuelto, introduciendo cada valor en una variable (*attr*) de tipo *Attribute*.

La siguiente instrucción utiliza el operador *as* para intentar convertir la variable *attr* en un tipo *RemoteObjectAttribute*:

```
RemoteObjectAttribute remoteAttr = attr as RemoteObjectAttribute;
```

Entonces comprobamos si es un valor nulo (*null*), el resultado que se obtiene si falla el operador *as*. Si el valor no es nulo —lo que significa que la variable *remoteAttr* tiene un atributo válido asociado al tipo *MyRemotableClass*—, invocamos una de las propiedades de *RemoteObjectAttribute* para imprimir el nombre del servidor remoto:

```
if (null != remoteAttr)
{
    Console.WriteLine("Crear este objeto en {0}",
                      remoteAttr.Server);
}
```

Atributos de método

Ahora que ya hemos visto cómo trabajar con atributos de clase, vamos a ver cómo se utilizan atributos de método. Este tema está en una sección separada, porque el código de 'reflection' que se necesita para consultar un atributo de método es diferente del que se necesita para consultar un atributo de clase. En este ejemplo utilizaremos un atributo que se emplearía para definir un método transaccional:

```
using System;
using System.Reflection;

public class TransactionableAttribute : Attribute
{
    public TransactionableAttribute()
    {
    }
}
```

```

class TestClass
{
    [Transactional]
    public void Foo()
    {}

    public void Bar()
    {}

    [Transactional]
    public void Baz()
    {}
}

class MethodAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType("TestClass");
        foreach (MethodInfo method in type.GetMethods())
        {
            foreach (Attribute attr in method.GetCustomAttributes())
            {
                if (attr is TransactionalAttribute)
                {
                    Console.WriteLine("{0} es transaccional",
                                      method.Name);
                }
            }
        }
    }
}

```

La salida del código es la siguiente:

```

Foo es transaccional.
Baz es transaccional.

```

En este ejemplo concreto, la mera presencia de *TransactionalAttribute* será suficiente para decirle al código que el método decorado con este atributo puede participar en una transacción. Por eso se define sólo con un constructor sin parámetros, muy limitado y sin otros miembros.

TestClass se define entonces con tres métodos (*Foo*, *Bar* y *Baz*), donde dos de ellos (*Foo* y *Baz*) se definen como transaccionales. Observe que cuando se añade un atributo con un constructor que no tiene parámetros, no es necesario incluir abrir y cerrar los paréntesis.

Ahora llega lo divertido. Vamos a ver más de cerca cómo podemos preguntar a los métodos de una clase por los atributos de los métodos. Empezamos utilizando el método estático *GetType* de la clase *Type* para obtener un objeto *System.Type* para la clase *TestClass*:

```
Type type = Type.GetType("TestClass");
```

Después utilizamos el método *Type.GetMethods* para recuperar un array de objetos *MethodInfo*. Cada uno de esos objetos contiene la información de un método de la clase *TestClass*. Utilizando una instrucción *foreach*, recorreremos cada método:

```
foreach(MethodInfo method in type.GetMethods())
```

Ahora que tenemos el objeto *MethodInfo*, podemos utilizar el método *MethodInfo.GetCustomAttributes* para recuperar todos los atributos creados por el usuario para el método. De nuevo, utilizamos una instrucción *foreach* para recorrer el array de objetos devuelto:

```
foreach (Attribute attr in method.GetCustomAttributes(true))
```

Llegados a este punto del código, tenemos un atributo para un método. Ahora, utilizando el operador *is*, le preguntamos si es un atributo *TransactionableAttribute*. Si lo es, imprimimos el nombre del método:

```
if (attr is TransactionableAttribute)
{
    Console.WriteLine("{0} es transaccional",
                      method.Name);
}
```

Atributos de campo

En el último ejemplo de consulta de miembros para conocer sus atributos asociados, veremos cómo consultar los campos de una clase. Supongamos que tenemos una clase que contiene en algunos campos los valores que queremos guardar en el Registro. Para hacer esto, podríamos definir un atributo con un constructor que tomara como parámetros un *enum* representando el subárbol adecuado del Registro y una cadena que representara el nombre de valor del Registro. Podríamos consultar entonces el campo en tiempo de ejecución mediante su clave de Registro:

```
using System;
using System.Reflection;

public enum RegistryHives
{
    HKEY_CLASSES_ROOT,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}

public class RegistryKeyAttribute : Attribute
{
    public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
    {
```

```

        this.Hive = Hive;
        this.ValueName = ValueName;
    }

    protected RegistryHives hive;
    public RegistryHives Hive
    {
        get {return hive;}
        set {hive = value;}
    }

    protected String valueName;
    public String ValueName
    {
        get {return valueName;}
        set {valueName = value;}
    }
}

class TestClass
{
    [RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]
    public int Foo;

    public int Bar;
}

class FieldAttrApp
{
    public static void Main()
    {
        Type type = Type.GetType("TestClass");
        foreach(FieldInfo field in type.GetFields())
        {
            foreach (Attribute attr in field.GetCustomAttributes())
            {
                RegistryKeyAttribute registryKeyAttr =
                    attr as RegistryKeyAttribute;
                if (null != registryKeyAttr)
                {
                    Console.WriteLine
                        ("{0} se guardará en {1}\\\\\\\\{2}",
                         field.Name,
                         registryKeyAttr.Hive,
                         registryKeyAttr.ValueName);
                }
            }
        }
    }
}

```

No examinaremos todo este código porque parte está en el ejemplo anterior. Sin embargo, hay un par de detalles importantes. Primero observe que de la misma forma que se define un objeto *MethodInfo* para recuperar información de los métodos de un objeto de tipo, el objeto *FieldInfo* proporciona la misma funcionalidad para obtener información

de los campos de un objeto de un tipo. Como en el ejemplo anterior, empezamos obteniendo el objeto de tipo asociado con nuestra clase de prueba. Entonces iteramos a través del array *FieldInfo*, y por cada objeto *FieldInfo*, recorremos sus atributos hasta que encontramos el que estamos buscando: *RegistryKeyAttribute*. Si lo localizamos, imprimimos el nombre del campo y recuperamos del atributo los campos *Hive* y *ValueNames*.

PARÁMETROS DE ATRIBUTOS

En los ejemplos anteriores vimos cómo asociar atributos mediante sus constructores. Ahora veremos algunos problemas asociados a los constructores de atributos que no tratamos anteriormente.

Parámetros posicionales y parámetros con nombre

En el ejemplo *FieldAttrApp* de la sección anterior vimos un atributo llamado *RegistryKeyAttribute*. Su constructor era similar al siguiente:

```
public RegistryKeyAttribute(RegistryHives Hive, String ValueName)
```

Basado en esa signatura de constructor, el atributo se asoció a un campo de la siguiente manera:

```
[RegistryKey(RegistryHives.HKEY_CURRENT_USER, "Foo")]  
public int Foo;
```

Hasta ahora es bastante fácil. El constructor tiene dos parámetros que se utilizan para asociar ese atributo al campo. Sin embargo, podemos programarlo de manera más sencilla. Si el parámetro va a ser igual la mayoría del tiempo, ¿por qué hacer que el usuario de la clase lo escriba a cada momento? Podemos establecer valores por defecto utilizando parámetros posicionales y parámetros con nombre.

Los parámetros posicionales son parámetros del constructor del atributo. Son obligatorios y deben especificarse cada vez que se utilice el atributo. En el ejemplo *RegistryKeyAttribute* anterior, *Hive* y *ValueName* son los dos parámetros posicionales. Los parámetros con nombre no se definen realmente en el constructor del atributo; es más, son campos y propiedades no estáticos. Por lo tanto, los parámetros con nombre dan al cliente la posibilidad de establecer campos y propiedades de un atributo cuando el atributo se instancia, sin que se haya tenido que crear un constructor para cada combinación posible de campos y propiedades que el cliente pudiera querer establecer.

Cada constructor público puede definir una secuencia de parámetros posicionales. Esto es igual que en cualquier tipo de clase. Sin embargo, en el caso de los atributos, una vez que se han establecido los parámetros posicionales, el usuario puede entonces referirse a ciertos campos o propiedades con la sintaxis *NombreDeCampoOPropiedad=Valor*. Modifiquemos el atributo *RegistryKeyAttribute* para ilustrar esto. En este ejemplo, haremos de *RegistryKeyAttribute.ValueName* el único parámetro posicional, y

RegistryKeyAttribute.Hive se convertirá en un parámetro con nombre opcional. Así que la pregunta es: «¿Cómo se define un parámetro con nombre?». Como sólo se incluyen en la definición de constructor los parámetros posicionales —y por lo tanto obligatorios—, simplemente hay que quitar el parámetro de la definición de constructor. Entonces, el usuario puede hacer referencia como parámetro con nombre a cualquier campo que no sea *readonly*, *static* o *const*, o cualquier propiedad que incluya un método de acceso para asignación (*setter*), que no sea estático. Por lo tanto, para hacer de *RegistryKeyAttribute.Hive* un parámetro con nombre, lo quitaríamos de la definición del constructor porque ya existe como propiedad de lectura/escritura pública:

```
public RegistryKeyAttribute(String ValueName)
```

El usuario ya puede asociar el atributo de cualquiera de las siguientes maneras:

```
[RegistryKey("Foo")]
[RegistryKey("Foo", Hive = RegistryHives.HKEY_LOCAL_MACHINE)]
```

Esto le da la flexibilidad de tener un valor por defecto para un campo mientras al mismo tiempo se le da al usuario la posibilidad de redefinir ese valor si es necesario. Pero ¡sigamos! Si el usuario *no* establece el valor del campo *RegistryKeyAttribute.Hive*, ¿cómo hacemos que esté predeterminado? Se podría pensar: «Bien, vamos a comprobar si se ha asignado el valor en el constructor». Sin embargo, el problema es que el *RegistryKeyAttribute.Hive* es un *enum* con un tipo *int* subyacente —es un tipo valor. Esto significa que por definición ¡el compilador lo ha inicializado a 0! Si examinamos el valor *RegistryKeyAttribute.Hive* en el constructor y encontramos que es igual a 0, no sabemos si ese valor lo asignó el invocador a través de un parámetro con nombre o si el compilador lo inicializó porque es un tipo valor. Desgraciadamente, en este momento, la única manera que tenemos de afrontar este problema es cambiar el código de tal forma que el valor 0 no sea válido. Esto se puede hacer cambiando el *enum RegistryHives* de esta manera:

```
public enum RegistryHives
{
    HKEY_CLASSES_ROOT = 1,
    HKEY_CURRENT_USER,
    HKEY_LOCAL_MACHINE,
    HKEY_USERS,
    HKEY_CURRENT_CONFIG
}
```

Ahora sabemos que la única manera en que *RegistryKeyAttribute.Hive* puede ser 0 es si el compilador lo inicializa a 0 y el usuario no redefine ese valor mediante un parámetro con nombre. Ahora podemos escribir código parecido al siguiente para inicializarlo:

```
public RegistryKeyAttribute(String ValueName)
{
    if (this.Hive == 0)
        this.Hive = RegistryHives.HKEY_CURRENT_USER;
```

```

    this.ValueName = ValueName;
}

```

Errores comunes con parámetros con nombre

Cuando utilizamos parámetros con nombre, debemos especificar primero los parámetros posicionales. Después, los parámetros con nombre se pueden definir en cualquier orden porque les precede el nombre del campo o de la propiedad. El siguiente ejemplo producirá un error de compilación:

```

//Esto es un error, ya que el parámetro posicional no puede seguir
//a un parámetro con nombre.
[RegistryKey(Hive=RegistryHives.HKEY_LOCAL_MACHINE, "Foo")]

```

Además, no podemos nombrar a los parámetros posicionales. Cuando el compilador está compilando la parte en la que se utiliza un atributo, intentará resolver primero los parámetros con nombre. Después intentará resolver lo que queda —los parámetros posicionales— con la signature del método. El siguiente ejemplo no compilará, porque aunque el compilador puede resolver los parámetros con nombre, cuando termina con éstos, no puede encontrar ningún parámetro posicional, y nos indica que no hay una sobrecarga del método *RegistryKeyAttribute* con 0 argumentos («*No overload for method 'RegistryKeyAttribute' takes '0' arguments*»):

```

[RegistryKey(ValueName="Foo", Hive=RegistryHives.HKEY_LOCAL_MACHINE)]

```

Por último, los parámetros con nombre pueden ser cualquier campo o propiedad accesibles públicamente —incluyendo un método de asignación— que no sea *static* o *const*.

Tipos de parámetros de atributo válidos

Los tipos de los parámetros posicionales y parámetros con nombre para una clase atributo están limitados a los tipos de parámetros que se listan a continuación:

- *bool*, *byte*, *char*, *double*, *float*, *int*, *long*, *short*, *string*.
- *System.Type*.
- *Object*.
- Un tipo *enum*, siempre que éste y cualquier tipo en el que esté anidado sean accesibles públicamente —como en el ejemplo utilizado con la enumeración *RegistryHives*.
- Un array unidimensional que implique cualquiera de los tipos listados.

Como los tipos de parámetros se limitan a los tipos de esta lista, no se pueden pasar a los constructores de atributos estructuras de datos como las clases. Esta restricción tiene sentido porque los atributos se asocian cuando realizamos el diseño y no tendríamos una

instancia de la clase (un objeto) en ese momento. Con los tipos válidos listados anteriormente, se pueden asociar los valores directamente en el código en tiempo de diseño, que es por lo que se pueden utilizar.

EL ATRIBUTO *ATTRIBUTEUSAGE*

Además de los atributos definidos por usuario que utilizamos para asociar información a los tipos normales de C#, podemos utilizar el atributo *AttributeUsage* para definir cómo queremos que se utilicen esos atributos. El atributo *AttributeUsage* tiene las siguientes convenciones de invocación documentadas:

```
[AttributeUsage(
    validon,
    AllowMultiple = allowmultiple,
    Inherited = inherited
)]
```

Como puede ver, es fácil distinguir lo que son parámetros posicionales y lo que son parámetros con nombre. Es muy recomendable documentar los atributos de esta manera, para que quien los utilice no tenga que buscar en el código fuente de la clase del atributo para encontrar los campos públicos y las propiedades de escritura/lectura que se pueden utilizar como atributos con nombre.

Cómo definir el objetivo de un atributo

Observando de nuevo el atributo *AttributeUsage* en la sección anterior, nos fijamos en que el parámetro *validon* es un parámetro posicional, y por lo tanto obligatorio. Este parámetro permite especificar los tipos a los que se puede asociar el atributo. En realidad, el parámetro *validon* del atributo *AttributeUsage* es del tipo *AttributeTargets*, que es una enumeración definida así:

```
public enum AttributeTargets
{
    Assembly      = 0x0001,
    Module        = 0x0002,
    Class         = 0x0004,
    Struct        = 0x0008,
    Enum          = 0x0010,
    Constructor   = 0x0020,
    Method        = 0x0040,
    Property      = 0x0080,
    Field         = 0x0100,
    Event         = 0x0200,
    Interface     = 0x0400,
    Parameter     = 0x0800,
    Delegate      = 0x1000,
```

```

All = Assembly | Module | Class | Struct | Enum | Constructor |
      Method | Property | Field | Event | Interface | Parameter |
      Delegate,
ClassMembers = Class | Struct | Enum | Constructor | Method |
               Property | Field | Event | Delegate | Interface,
}

```

Observe que cuando utilice el atributo *AttributeUsage* puede especificar *AttributeTargets.All*, de tal manera que el atributo se pueda asociar a cualquier tipo listado en la enumeración *AttributeTargets*. Este valor es el predeterminado si no se especifica el Atributo *AttributeUsage*. Dado que ese *AttributeTargets.All* es el valor por defecto, deberíamos preguntarnos por qué utilizaríamos alguna vez el valor *validon*. Bien, se pueden utilizar parámetros con nombre en este atributo, y podríamos querer cambiar uno de ellos. Recordemos que si utilizamos un parámetro con nombre, debemos prece-derlo de todos los parámetros posicionales. Esto nos da una manera fácil de especificar que queremos el valor por defecto del atributo *AttributeTargets.All* y además nos permite establecer los parámetros con nombre.

Por eso, ¿cuándo especificaríamos el parámetro *validon* (*AttributeTargets*) y por qué? Lo utilizamos cuando queremos controlar exactamente cómo se está utilizando un atributo. En los ejemplos anteriores, creamos un atributo *RemoteObjectAttribute* que fuera aplicable sólo para clases, un atributo *TransactionableAttribute* que se aplicara sólo a los métodos, y un atributo *RegistryKeyAttribute* que sólo tuviera sentido para los campos. Si quisiéramos estar seguros de que estos atributos se estaban utilizando para asociar información sólo a los tipos para los que se diseñaron, podríamos definirlos de la siguiente manera (se ha eliminado el código para una mayor brevedad):

```

[AttributeUsage(AttributeTargets.Class)]
public class RemoteObjectAttribute : Attribute
{
    ...
}

[AttributeUsage(AttributeTargets.Method)]
public class TransactionableAttribute : Attribute
{
    ...
}

[AttributeUsage(AttributeTargets.Field)]
public class RegistryKeyAttribute : Attribute
{
    ...
}

```

Un último aspecto con relación a la enumeración *AttributeTargets*: podemos combinar miembros utilizando el operador *|*. Si tenemos un atributo que se aplica tanto a campos como a propiedades, le asociaríamos el atributo *AttributeUsage* de la siguiente manera:

```

[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)]

```

Atributos de un solo uso y multiuso

Se puede utilizar *AttributeUsage* para definir atributos tanto de un solo uso como multiuso. Esta decisión depende de cuántas veces se puede utilizar un atributo simple en un único campo. Por defecto, todos los atributos son de un solo uso, lo que significa que compilar el siguiente ejemplo produciría un error de compilación:

```
using System;
using System.Reflection;

public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

//ERROR: Esto produce un error de compilación de "atributo duplicado".
[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
}

class SingleUseApp
{
    public static void Main()
    {
    }
}
```

Para resolver este problema, especificamos en la línea *AttributeUsage* que permitiremos que el atributo se asocie a un determinado tipo múltiples veces. El siguiente ejemplo funcionaría:

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.All, AllowMultiple=true)]
public class SingleUseAttribute : Attribute
{
    public SingleUseAttribute(String str)
    {
    }
}

[SingleUse("abc")]
[SingleUse("def")]
class MyClass
{
}

class MultiUseApp
{
}
```

```

    public static void Main()
    {
    }
}

```

Un ejemplo práctico de utilización de este enfoque es con el atributo *RegistryKeyAttribute*, tratado en el apartado «Cómo definir atributos». Dado que podemos encontrarnos con un campo que pueda guardarse en múltiples lugares en el Registro, asociaríamos un atributo *AttributeUsage* con el parámetro con nombre *AllowMultiple* como en el código descrito.

Cómo especificar las reglas de herencia de atributos

El último parámetro del atributo *AttributeUsage* es la opción *inherited*, que dictamina si el atributo se puede heredar. El valor por defecto es *false*. Sin embargo, si se asigna el valor *true* a la opción *inherited*, su significado depende del valor de la opción *AllowMultiple*. Si la opción *inherited* tiene un valor *true* y la opción *AllowMultiple* es *false*, el atributo redefinirá el atributo heredado. Sin embargo, si la opción *inherited* tiene un valor *true* y la opción *AllowMultiple* también, el atributo se añade al miembro.

IDENTIFICADORES DE ATRIBUTO

Echemos un vistazo a la siguiente línea de código, e intentemos averiguar si el atributo añade información al valor de retorno o al método:

```

class MyClass
{
    [HRESULT]
    public long Foo();
}

```

Si tenemos algo de experiencia con COM, sabemos que un *HRESULT* es el tipo de retorno estándar para todos los métodos que no se llamen *AddRef* o *Release*. Sin embargo, es fácil ver que si el nombre del atributo se pudiera aplicar tanto al valor de retorno como al nombre del método, sería imposible para el compilador saber cuál es la intención. A continuación se muestran algunos escenarios distintos donde el compilador no sabría la intención por el contexto:

- Método frente al tipo de retorno.
- Evento frente a campo frente a propiedad.
- Delegado frente a tipo de retorno.
- Propiedad frente a método de acceso frente a valor de retorno de un método de asignación frente al valor del parámetro de un método de asignación.

En cada uno de estos casos, el compilador toma una determinación basada en lo que se considera «más común». Para redefinir esta determinación, hay que utilizar un identificador de atributo de entre los siguientes:

- assembly (ensamblaje).
- module (módulo).
- type (tipo).
- method (método).
- property (propiedad).
- event (evento).
- field (campo).
- param (parámetro).
- return (valor de retorno).

Para utilizar un identificador de atributo, hay que anteponer al nombre del atributo el identificador y dos puntos. En el ejemplo de *MyClass*, si quisiéramos estar seguros de que el compilador determine que el atributo HRESULT asocia información al valor de retorno y no al método, lo especificaríamos como sigue:

```
class MyClass
{
    [return:HRESULT]
    public long Foo();
}
```

RESUMEN

Los atributos de C# proporcionan un mecanismo para asociar información a los tipos y miembros en tiempo de diseño para que se pueda recuperar en tiempo de ejecución mediante *'reflection'*. Esto nos da la posibilidad de crear componentes enteramente autocontenidos y autodescriptivos sin tener que recurrir a guardar información de relleno en archivos de recursos y constantes. La ventaja es un componente móvil que es más fácil de escribir y más fácil de mantener.

Capítulo 9

Interfaces

La clave para entender las interfaces podría ser compararlas con clases. Las clases son objetos que tienen propiedades y métodos que actúan sobre esas propiedades. Mientras que las clases exhiben algunas características de comportamiento (los métodos), las clases en realidad son *entidades*, un concepto que contrasta con el *comportamiento*, y debido a ello aparecen las interfaces. Las interfaces nos permiten definir características de comportamiento o capacidades, y aplican estos comportamientos a las clases sin tener en cuenta la jerarquía de éstas. Por ejemplo, digamos que tiene una aplicación de distribución en la que alguna de las entidades puede serializarse. Éstas podrían incluir las clases *Customer*, *Supplier* e *Invoice*. Otras clases, como *MaintenanceView* y *Document*, podrían definirse como no serializables. ¿Cómo haría serializables sólo aquellas clases que usted eligiera? Una forma obvia sería el crear una clase base llamada algo así como *Serializable*. Sin embargo, esta aproximación tiene una pega importante. Una herencia simple no funcionará porque no queremos que se compartan todos los comportamientos de la clase. C# no permite la herencia múltiple, por lo que no hay forma de hacer que una clase dada se derive de forma selectiva de varias clases. La respuesta a este problema son las interfaces. Las interfaces le otorgan la capacidad de definir un conjunto de métodos y propiedades relacionados semánticamente que puedan implementar unas clases seleccionadas con independencia de la jerarquía de clases.

Desde un punto de vista conceptual, las interfaces son los contratos entre dos trozos de código distintos. Esto es, una vez que se define una interfaz y una clase que implementa esa interfaz, los clientes de la clase tienen garantizado que la clase ha implementado todos los métodos que se han definido en la interfaz. Pronto lo veremos en algunos ejemplos.

En este capítulo, veremos por qué las interfaces son una parte tan importante del C# y de la programación basada en componentes en general. A continuación veremos cómo se declaran e implementan las interfaces en una aplicación C#. Finalmente, entraremos en más detalle en los aspectos específicos relativos a la utilización de interfaces y resolución de los problemas inherentes a la herencia múltiple y a la colisión de nombres.

NOTA Cuando define una interfaz y una clase específica en su definición que va a utilizar esa interfaz, decimos que la clase *implementa la interfaz* o *hereda de la interfaz*. Puede utilizar ambas expresiones y las verá utilizadas de forma intercambiable en otros textos. Personalmente, considero que *implementa* es semánticamente el término más correcto —las interfaces son comportamientos definidos y la clase se define como algo que implementa ese comportamiento, de forma opuesta a heredar de otra clase—, aunque ambos términos lo son.

LA UTILIZACIÓN DE LA INTERFAZ

Para entender dónde son útiles las interfaces, echemos un vistazo a un problema de programación tradicional en el desarrollo de Microsoft Windows que no utiliza una interfaz pero en el que dos trozos de código distinto tienen que comunicarse de forma genérica. Imagínese que trabaja para Microsoft y que es el programador jefe del equipo para el Panel de control. Tiene que proporcionar un mecanismo genérico para que cualquier programa de cliente (*applet*) se incorpore al Panel de control de forma que aparezca allí su icono y el usuario final pueda ejecutar el programa. Teniendo en cuenta que esta funcionalidad se diseñó antes de que se introdujera el modelo COM, ¿cómo proporcionaría un mecanismo que permitiera que cualquier aplicación futura se integrara en el Panel de control? La solución que se creó ha sido un estándar del desarrollo para Windows durante años.

Usted, como programador jefe del equipo del Panel de control, diseña y documenta la función (o funciones) que necesita que implemente la aplicación cliente, junto con algunas reglas. En el caso de los applets de cliente, Microsoft decidió que para escribir uno de ellos de Panel de control tenía que crear una biblioteca de vínculos dinámicos (DLL) que implementara y exportase una función llamada *CPIApplet*. También tenía que añadir al nombre de esta DLL una extensión .cpl y colocar la DLL en la carpeta Windows System32. (En Windows Me o Windows 98 será Windows\System32, y en Windows 2000 será WINNT\System32). Cuando se carga el Panel de control, carga todas las DLL de la carpeta System32 con la extensión .cpl (mediante la función *LoadLibrary*) y luego utiliza la función *GetProcAddress* para cargar la función *CPIApplet*, comprobando por lo tanto que se han seguido las reglas y que se puede comunicar adecuadamente con el Panel de control.

Como mencionamos, esto es un modelo de programación estándar de Windows para el manejo de situaciones en las que tiene una parte del código con la que quiere que se comunique en el futuro código desconocido de una forma genérica. Sin embargo, no es la solución más elegante del mundo y sin duda alguna tiene sus problemas. La mayor desventaja en esta técnica es que fuerza al cliente —el código de Panel de control, en este caso— a incluir un montón de código de validación. Por ejemplo, el Panel de control no podría asumir que cualquier archivo .cpl en la carpeta es una DLL Windows. Además, el Panel de control necesita comprobar que en la DLL están las funciones correctas y que éstas hacen lo que especifica la documentación. Aquí es donde aparece la interfaz. Las

interfaces le permiten crear los mismos arreglos contractuales entre trozos de código diferentes pero de una forma más orientada a objetos y flexible. Además, dado que las interfaces son parte del lenguaje C#, el compilador se asegura que cuando se define que una clase implementa una interfaz dada, la clase hace lo que dice.

En C#, una interfaz es un concepto de primer orden que declara un tipo referencia que incluye solamente declaraciones de métodos. Pero ¿qué queremos decir con concepto «de primer orden»? Este término significa que la característica en cuestión es una parte integrante e integral del lenguaje. En otras palabras, no ha sido algo que se ha añadido después de que se diseñara el lenguaje. Entremos ahora en más profundidad a los detalles de qué son y cómo se declaran las interfaces.

NOTA A los desarrolladores C++ que estén leyendo, una interfaz es básicamente una clase abstracta que sólo tiene métodos virtuales puros declarados, además de otros miembros tipo de las clases C#, tales como propiedades, eventos e indicadores.

CÓMO DECLARAR INTERFACES

Las interfaces pueden contener métodos, propiedades, indicadores y eventos —ninguno de los cuales se implementan propiamente en la interfaz. Veamos un ejemplo para entender cómo utilizar esta característica. Suponga que está diseñando un editor para su compañía que aloja distintos controles Windows. Está escribiendo el editor y las rutinas que validan los controles que los usuarios colocan en el formulario del editor. El resto del equipo está escribiendo los controles que alojará el formulario. Casi seguro, necesitará proporcionar algún tipo de validación al nivel de formulario. En los momentos adecuados —tales como cuando el usuario pide de forma explícita al formulario que valide todos los controles o cuando se está procesando éste—, el formulario podría recorrer todos los controles asociados y validar cada control, o mejor aún, decirle al control que se valide a sí mismo.

¿Cómo le proporcionaría a este control la capacidad de validación? Es en esta situación donde sobresalen las interfaces. A continuación mostramos una interfaz sencilla que contiene un único método llamado *Validate*:

```
interface IValidate
{
    bool Validate();
}
```

Ahora puede documentar el hecho de que si el control implementa la interfaz *Ivalidate*, se puede validar el control.

Examinemos un par de aspectos del fragmento de código anterior. En primer lugar, no es necesario que especifique un modificador de acceso como *public* en un método de la interfaz. En realidad, si se antepone un modificador de acceso a la declaración del método, se produce un error en tiempo de compilación. Esto se debe a que todos los

métodos de las interfaces son públicos. (Los desarrolladores C++ podrían darse cuenta de que dado que las interfaces, por definición, son clases abstractas, no es necesario declarar explícitamente que los métodos son *virtuales puros* mediante la adición de un `=0` a la declaración del método.

Además de métodos, las interfaces pueden definir propiedades, indizadores y eventos, tal y como mostramos a continuación:

```
interface IExampleInterface
{
    //Ejemplo de declaración de propiedad.
    int testProperty {get;}

    //Ejemplo de declaración de evento.
    event testEvent Changed;

    //Ejemplo de declaración de indizador.
    string this [int index] {get; set;}
}
```

CÓMO IMPLEMENTAR INTERFACES

Dado que las interfaces definen un contrato, cualquier clase que implemente una interfaz *tiene que definir todos y cada uno de los elementos de esa interfaz*, o el código no se compilará. Utilizando el ejemplo de *Ivalidate* de la sección anterior, una clase cliente sólo tendría que implementar los métodos de la interfaz. En el siguiente ejemplo, tenemos una clase base llamada *FancyControl* y una interfaz llamada *Ivalidate*. A continuación tenemos una clase, *MyControl*, que se deriva de *FancyControl* y que implementa *Ivalidate*. Observe la sintaxis y cómo se puede convertir el tipo del objeto *MyControl* a la interfaz *Ivalidate* para que hacer referencia a sus miembros.

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface IValidate
{
```

```

    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "mis datos de cuadrícula";
    }

    public bool Validate()
    {
        Console.WriteLine("Validando...{0}", data);
        return true;
    }
}

class InterfaceApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        //Invoca una función para colocar el control en el
        //formulario. Ahora, cuando el editor tiene que
        //validar el control, puede hacer lo siguiente:

        IValidate val = (IValidate)myControl;
        bool success = val.Validate();
        Console.WriteLine("La validación de '{0}' {1} fue correcta",
            myControl.data,
            (true == success ? " " : "no"));
    }
}

```

Utilizando las definiciones de clase e interfaz anteriores, el editor puede preguntar al control con relación a si éste implementa la interfaz *Ivalidate*. (Verá cómo realizar este paso en la siguiente sección). Si lo hace, el editor puede validar y luego invocar los métodos que implementa de la interfaz. Esto podría llevarle a preguntarse: «¿Por qué no defino simplemente una clase base que tenga una función virtual pura llamada *Validate* para utilizar con este editor? El editor sólo aceptaría controles que se derivaran de esta clase base, ¿no?».

Eso sería una solución que funciona, aunque estaría seriamente limitada. Digamos que crea sus propios controles y que todos derivan de esta hipotética clase base. En consecuencia, todos implementarán el método virtual *Validate*. Esto funcionará hasta el día en que encuentre un control realmente estupendo que quiere utilizar con el editor. Digamos que encuentra una cuadrícula que ha escrito otra persona. Como tal, no estará derivada de la clase base obligatoria del editor. En C++, la respuesta es utilizar herencia múltiple y derivar su cuadrícula de la cuadrícula desarrollada por un tercero y de la clase base del editor. Sin embargo, C# no permite herencia múltiple.

Mediante la utilización de interfaces, puede implementar características de comportamiento múltiples en una única clase. En C#, puede derivar de una única clase, y además

de la funcionalidad heredada, implementar tantas interfaces como necesite la clase. Por ejemplo, si quisiera que la aplicación del editor validara el contenido del control, enlazar el control con una base de datos y serializar el contenido del control en el disco duro, declararía su clase de la siguiente manera:

```
public class MyGrid : ThirdPartyGrid, IValidate,
                    ISerializable, IDataBound
{
    :
}
```

Como prometimos, la siguiente sección responde a la pregunta «¿Cómo puede saber un fragmento de código si una clase implementa una cierta interfaz?».

Cómo consultar la implementación mediante *is*

En el ejemplo *InterfaceApp* vio el siguiente código, que se utilizó para convertir el tipo de un objeto (*MyControl*) a uno de sus interfaces implementados (*IValidate*) para luego invocar uno de los miembros de la interfaz (*Validate*):

```
MyControl myControl = new MyControl();
:
IValidate val = (IValidate)myControl;
bool success = val.Validate();
```

¿Qué sucedería si un cliente intentara utilizar una clase como si ésta hubiera implementado un método que en realidad no implementa? El siguiente ejemplo compilará porque *ISerializable* es una interfaz válida. Sin embargo, en tiempo de ejecución, se lanzará una excepción *System.InvalidCastException*, porque *MyGrid* no implementa la interfaz *ISerializable*. La aplicación anularía su ejecución, a menos que se estuviera capturando explícitamente esa excepción.

```
using System;

public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}
```

```

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "mis datos de cuadrícula";
    }

    public bool Validate()
    {
        Console.WriteLine("Validando...{0}", data);
        return true;
    }
}

class IsOperator1App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = (ISerializable)myControl;

        //NOTA: Esto lanzará una excepción System.InvalidCastException
        //porque la clase no implementa la interfaz ISerializable.
        bool success = ser.Save();

        Console.WriteLine("Los datos de '{0}' {1} se ha guardado
                           satisfactoriamente", myControl.data,
                           (true == success ? "" : "no"));
    }
}

```

Por supuesto, el capturar la excepción no cambia el hecho de que el código que quiere ejecutar no lo hará en este caso. Lo que necesita es una forma de consultar el objeto *antes* de intentar convertir su tipo. Una forma de hacerlo es utilizando el operador *is*. El operador *is* le permite comprobar en tiempo de ejecución si un tipo es compatible con otro. Tiene la siguiente forma, donde *expresión* es un tipo referencia:

expresión is tipo

El operador *is* produce un valor booleano, y por lo tanto puede utilizarse en instrucciones condicionales. En el siguiente ejemplo, hemos modificado el código para comprobar la compatibilidad entre la clase *MyControl* y la interfaz *ISerializable* antes de utilizar un método de *ISerializable*:

```

using System;
public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{
    public MyControl()
    {
        data = "my grid data";
    }

    public bool Validate()
    {
        Console.WriteLine("Validando...{0}", data);
        return true;
    }
}

class IsOperator2App
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        if (myControl is ISerializable)
        {
            ISerializable ser = (ISerializable)myControl;
            bool success = ser.Save();

            Console.WriteLine("Los datos de '{0}' {1} se han guardado satisfactoriamente", myControl.data, (true == success ? "" : "no"));
        }
        else
    }
}

```

```

    {
        Console.WriteLine("No se implementa la interfaz ISerializable");
    }
}

```

Ahora que ha visto cómo el operador *is* le permite comprobar la compatibilidad de dos tipos para asegurar su correcta utilización, veamos uno de sus parientes cercanos —el operador *as*— y comparemos los dos.

Cómo consultar la implementación mediante *as*

Si echa un vistazo de cerca al código generado para el ejemplo anterior *IsOperator2App* —el código MSIL está a continuación de este párrafo—, se dará cuenta de un problema del operador *is*. Puede ver que *isinst* se invoca justo después de que se reserve memoria para el objeto y la pila esté lista. El código de operación *isinst* lo genera el compilador para el operador C# *is*. Este código de operación comprueba si el objeto es una instancia de una clase o interfaz. Observe que sólo unas líneas después —asumiendo que se pasan las comprobaciones condicionales— el compilador ha generado el código de operación *IL castclass*. Este código de operación hace su propia comprobación, y mientras este código de operación trabaja de forma ligeramente distinta de *isinst*, el resultado es que el IL generado está haciendo un trabajo ineficiente comprobando la validez de la conversión de tipo dos veces.

```

.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size          72 (0x48)
    .maxstack 4
    .locals (class MyControl V_0,
            class ISerializable V_1,
            bool V_2)
    IL_0000: newobj      instance void MyControl::.ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: isinst      ISerializable
    IL_000c: brfalse.s   IL_003d
    IL_000e: ldloc.0
    IL_000f: castclass   ISerializable
    IL_0014: stloc.1
    IL_0015: ldloc.1
    IL_0016: callvirt    instance bool ISerializable::Save()
    IL_001b: stloc.2
    IL_001c: ldstr      "Los datos de '{0}' {1} se han guardado
                    satisfactoriamente"
    IL_0021: ldloc.0
    IL_0022: call      instance class System.String FancyControl::get_data()
    IL_0027: ldloc.2
    IL_0028: brtrue.s    IL_0031
    IL_002a: ldstr      "no"
    IL_002f: br.s      IL_0036

```

```

IL_0031: ldstr      ""
IL_0036: call       void [mscorlib]System.Console::WriteLine(class
                                     System.String,
                                     class System.Object,
                                     class System.Object)
IL_003b: br.s        IL_0047
IL_003d: ldstr      "No se implementa la interfaz ISerializable."
IL_0042: call       void [mscorlib]System.Console::WriteLine(class
                                     System.String)
IL_0047: ret
} //end of method IsOperator2App::Main

```

Podemos hacer este proceso de comprobación más eficiente si utilizamos el operador *as*. Este operador convierte entre dos tipos compatibles y tiene el siguiente formato, donde *expresión* es cualquier tipo referencia:

objeto = expresión as tipo

Puede pensar en el operador *as* como una combinación del operador *is*, y si los dos tipos son compatibles, una conversión de tipo. Una diferencia importante entre el operador *as* y el operador *is* es que el operador *as* hace que objeto sea *null* en vez de devolver un valor booleano si *expresión* y *tipo* son incompatibles. Nuestro ejemplo se puede reescribir ahora de forma más eficiente:

```

using System;
public class FancyControl
{
    protected string Data;
    public string data
    {
        get
        {
            return this.Data;
        }
        set
        {
            this.Data = value;
        }
    }
}

interface ISerializable
{
    bool Save();
}

interface IValidate
{
    bool Validate();
}

class MyControl : FancyControl, IValidate
{

```

```

public MyControl()
{
    data = "my grid data";
}

public bool Validate()
{
    Console.WriteLine("Validando...{0}", data);
    return true;
}
}

class AsOperatorApp
{
    public static void Main()
    {
        MyControl myControl = new MyControl();

        ISerializable ser = myControl as ISerializable;
        if (null != ser)
        {
            bool success = ser.Save();

            Console.WriteLine("Los datos de '{0}' {1} se han guardado
                               satisfactoriamente",
                               myControl.data,
                               (true == success ? "" : "no"));
        }
        else
        {
            Console.WriteLine("No se implementa la interfaz ISerializable");
        }
    }
}

```

Ahora sólo se hace una vez la comprobación de que se ha realizado una conversión de tipo válida, lo que obviamente es mucho más eficiente. En este punto, volvamos a ver el código MSIL para ver el impacto que ha tenido la utilización del operador *as*:

```

.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size          67 (0x43)
    .maxstack 4
    .locals (class MyControl V_0,
             class ISerializable V_1,
             bool V_2)
    IL_0000: newobj      instance void MyControl::.ctor()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: isinst      ISerializable
    IL_000c: stloc.1
    IL_000d: ldloc.1
    IL_000e: brfalse.s IL_0038
    IL_0010: ldloc.1

```

```

IL_0011: callvirt instance bool ISerializable::Save()
IL_0016: stloc.2
IL_0017: ldstr      "Los datos de '{0}' {1} se han guardado
                  satisfactoriamente"

IL_001c: ldloc.0
IL_001d: call      instance class System.String FancyControl::get_data()
IL_0022: ldloc.2
IL_0023: brtrue.s   IL_002c
IL_0025: ldstr      "no"
IL_002a: br.s      IL_0031
IL_002c: ldstr      ""
IL_0031: call      void [mscorlib]System.Console::WriteLine(class
                                                         System.String,
                                                         class System.Object,
                                                         class System.Object)

IL_0036: br.s      IL_0042
IL_0038: ldstr      "No se implementa la interfaz ISerializable"
IL_003d: call      void [mscorlib]System.Console::WriteLine(class
                                                         System.String)

IL_0042: ret
} //end of method AsOperatorApp::Main

```

CALIFICACIÓN EXPLÍCITA DEL NOMBRE DEL MIEMBRO DE LA INTERFAZ

Hasta ahora hemos visto clases que implementan interfaces especificando el modificador de acceso *public* seguidas de la signatura del método de la interfaz. Sin embargo, a veces querrá (o incluso necesitará) indicar de forma explícita el nombre del método junto al nombre de la interfaz. En esta sección examinaremos dos razones comunes que nos llevan a esto.

Ocultando nombre con interfaces

La manera más común de invocar un método de una interfaz implementado por una clase es convertir el tipo de una instancia de la clase a la interfaz y luego invocar el método deseado. Mientras que esto es válido y mucha gente (incluyendo al autor) utiliza este procedimiento; desde un punto de vista técnico, no es necesario convertir el tipo del objeto a la interfaz que implementa para invocar el método de la interfaz. Esto se debe a que cuando una clase implementa los métodos de una interfaz, estos métodos son también métodos públicos de la clase. Eche un vistazo al siguiente código C# y especialmente al método *Main* para ver lo que queremos decir:

```

using System;

public interface IDataBound
{
    void Bind();
}

```

```

public class EditBox : IDataBound
{
    //Implementación de IDataBound.
    public void Bind()
    {
        Console.WriteLine("Enlazando con el almacén de datos...");
    }
}

class NameHiding1App
{
    //Punto de entrada principal.
    public static void Main()
    {
        Console.WriteLine();

        EditBox edit = new EditBox();
        Console.WriteLine("Invocando EditBox.Bind()...");
        edit.Bind();

        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Invocando (IDataBound)EditBox.Bind()...");
        bound.Bind();
    }
}

```

Si ejecutamos el ejemplo, obtendremos la siguiente salida:

```

Invocando EditBox.Bind()...
Enlazando con el almacén de datos...

Invocando (IDataBound)EditBox.Bind()...
Enlazando con el almacén de datos...

```

Observe que aunque esta aplicación invoca el método *Bind* implementado de dos formas diferentes —una con conversión de tipos y otra sin ella—, ambas invocan de manera adecuada la función y el método *Bind* se procesa. Aunque a primera vista la capacidad de invocar el método implementado sin necesidad de convertir el tipo del objeto en interfaz podría parecer algo bueno, a veces es poco deseable. La razón más obvia es que el implementar varios interfaces —cada uno conteniendo muchos miembros— podría corromper el espacio de nombres públicos de su clase con miembros que no tienen sentido fuera del ámbito de la clase que los implementa. Puede evitar que los miembros de las interfaces implementadas se conviertan en miembros públicos de la clase utilizando una técnica llamada *ocultación de nombres*.

La ocultación de nombres, descrita de la forma más sencilla posible, es la capacidad de ocultar el nombre de un miembro heredado a cualquier código fuera de la clase derivada o de la que lo implementa (comúnmente conocido como *mundo exterior*). Digamos que tenemos el mismo ejemplo que utilizamos anteriormente en el que una clase *EditBox* necesita implementar la interfaz *IDataBound*; sin embargo, esta vez la clase *EditBox* no quiere exponer los métodos de *IDataBound* al mundo exterior. En vez de eso, necesita la interfaz para sus propios objetivos, o quizás el programador simplemente no

quiere confundir el espacio de nombres con un gran número de métodos que un cliente típico no utilizará. Para ocultar un miembro de una interfaz implementada, necesita sólo eliminar el modificador de acceso *público* al miembro y calificar el nombre del método con el nombre de la interfaz, como mostramos aquí:

```
using System;

public interface IDataBound
{
    void Bind();
}

public class EditBox : IDataBound
{
    //Implementación de IDataBound.
    void IDataBound.Bind()
    {
        Console.WriteLine("Enlazando con el almacén de datos...");
    }
}

class NameHiding2App
{
    public static void Main()
    {
        Console.WriteLine();

        EditBox edit = new EditBox();
        Console.WriteLine("Invocando EditBox.Bind()...");

        //ERROR: Esta línea no compilará porque el método Bind
        //ya no existe en el espacio de nombres de la clase EditBox.
        edit.Bind();

        Console.WriteLine();

        IDataBound bound = (IDataBound)edit;
        Console.WriteLine("Invocando (IDataBound)EditBox.Bind()...");

        //Esto funciona porque primero se convirtió el tipo
        //del objeto a IDataBound.
        bound.Bind();
    }
}
```

El código anterior no compilará porque el nombre del miembro *Bind* ya no forma parte de la clase *EditBox*. Por lo tanto, esta técnica le permite eliminar el miembro del espacio de nombres de la clase mientras que aún permite el acceso explícito mediante una operación de conversión de tipo.

Un aspecto que querríamos reiterar es que cuando se oculta un miembro no se puede utilizar un modificador de acceso. Si intentara utilizar un modificador de acceso en un miembro implementado de una interfaz, se produciría un error en tiempo de compilación. Esto podría parecerle extraño, pero considere que la única razón para ocultar algo es el

evitar que sea visible fuera de la clase actual. Dado que los modificadores de acceso sólo existen para definir el nivel de visibilidad fuera de la clase base, no tienen sentido cuando utiliza ocultación de nombres.

Cómo evitar la ambigüedad de nombres

Una de las razones principales por las que C# no permite herencia múltiple es el problema de la colisión de nombres, que se debe a la ambigüedad de los nombres. Aunque C# no permite herencia múltiple al nivel de objeto (derivar de una clase), permite herencia de una clase y la implementación adicional de múltiples interfaces. Sin embargo, con esta potencia viene un problema: la colisión de nombres.

En el siguiente ejemplo tenemos dos interfaces, *ISerializable* e *IDataStore*, que permiten la lectura y almacenamiento de datos en dos formatos distintos: uno en formato binario como un objeto en el disco y el otro a una base de datos. El problema es que ambos contienen métodos llamados *SaveData*:

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    public void SaveData()
    {
        Console.WriteLine("Test.SaveData invocado");
    }
}

class NameCollisions1App
{
    public static void Main()
    {
        Test test = new Test();
        Console.WriteLine("Invocando Test.SaveData()");
        test.SaveData();
    }
}
```

En el momento de escribir el libro, este código ha compilado. Sin embargo, debemos decir que en una versión futura del compilador C#, el código producirá un error en tiempo de compilación dada la ambigüedad del método *SaveData*. Con independencia de si el código compila o no, tendría un problema en tiempo de ejecución porque el com-

portamiento resultado de la invocación del método *SaveData* no estaría claro para el programador que intentara utilizar la clase. ¿*SaveData*, serializaría el objeto en el disco o guardaría los datos en una base de datos?

Además de esto, eche un vistazo al siguiente código:

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    public void SaveData()
    {
        Console.WriteLine("Test.SaveData invocado");
    }
}

class NameCollisions2App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine("Se implementa ISerializable");
        }

        if (test is IDataStore)
        {
            Console.WriteLine("Se implementa IDataStore");
        }
    }
}
```

En el ejemplo, el operador *is* tiene éxito para ambos interfaces, lo que indica que se implementan ambos interfaces, ¡aunque sabemos que no es así! Incluso el compilador genera los siguientes avisos cuando se compila el ejemplo:

```
NameCollisions2.cs(27,7): warning CS0183: The given expression is
always of the provided ('ISerializable') type
(La expresión dada siempre es del tipo proporcionado ('ISerializable'))
NameCollisions2.cs(32,7): warning CS0183: The given expression is
always of the provided ('IDataStore') type
(La expresión dada siempre es del tipo proporcionado ('IDataStore'))
```

El problema es que la clase ha implementado o una versión serializada o una versión de base de datos del método *Bind* (pero no ambas). Sin embargo, si el cliente comprueba si se ha implementado una de las interfaces —se indica que se han implementado ambas— y a continuación intenta utilizar la que en realidad no se ha implementado, ocurrirán resultados inesperados.

Puede utilizar la calificación explícita de nombres de miembros para resolver este problema: eliminar el modificador de acceso y anteponer al nombre del miembro —*SaveData* en este caso— el nombre de la interfaz:

```
using System;

interface ISerializable
{
    void SaveData();
}

interface IDataStore
{
    void SaveData();
}

class Test : ISerializable, IDataStore
{
    void ISerializable.SaveData()
    {
        Console.WriteLine("Test.ISerializable.SaveData invocado");
    }
    void IDataStore.SaveData()
    {
        Console.WriteLine("Test.IDataStore.SaveData invocado");
    }
}

class NameCollisions3App
{
    public static void Main()
    {
        Test test = new Test();

        if (test is ISerializable)
        {
            Console.WriteLine("Se implementa ISerializable");
            ((ISerializable)test).SaveData();
        }

        Console.WriteLine();

        if (test is IDataStore)
        {
            Console.WriteLine("Se implementa IDataStore");
            ((IDataStore)test).SaveData();
        }
    }
}
```

Ahora no hay ambigüedad con respecto a qué método se invocará. Ambos métodos están implementados con sus nombres completamente calificados, y el resultado de la aplicación será lo que esperamos:

```
Se implementa ISerializable
Test.ISerializable.SaveData invocado
```

```
Se implementa IDataStore
Test.IDataStore.SaveData invocado
```

INTERFACES Y HERENCIA

Hay dos problemas asociados con las interfaces y la herencia. El primer problema, ilustrado a continuación con un programa de ejemplo, se refiere al hecho de derivar de una clase base que contiene un nombre de método idéntico al nombre de un método de la interfaz que la clase tiene que implementar.

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize invocado");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInhlApp
{
    public static void Main()
    {
        EditBox edit = new EditBox();
        edit.Serialize();
    }
}
```

Como sabe, para implementar una interfaz debe proporcionar una definición para cada miembro de la declaración de la interfaz. Sin embargo, en el ejemplo anterior no hacemos eso ¡y el código compila! La razón de que compile es que el compilador de C# busca un método *Serialize* implementado en la clase *EditBox* y encuentra uno. Sin embargo, el compilador se equivoca al determinar que este es el método implementado. El método *Serialize* que encuentra el compilador es el método *Serialize* que se hereda de la

clase *Control* y no una implementación real del método *IDataBound.Serialize*. Por lo tanto, aunque el código se compila, no funciona como esperamos, como veremos a continuación.

Ahora haremos las cosas un poco más interesantes. Observe que el siguiente código comprueba primero —mediante el operador *as*— que la interfaz se implementa y luego intenta invocar un método *Serialize* implementado. El código compila y funciona. Sin embargo, como sabemos, la clase *EditBox* no implementa realmente un método *Serialize* como resultado de la herencia de *IDataBound*. La clase *EditBox* ya tiene un método *Serialize* (heredado) de la clase *Control*. Esto significa que con toda probabilidad el cliente no va a obtener los resultados esperados.

```
using System;

public class Control
{
    public void Serialize()
    {
        Console.WriteLine("Control.Serialize invocado");
    }
}

public interface IDataBound
{
    void Serialize();
}

public class EditBox : Control, IDataBound
{
}

class InterfaceInh2App
{
    public static void Main()
    {
        EditBox edit = new EditBox();

        IDataBound bound = edit as IDataBound;
        if (bound != null)
        {
            Console.WriteLine("Se soporta IDataBound...");
            bound.Serialize();
        }
        else
        {
            Console.WriteLine("NO se soporta IDataBound...");
        }
    }
}
```

Otro problema potencial que se debe controlar es el que sucede cuando una clase derivada tiene un método con el mismo nombre que la implementación de un método de interfaz de la clase base. Veamos esto en un ejemplo también:

```

using System;

interface ITest
{
    void Foo();
}

//Base implementa ITest.
class Base : ITest
{
    public void Foo()
    {
        Console.WriteLine("Base.Foo (implementación de ITest)");
    }
}

class MyDerived : Base
{
    public new void Foo()
    {
        Console.WriteLine("MyDerived.Foo");
    }
}

public class InterfaceInh3App
{
    public static void Main()
    {
        MyDerived myDerived = new MyDerived();
        myDerived.Foo();

        ITest test = (ITest)myDerived;
        test.Foo();
    }
}

```

Este código produce el siguiente resultado en pantalla:

```

MyDerived.Foo
Base.Foo (implementación de ITest)

```

En esta situación, la clase *Base* implementa la interfaz *ITest* y su método *Foo*. Sin embargo, la clase *MyDerived* deriva de *Base* como una clase nueva e implementa un nuevo método *Foo* para esa clase. ¿Qué *Foo* se invoca? Depende de la referencia que tengamos. Si tenemos una referencia a un objeto *MyDerived*, se invocará su método *Foo*. Esto se debe a que, aunque el objeto *MyDerived* tiene una implementación heredada de *ITest.Foo*, el entorno de ejecución ejecutará el método *MyDerived.Foo* porque la palabra reservada *new* especifica una redefinición del método heredado.

Sin embargo, si explícitamente hacemos una conversión de tipo del objeto *myDerived* a una interfaz *ITest*, el compilador hace referencia a la implementación de la interfaz. La clase *MyDerived* tiene un método del mismo nombre, pero no es esto lo que está buscando el compilador. Cuando se convierte el tipo de un objeto a una interfaz, el compilador recorre el árbol de herencia hasta que se encuentra una clase que contiene la interfaz en

su lista base. Este es el motivo por el que las dos últimas líneas de código del método *Main* concluyen con la invocación del método *Foo* de la interfaz *ITest* implementada.

Por suerte, algunas de estas dificultades potenciales que implican colisión de nombres y herencia de interfaces han apoyado nuestra recomendación inicial: convierta siempre el tipo del objeto a la interfaz cuyo miembro está intentando utilizar.

CÓMO COMBINAR INTERFACES

Otra característica eficaz de C# es la capacidad de combinar dos o más interfaces juntas de forma que una clase sólo tenga que implementar el resultado combinado. Por ejemplo, supongamos que queremos crear una nueva clase *TreeView* que implemente las interfaces *IDragDrop* e *ISortable*. Dado que es razonable asumir que otros controles, como *List-View* y *ListBox*, también querrían combinar estas características, podría querer combinar las interfaces *IDragDrop* e *ISortable* en una única interfaz:

```
using System;

public class Control
{
}

public interface IDragDrop
{
    void Drag();
    void Drop();
}

public interface ISerializable
{
    void Serialize();
}

public interface ICombo : IDragDrop, ISerializable
{
    //Esta interfaz no añade nada nuevo en términos
    //de comportamiento, ya que su único propósito es
    //combinar las interfaces IDragDrop e ISerializable
    //en una única interfaz.
}

public class MyTreeView : Control, ICombo
{
    public void Drag()
    {
        Console.WriteLine("MyTreeView.Drag invocado");
    }

    public void Drop()
    {
        Console.WriteLine("MyTreeView.Drop invocado");
    }
}
```

```
        public void Serialize()
        {
            Console.WriteLine("MyTreeView.Serialize invocado");
        }
    }

    class CombiningApp
    {
        public static void Main()
        {
            MyTreeView tree = new MyTreeView();
            tree.Drag();
            tree.Drop();
            tree.Serialize();
        }
    }
}
```

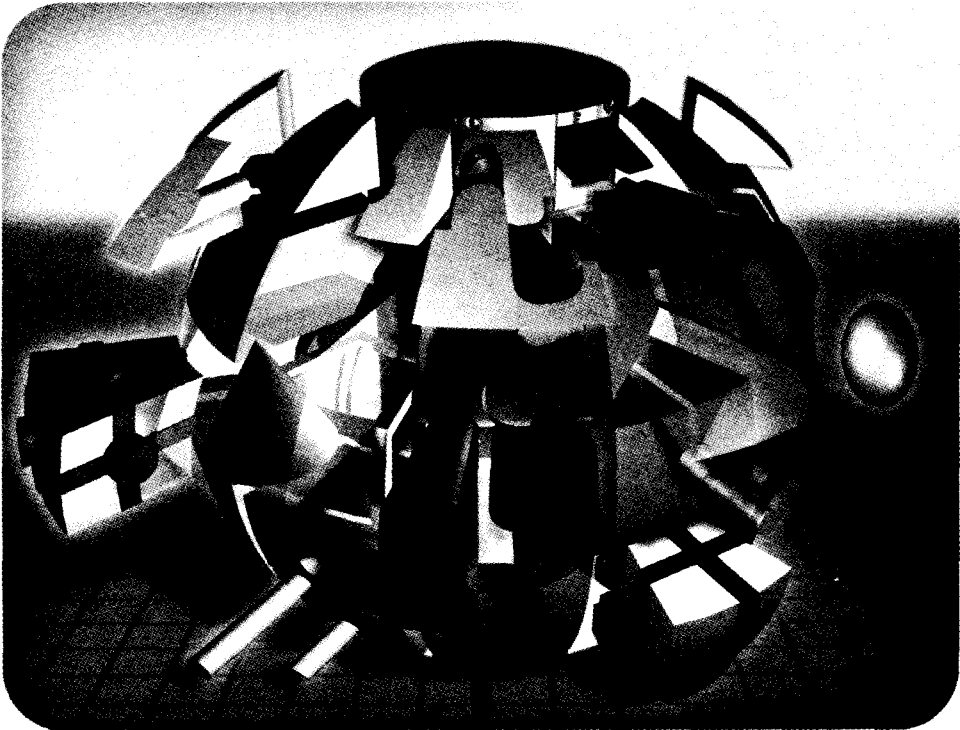
Con la habilidad de combinar interfaces, no sólo podemos simplificar la capacidad de agregar interfaces relacionadas semánticamente en una única interfaz, sino que podemos añadir métodos adicionales a la nueva interfaz «compuesta», si fuera necesario.

RESUMEN

Las interfaces en C# permiten el desarrollo de clases que pueden compartir características pero que no forman parte de la misma jerarquía de clases. Las interfaces juegan un papel especial en el desarrollo de C# porque éste no permite herencia múltiple. Para compartir métodos y propiedades relacionados semánticamente, las clases pueden implementar múltiples interfaces. También los operadores *is* y *as* se pueden utilizar para determinar si una interfaz particular es implementada por un objeto, algo que ayuda a evitar errores asociados a la utilización de miembros de la interfaz. Finalmente, se puede utilizar el nombrado explícito de miembros y la ocultación de nombres para controlar la implementación de interfaz y para ayudar a evitar errores.

Parte III

CÓMO ESCRIBIR CÓDIGO



Capítulo 10

Expresiones y operadores

En este capítulo echaremos un vistazo a la parte más básica de cualquier lenguaje de programación: su capacidad de expresar asignaciones y comparaciones mediante la utilización de operadores. Veremos qué son los operadores y cómo se determina la precedencia de los operadores en C#, para, a continuación, profundizar en las categorías específicas de expresiones que nos permiten llevar a cabo cosas tales como operaciones matemáticas, asignación de valores y realizar comparaciones entre operandos.

DEFINICIÓN DE OPERADORES

Un operador es un símbolo que indica una operación que se va a realizar sobre uno o más parámetros. Esta operación producirá un resultado. La sintaxis que se utiliza con los operadores es un poco distinta de la que se utiliza para la invocación de métodos, aunque el formato en C# de una expresión que utiliza operadores *le debería ser completamente natural*. Al igual que los operadores en la mayoría de los lenguajes de programación, la semántica de los operadores de C# sigue las reglas y notaciones básicas que aprendimos cuando éramos niños en las clases de matemáticas. Los operadores más básicos de C# incluyen la multiplicación (*), división (/), suma y signo unitario positivo (+), resta y signo unitario negativo (-), módulo (%) y asignación (=).

Los operadores están especialmente diseñados para producir un nuevo valor a partir de los valores con los que está operando. Los valores sobre los que se opera se llaman *operandos*. El resultado de una operación se debe almacenar en algún lugar de la memoria. En algunos casos, el valor que se ha generado a partir de la operación se almacena en la variable que contenía uno de los operandos originales. El compilador de C# genera un mensaje de error si se utiliza un operador y no se puede generar o almacenar un nuevo valor. En el siguiente ejemplo, el código termina sin que se haya producido ningún cambio. El compilador genera un mensaje de error, porque si se escribe una operación aritmética que no produce un cambio de al menos un valor, generalmente se debe a un defecto achacable al desarrollador.

```

class NoResultApp
{
    public static void Main()
    {
        int i;
        int j;
        i + j; //Error, ya que el resultado no se asigna a una variable.
    }
}

```

La mayoría de los operadores trabaja sólo con tipos de datos numéricos, como *Byte*, *Short*, *Long*, *Integer*, *Single*, *Double* y *Decimal*. Los operadores de comparación (`==` y `!=`) y el operador de asignación (`=`) son excepciones, ya que también pueden trabajar con objetos. Además, C# define los operadores `+` y `+=` para la clase *String*, e incluso permite el uso de operadores de incremento y decremento (`++`) y (`--`) en construcciones únicas del lenguaje, como son los delegados. Entraremos en este último ejemplo en el Capítulo 14, «Delegados y manejadores de eventos».

PRECEDENCIA DE OPERADORES

Cuando una expresión o instrucción simple contiene múltiples operadores, el compilador debe determinar el orden en el que se evaluarán los diferentes operadores. Las reglas que gobiernan la manera en que el compilador toma esta decisión se conocen como *precedencia de operadores*. El comprender la precedencia de los operadores puede suponer la diferencia entre escribir las expresiones con confianza y el mirar fijamente una simple línea de código a la vez que se pregunta por qué el resultado no es el que piensa que debería ser.

Eche un vistazo a la siguiente expresión: $42+6*10$. Si suma 42 y 6 y luego multiplica el resultado de la suma por 10, el producto es 480. Si multiplica $6*10$ y luego suma el resultado del producto con 42, el resultado es 102. Cuando se compila el código, una parte especial del compilador llamada *analizador léxico* se encarga de determinar cómo se debería leer el código. Es el analizador léxico el que determina la precedencia relativa de los operadores cuando está presente más de una clase de operadores en la expresión. Para conseguir esto, especifica un valor (o precedencia) para cada operador que admite. Los operadores con una precedencia se resuelven primero. En nuestro ejemplo, el operador `*` tiene una precedencia mayor que el operador `+`, porque `*` *toma* sus operandos —profundizaremos en este término en un momento— antes que `+`. La razón para esto se debe a las reglas básicas de la aritmética: la multiplicación y la división *siempre* tienen mayor precedencia que la suma y la resta. Teniendo en cuenta esto, volvamos al ejemplo: decimos que `*` *toma* el operador 6 tanto en $42+6*10$ como en $42*6+10$, de manera que estas expresiones son equivalentes a $42+(6*10)$ y $(42*6)+10$.

Cómo determina C# la precedencia

Veamos específicamente cómo asigna C# la precedencia a los operadores. La Tabla 10.1 ilustra la precedencia de los operadores en C# desde la precedencia mayor a la menor. Después de esta sección, entraré con más detalle en las diferentes categorías de operadores que permite C#.

Tabla 10.1. Precedencia de operadores de C#

Categoría del operador	Operadores
Primario	(x), x.y, f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked
Unitario	+, -, !, ~, ++x, --x, (T)x
Multiplicativo	*, /, %
Aditivo	+, -
De desplazamiento	<<, >>
Relacional	<, >, <=, >=, is
De igualdad	==
Y lógica	&
O exclusiva lógica	^
O lógica	
Y condicional	&&
O condicional	
Condicional	?:
De asignación	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =

Asociatividad por la izquierda y por la derecha

La asociatividad determina qué lado de la expresión se debería evaluar primero. Como ejemplo, la siguiente expresión podría dar como resultado final 21 o 33 dependiendo en la asociatividad por la derecha o por la izquierda del operador `-`:

42 - 15 - 6

El operador `-` se define como asociativo por la izquierda, lo que significa que `45-15` se evalúa primero y que 6 se restará del resultado. Si el operador `-` se definiera como asociativo por la derecha, la expresión a la derecha del operador se habría evaluado primero: se evaluaría `15-6` y luego se restaría de 42.

Todos los operadores binarios —esto es, los operadores que toman dos operandos—, excepto el operador de asignación, se consideran *asociativos por la izquierda*, lo que significa que la expresión se evalúa de izquierda a derecha. Por lo tanto, $a+b+c$ es lo mismo que $(a+b)+c$, donde $a+b$ se evalúa primero y c se suma con el resultado de la operación anterior. El operador de asignación y los operadores condicionales se consideran *asociativos por la derecha* —la expresión se evalúa desde la derecha hacia la izquierda. En otras palabras, $a=b=c$ es equivalente a $a=(b=c)$. Esto induce a error a algunas personas que quieren realizar asignaciones múltiples en la misma línea, así que veamos el siguiente código:

```
using System;

class RightAssocApp
{
    public static void Main()
    {
        int a = 1;
        int b = 2;
        int c = 3;

        Console.WriteLine("a={0} b={1} c={2}", a, b, c);
        a = b = c;
        Console.WriteLine("Después 'a=b=c' : a={0} b={1} c={2}", a, b, c);
    }
}
```

Los resultados de ejecutar el ejemplo son los siguientes:

```
a=1 b=2 c=3
Después 'a=b=c' : a=3 b=3 c=3
```

Ver una expresión que se evalúa desde la derecha puede ser un poco confuso al principio, pero piense en ello de la siguiente manera: si el operador de asignación fuera asociativo por la izquierda, el compilador primero evaluaría $a=b$, lo que daría un valor de 2 a la variable a , y luego evaluaría $b=c$, lo que daría un valor de 3 a la variable b . El resultado final sería $a=2$ $b=3$ $c=3$. Evidentemente, ese no sería el resultado esperado para $a=b=c$, y a esto se debe que los operadores de asignación y condicional sean ambos asociativos por la derecha.

Utilización práctica

Nada es más insidioso que perseguir un defecto durante un período largo de tiempo para terminar dándonos cuenta de que se debía a que un desarrollador no conocía las reglas de precedencia o las de asociatividad. Se han encontrado distintos mensajes en listas de correo en los que gente inteligente ha sugerido una convención de programación en la que los espacios se utilizan para indicar qué operadores *se piensa* que tienen precedencia, una especie de mecanismo de autodocumentación. Así, por ejemplo, como sabemos que

el operador de multiplicación tiene precedencia sobre el operador de suma, podríamos escribir código de forma parecida al siguiente ejemplo en el que los espacios en blanco indican la precedencia *pretendida*:

```
a = b*c + d;
```

Esta aproximación está muy penalizada —el compilador no hace un análisis del código de forma adecuada sin una sintaxis específica. El compilador analiza las expresiones basándose en reglas que determina la gente que lo ha desarrollado. A este efecto, existen unos símbolos que nos permiten fijar la precedencia y asociatividad: los paréntesis. Por ejemplo, se puede reescribir la expresión $a=b*c+d$ bien como $a=(b*c)+d$ o como $a=b*(c+d)$, y el compilador evaluará lo que está dentro del paréntesis primero. Si existen dos o más pares de paréntesis, el compilador evalúa el valor en cada par y luego la instrucción completa utilizando las reglas de precedencia y asociatividad que se han descrito.

Mi firme opinión es que siempre se deberían utilizar paréntesis cuando se utilizan varios operadores en una única expresión. Yo recomendaría esto incluso cuando se conoce el orden de precedencia, ya que los encargados de mantener el código podrían no estar tan bien informados.

OPERADORES EN C#

Es útil pensar en los operadores según su precedencia. En esta sección describiremos los operadores utilizados más comúnmente en el orden en que los presentamos en la Tabla 10.1.

Operadores primarios de expresión

La primera categoría de operadores que estudiaremos son los operadores primarios de expresión. Dado que la mayoría de los operadores son bastante básicos, solamente los listaremos y daremos una corta descripción sobre su función. A continuación explicaremos otros operadores menos obvios de forma más completa.

- (x) Esta forma del operador paréntesis se utiliza para controlar la precedencia, bien en operaciones matemáticas o en invocaciones a métodos.
- x.y El operador «punto» se utiliza para especificar un miembro de una clase o estructura, donde x representa la entidad contenedora e y representa la entidad miembro.
- f(x) Esta forma de los operadores paréntesis se utiliza para enumerar los parámetros de un método.
- a[x] Los corchetes se utilizan para indizar en un array. También se utilizan en conjunción con indizadores en los que los objetos se pueden tratar como arrays. Para más información sobre indizadores, diríjase al Capítulo 7, «Propiedades, arrays e indizadores».

- **x++** Dado que hay mucho que hablar de este operador, el operador de incremento se cubrirá más adelante en este mismo capítulo, en «Operadores de incremento y de decremento».
- **x--** Del operador de decremento hablaremos también más adelante en este capítulo.
- **new** El operador `new` se utiliza para instanciar objetos a partir de definiciones de clase.

typeof

La *reflexión* es la capacidad de recuperar la información de tipo en tiempo de ejecución. Esta información incluye nombres de tipos, nombres de clases y miembros de estructura. *Dentro de esta capacidad, tenemos en el Framework .NET una clase llamada `System.Type`.* Esta clase es la raíz de todas las operaciones de reflexión y se puede conseguir utilizando el operador `typeof`. No vamos a entrar con profundidad al detalle de la reflexión aquí —esta tarea la dejaremos para el Capítulo 16, «Cómo obtener información sobre metadatos con Reflection»—, pero mostraremos un ejemplo sencillo para ilustrar lo fácil que es utilizar el operador `typeof` para recuperar casi cualquier información que desee de un tipo u objeto en tiempo de ejecución:

```
using System;
using System.Reflection;

public class Apple
{
    public int nSeeds;
    public void Ripen()
    {
    }
}

public class TypeOfApp
{
    public static void Main()
    {
        Type t = typeof(Apple);
        string className = t.ToString();

        Console.WriteLine("\nInformación de la clase {0}", className);

        Console.WriteLine("\nMétodos de {0}", className);
        Console.WriteLine("-----");
        MethodInfo[] methods = t.GetMethods();
        foreach (MethodInfo method in methods)
        {
            Console.WriteLine(method.ToString());
        }

        Console.WriteLine("\nTodos los miembros de {0}", className);
        Console.WriteLine("-----");
        MemberInfo[] allMembers = t.GetMembers();
```

```

        foreach (MemberInfo member in allMembers)
        {
            Console.WriteLine(member.ToString());
        }
    }
}

```

El programa contiene una clase llamada *Apple* que tiene sólo dos miembros: un campo llamado *nSeeds* y un método llamado *Ripen*. Lo primero que hacemos es utilizar el operador *typeof* y el nombre de clase para obtener un objeto *System.Type*, que a continuación almacenamos en una variable llamada *t*. A partir de ese momento, podemos utilizar los métodos de *System.Type* para obtener todos los métodos y miembros de la clase *Apple*. Estas tareas se realizan mediante la utilización de los métodos *GetMethods* y *GetMembers*, respectivamente. Los resultados de estos métodos se imprimen en la salida estándar, según se muestra a continuación:

Información de la clase Apple

Métodos de Apple

```

-----
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()

```

Todos los miembros de Apple

```

-----
Int32 nSeeds
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void Ripen()
System.Type GetType()
Void .ctor()

```

Antes de seguir, hay un par de puntos adicionales que nos gustaría explicar. En primer lugar, observemos que los miembros de clase heredados también se muestran en la salida. Dado que no derivamos explícitamente de otra clase, podemos inferir que todos los miembros que no están definidos en la clase *Apple* se heredan de la clase base implícita, *System.Object*. En segundo lugar, observemos que también se puede utilizar el método *GetType* para recuperar el objeto *System.Type*. Este es un método heredado de *System.Object* que nos permite trabajar con objetos en vez de trabajar con clases. Los dos fragmentos de código que se muestran a continuación se podrían utilizar de manera intercambiable para recuperar un objeto *System.Type*.

```

//Recupera el objeto System.Type a partir de la definición de clase.
Type t1 = typeof(Apple);

```

```

//Recupera el objeto System.Type a partir del objeto.
Apple apple = new Apple();
Type t2 = apple.GetType();

```

sizeof

El operador *sizeof* se utiliza para obtener el tamaño en bytes de un tipo dado. Debería tener en cuenta dos factores en extremo importantes cuando utilice este operador. En primer lugar, sólo se podrá utilizar el operador *sizeof* con tipo valor. Por lo tanto, aunque este operador se puede utilizar en miembros de clase, no se puede utilizar sobre el tipo de clase en sí mismo. En segundo lugar, este operador se puede utilizar sólo en un método o bloque de código marcado como *inseguro*. Entraremos en el código *inseguro* en el Capítulo 17, «Cómo interoperar con código no gestionado». A continuación se muestra un ejemplo de la utilización del operador *sizeof* desde un método de clase marcado como *inseguro*:

```
using System;

class BasicTypes
{
    //NOTA: Se ha de declarar el código que utiliza el operador sizeof como
    //inseguro.
    static unsafe public void ShowSizes()
    {
        Console.WriteLine("\nTamaño de los tipos básicos");
        Console.WriteLine("tamaño de short = {0}", sizeof(short));
        Console.WriteLine("tamaño de int = {0}", sizeof(int));
        Console.WriteLine("tamaño de long = {0}", sizeof(long));
        Console.WriteLine("tamaño de bool = {0}", sizeof(bool));
    }
}

class Unsafe1App
{
    unsafe public static void Main()
    {
        BasicTypes.ShowSizes();
    }
}
```

El resultado de ejecutar esta operación es el siguiente:

```
Tamaño de los tipos básicos
tamaño de short = 2
tamaño de int = 4
tamaño de long = 8
tamaño de bool = 1
```

Aparte de los tipos integrados, el operador *sizeof* se puede utilizar para determinar los tamaños de otros tipos valor creados por el usuario, como por ejemplo los structs. Sin embargo, el resultado del operador *sizeof* puede a veces ser poco obvio, como en el siguiente ejemplo:

```
//Utilizando el operador sizeof.
using System;
```

```

struct StructWithNoMembers
{
}

struct StructWithMembers
{
    short s;
    int i;
    long l;
    bool b;
}

struct CompositeStruct
{
    StructWithNoMembers a;
    StructWithMembers b;
    StructWithNoMembers c;
}

class Unsafe2App
{
    unsafe public static void Main()
    {
        Console.WriteLine("\ntamaño de la estructura StructWithNoMembers = {0}",
            sizeof(StructWithNoMembers));
        Console.WriteLine("\n tamaño de la estructura StructWithMembers = {0}",
            sizeof(StructWithMembers));
        Console.WriteLine("\n tamaño de la estructura CompositeStruct = {0}",
            sizeof(CompositeStruct));
    }
}

```

Cuando esperamos que esta aplicación imprima un valor 0 para la estructura sin miembros (*StructWithNoMembers*), el valor 15 para la estructura con cuatro de los miembros básicos (*StructWithMembers*) y el valor 15 para la estructura que agrega las dos anteriores (*CompositeStruct*), el resultado real es el siguiente:

```

tamaño de la estructura StructWithNoMembers = 1
tamaño de la estructura StructWithMembers = 16
tamaño de la estructura CompositeStruct = 24

```

La razón de esto se debe al relleno y a la alineación de estructura, que se relaciona con cómo el compilador organiza un *struct* en el archivo imagen de salida. Por ejemplo, si un *struct* fuera de 3 bytes de largo y la alineación de bytes se fijara en 4 bytes, el compilador lo rellenaría automáticamente con un byte extra y el operador *sizeof* informaría de que el *struct* tiene 4 bytes de largo. Por lo tanto, debe tener esto en cuenta cuando determine el tamaño de una estructura en C#.

checked y unchecked

Estos dos operadores controlan la comprobación de desbordamiento para las operaciones matemáticas. Como estos operadores tienen que ver con la gestión de errores, se explicarán en el Capítulo 12, «Manejo de errores con excepciones».

Operadores matemáticos

El lenguaje C# admite los mismos operadores matemáticos básicos que admiten casi todos los lenguajes de programación: multiplicación (*), división (/), suma (+), resta (-) y módulo (%). El significado de los cuatro primeros operadores es obvio: el operador módulo devuelve el resto de la división entera. El siguiente código ilustra la utilización de estos operadores matemáticos de forma práctica:

```
using System;

class MathOpsApp
{
    public static void Main()
    {
        //La clase System.Random forma parte la biblioteca de clases
        //del Framework .NET. Su constructor por defecto pone el valor
        //semilla para el método Next con la fecha y hora actual
        Random rand = new Random();
        int a, b, c;

        a = rand.Next() % 100; //Limitamos el máximo a 99.
        b = rand.Next() % 100; //Limitamos el máximo a 99.

        Console.WriteLine("a={0} b={1}", a, b);

        c = a * b;
        Console.WriteLine("a * b = {0}", c);

        //Debemos indicar que el siguiente código utiliza enteros.
        //Por lo tanto, si a es menor que b, el resultado será siempre 0.
        //Para conseguir un resultado más preciso, deberíamos utilizar
        //variables de tipo doble o float

        c = a / b;
        Console.WriteLine("a / b = {0}", c);

        c = a + b;
        Console.WriteLine("a + b = {0}", c);

        c = a - b;
        Console.WriteLine("a - b = {0}", c);

        c = a % b;
        Console.WriteLine("a % b = {0}", c);
    }
}
```

Operadores unitarios

Hay dos tipos unitarios: más y menos. El operador unitario menos indica al compilador que el operando ha de ser negativo. Por lo tanto, el siguiente ejemplo tendrá como resultado en *a* el valor *-42*:

```
using System;

class Unary1App
{
    public static void Main()
    {
        int a = 0;

        a = -42;
        Console.WriteLine("{0}", a);
    }
}
```

Sin embargo, la ambigüedad aumenta cuando haces algo como lo siguiente:

```
using System;

class Unary2App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * -c;
        Console.WriteLine("{0}", a);
    }
}
```

La expresión $a = b * -c$ puede ser un poco confusa. Una vez más, los paréntesis ayudan a expresar esta línea de código de manera mucho más clara:

```
//Con paréntesis, es obvio que estamos multiplicando b por el negativo de c
a = b * (-c);
```

Si el menos unitario devuelve el valor negativo de un operando, podríamos pensar que el más unitario devolvería el valor positivo de un operando. Sin embargo, el más unitario no hace más que devolver el operando en su forma original, sin tener por lo tanto efecto alguno en éste. Por ejemplo, el siguiente código produce una salida de *-84*:

```
using System;

class Unary3App
{
    public static void Main()
```

```

    {
        int a;
        int b = 2;
        int c = -42;

        a = b * (+c);
        Console.WriteLine("{0}", a);
    }
}

```

Para recuperar el valor positivo de un operando, se puede utilizar la función *Math.Abs*. El siguiente código genera un resultado de 84:

```

using System;

class Unary4App
{
    public static void Main()
    {
        int a;
        int b = 2;
        int c = -42;

        a = b * Math.Abs(c);
        Console.WriteLine("{0}", a);
    }
}

```

Un último operador unitario que mencionaré brevemente es el operador (T), que es una forma del operador paréntesis que permite convertir un tipo a otro. Dado que este operador se puede sobrecargar utilizando una conversión definida por el usuario, se cubrirá en el Capítulo 13, «Sobrecarga de operadores y conversiones definidas por el usuario».

Operadores compuestos de asignación

Un *operador compuesto de asignación* es una combinación de un operador binario y el operador de asignación (=). Su sintaxis es

x op= y

donde *op* representa el operador. Fíjese que en vez de reemplazar el *valor-izq* con el *valor-der*, el operador tiene el mismo efecto que si escribiéramos

x = x op y

en el que el *valor-izq* sirve como base para el resultado de la operación.

Fijémonos en que decimos «tiene el mismo efecto». El compilador no traduce literalmente la expresión *x+=5* en *x=x+5*. Sólo funciona de esta manera desde un punto de

vista lógico. En realidad, debemos llamar la atención al hecho de utilizar una operación cuando el *valor-izq* es un método. Veamos este ejemplo:

```
using System;

class CompoundAssignment1App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment1App()
    {
        elements = new int[1];
        elements[0] = 42;
    }

    public static void Main()
    {
        CompoundAssignment1App app = new CompoundAssignment1App();

        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] = app.GetArrayElement()[0] + 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}
```

Observe en la línea en negrita —la invocación al método *CompoundAssignment1App.GetArrayElements* y las modificaciones siguientes de su primer elemento— para el que utilizamos la sintaxis de asignación de

$x = x \text{ op } y$

A continuación mostramos el código MSIL generado:

//Técnica ineficiente para $x = x \text{ op } y$.

```
.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size          79 (0x4f)
    .maxstack 4
    .locals (class CompoundAssignment1App V_0)
    IL_0000: newobj        instance void CompoundAssignment1App::.ctor()
    IL_0005: stloc.0
    IL_0006: ldstr          "{0}"
    IL_000b: ldloc.0
    IL_000c: call instance int32 [] CompoundAssignment1App::GetArrayElement()
    IL_0011: ldc.i4.0
    IL_0012: ldelema        ['mscorlib']System.Int32
    IL_0017: box           ['mscorlib']System.Int32
    IL_001c: call          void ['mscorlib']System.Console::WriteLine
```

```

                                (class System.String, class System.Object)
IL_0021: ldloc.0
IL_0022: call
instance int32[] CompoundAssignment1App::GetArrayElement()
IL_0027: ldc.i4.0
IL_0028: ldloc.0
IL_0029: call
instance int32[] CompoundAssignment1App::GetArrayElement()
IL_002e: ldc.i4.0
IL_002f: ldelem.i4
IL_0030: ldc.i4.5
IL_0031: add
IL_0032: stelem.i4
IL_0033: ldstr      "{0}"
IL_0038: ldloc.0
IL_0039: call
instance int32[] CompoundAssignment1App::GetArrayElement()
IL_003e: ldc.i4.0
IL_003f: ldelema    ['mscorlib']System.Int32
IL_0044: box        ['mscorlib']System.Int32
IL_0049: call void  ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
IL_004e :ret
} //end of method.'CompoundAssignment1App::Main'

```

Si nos fijamos en las líneas en **negrita** en el código MSIL, podemos ver que el método *CompoundAssignment1App.GetArrayElements* está siendo invocado en realidad ¡dos veces! En un escenario de ‘mejor caso’, esto es ineficiente. En el caso peor, podría ser desastroso, dependiendo de qué otras cosas haga el método.

Eche un vistazo ahora al siguiente código y fíjese en el cambio de sintaxis en la asignación por la del operador compuesto de asignación:

```

using System;

class CompoundAssignment2App
{
    protected int[] elements;
    public int[] GetArrayElement()
    {
        return elements;
    }

    CompoundAssignment2App()
    {
        elements = new int [1];
        elements[0] = 42;
    }

    public static void Main()
    {
        CompoundAssignment2App app = new CompoundAssignment2App();
    }
}

```

```

        Console.WriteLine("{0}", app.GetArrayElement()[0]);
        app.GetArrayElement()[0] += 5;
        Console.WriteLine("{0}", app.GetArrayElement()[0]);
    }
}

```

El uso del operador compuesto de asignación produce un código MSIL mucho más eficiente:

//Técnica más eficiente para $x \text{ op } = y$.

```

.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size          76 (0x4c)
    .maxstack 4
    .locals (class CompoundAssignment2App V_0,int32[] V_1)
    IL_0000: newobj          instance void CompoundAssignment2App::.ctor()
    IL_0005: stloc.0
    IL_0006: ldstr           "{0}"
    IL_000b: ldloc.0
    IL_000c: call instance int32[] CompoundAssignment2App::GetArrayElement()
    IL_0011: ldc.i4.0
    IL_0012: ldelema         ['mscorlib']System.Int32
    IL_0017: box            ['mscorlib']System.Int32
    IL_001c: call void      ['mscorlib']System.Console::WriteLine
                                   (class System.String, class System.Object)

    IL_0021: ldloc.0
    IL_0022: call instance int32[] CompoundAssignment2App::GetArrayElement()

    IL_0027: dup
    IL_0028: stloc.1
    IL_0029: ldc.i4.0
    IL_002a: ldloc.1

    IL_002b: ldc.i4.0
    IL_002c: ldelem.i4
    IL_002d: ldc.i4.5
    IL_002e: add
    IL_002f: stelem.i4
    IL_0030: ldstr           "{0}"
    IL_0035: ldloc.0
    IL_0036: call instance int32[] CompoundAssignment2App::GetArrayElement()
    IL_003b: ldc.i4.0
    IL_003c: ldelema         ['mscorlib']System.Int32
    IL_0041: box            ['mscorlib']System.Int32
    IL_0046: call void      ['mscorlib']System.Console::WriteLine
                                   (class System.String, class System.Object)

    IL_004b: ret
} //end of method 'CompoundAssignment2App::Main'

```

Podemos ver que se está utilizando el código de operación MSIL *dup*. El código de operación *dup* duplica el elemento de la cima de la pila, realizando de este modo una copia del valor recuperado del método *CompoundAssignment2App.GetArrayElements*.

El objetivo de este ejercicio ha sido ilustrar que aunque conceptualmente $x+=y$ es equivalente a $x = x+y$, se pueden encontrar sutiles diferencias en el MSIL generado. Estas diferencias significan que tiene que pensar cuidadosamente qué sintaxis va a utilizar en cada circunstancia. Como regla general, y nuestra recomendación, es utilizar los operadores compuestos de asignación cuando y donde sea posible.

Operadores de incremento y operadores de decremento

Como vestigios de los mismos mecanismos abreviados que se introdujeron por primera vez en el lenguaje C y que se han llevado tanto a C++ como a Java, los operadores de incremento y decremento permiten indicar de manera breve que queremos incrementar o decrementar una variable que representa un valor numérico en 1. Por lo tanto, $i++$ es el equivalente a sumar 1 al valor actual de i .

Existen dos versiones de operadores de incremento y de decremento y a menudo constituyen una fuente de confusión. Conocidas normalmente como *prefija* y *postfija*, el tipo de operador indica en qué momento el efecto lateral modifica la variable. Con las versiones prefijas de los operadores de incremento y decremento —esto es $++a$ y $--a$, respectivamente— la operación se ejecuta y a continuación se obtiene el valor. Con las versiones postfijas de los operadores de incremento y decremento — $a++$ y $a--$, respectivamente— se generan los valores y luego se ejecuta la operación. Echemos un vistazo al siguiente ejemplo:

```
using System;

class IncDecApp
{
    public static void Foo(int j)
    {
        Console.WriteLine("IncDecApp.Foo j = {0}", j);
    }

    public static void Main()
    {
        int i = 1;

        Console.WriteLine("Antes de invocar Foo(i++) = {0}", i);
        Foo(i++);
        Console.WriteLine("Después de invocar Foo(i++) = {0}", i);

        Console.WriteLine("\n");

        Console.WriteLine("Antes de invocar Foo(++i) = {0}", i);
        Foo(++i);
        Console.WriteLine("Después de invocar Foo(++i) = {0}", i);
    }
}
```

Esta aplicación genera los siguientes resultados:

```

Antes de invocar Foo(i++) = 1
IncDecApp.Foo j = 1
Después de invocar Foo(i++) = 2

```

```

Antes de invocar Foo(++i) = 2
IncDecApp.Foo j = 3
Después de invocar Foo(++i) = 3

```

La diferencia en el ejemplo es *cuándo* se genera el valor y *cuándo* se modifica el operando. En la llamada a *Foo(++i)*, el valor de *i* se pasa (sin modificar) a *Foo*, y *después* de que *Foo* termine, *i* se incrementa. Esto puede observarlo en los siguientes extractos de MSIL. Fíjese en que el código de operación MSIL *add* no se invoca hasta después de que el valor se haya colocado en la pila.

```

IL_0013:    ldloc.0
IL_0014:    dup
IL_0015:    ldc.i4.1
IL_0016:    add
IL_0017:    stloc.0
IL_0018:    call        void IncDecApp::Foo(int32)

```

Ahora veamos el operador de preincremento que hemos utilizado en la llamada a *Foo(++a)*. En este caso, el MSIL generado es parecido al código que se muestra a continuación. Fíjese en que el código de operación MSIL *add* se invoca *antes* de que el valor se coloque en la pila para la siguiente llamada al método *Foo*.

```

IL_0049:    ldloc.0
IL_004a:    ldc.i4.1
IL_004b:    add
IL_004c:    dup
IL_004d:    stloc.0
IL_004e:    call        void IncDecApp::Foo(int32)

```

Operadores relacionales

La mayoría de operadores devuelven un resultado numérico. Sin embargo, los operadores relacionales son un poco diferentes, ya que generan un resultado booleano. En vez de realizar una operación matemática sobre un conjunto de operandos, un operador relacional compara la relación entre los operandos y devuelve un valor *true* si la relación es cierta y *false* si la relación no lo es.

Operadores de comparación

El conjunto de operadores relacionales conocido como operadores de comparación está formado por menor que (<), menor o igual que (<=), mayor que (>), mayor o igual que (>=), igual a (==) y distinto de (!=). El significado de estos operadores es obvio cuando trabajamos con números, pero no es tan obvio cómo funciona cada operador sobre objetos. Veamos el siguiente ejemplo:

```

using System;

class NumericTest
{
    public NumericTest(int i)
    {
        this.i = i;
    }

    protected int i;
}

class RelationalOps1App
{
    public static void Main()
    {
        NumericTest test1 = new NumericTest(42);
        NumericTest test2 = new NumericTest(42);

        Console.WriteLine("{0}", test1 == test2);
    }
}

```

Si usted es programador de Java, sabrá lo que va a suceder aquí. Sin embargo, la mayoría de programadores de C++ se sorprenderán probablemente al ver que este ejemplo imprime una cadena *false*. Recordemos que al instanciar un objeto, consigue una referencia a un objeto cuya memoria se ha reservado en el montón. Por lo tanto, cuando utiliza un operador relacional para comparar dos objetos, el compilador C# no compara el contenido de los objetos. En vez de hacer esto, compara sus respectivas direcciones. Una vez más, para entender qué sucede aquí, veremos el MSIL para el programa:

```

.method public hidebysig static void Main() il managed
{
    .entrypoint
    //Code size          39 (0x27)
    .maxstack 3
    .locals (class NumericTest V_0,
            class NumericTest V_1,
            bool V_2)
    IL_0000: ldc.i4.s    42
    IL_0002: newobj      instance void NumericTest::.ctor(int32)
    IL_0007: stloc.0
    IL_0008: ldc.i4.s    42
    IL_000a: newobj      instance void NumericTest::.ctor(int32)
    IL_000f: stloc.1
    IL_0010: ldstr       "{0}"
    IL_0015: ldloc.0
    IL_0016: ldloc.1
    IL_0017: ceq
    IL_0019: stloc.2
    IL_001a: ldloc.s     V_2
    IL_001c: box         ['mscorlib']System.Boolean
    IL_0021: call void    ['mscorlib']System.Console::WriteLine
                                (class System.String, class System.Object)
}

```

```
IL_0026: ret
} //end of method 'RelationalOps1App::Main'
```

Echemos un vistazo a la línea de *.locals*. El compilador está declarando que su método *Main* tendrá tres variables locales. Las dos primeras son objetos *NumericTest* y la tercera es un tipo booleano. Ahora saltaremos a las líneas *IL_0002* y *IL_0007*. En éstas, el MSIL instancia el objeto *test1*, y con el código de operación *stloc*, almacena la referencia devuelta en la primera variable local. Sin embargo, el punto clave aquí es que el MSIL está almacenando la dirección del nuevo objeto que hemos creado. A continuación, en las líneas *IL_000a* e *IL_000f*, podemos ver los códigos de operación MSIL que crean el objeto *test2* y almacenan la referencia devuelta en una segunda variable local. Finalmente, las líneas *IL_0015* e *IL_0016* invocan el código de operación *ceq*, que compara los dos valores de la cima de la pila (esto es, las referencias a los objetos *test1* y *test2*). El valor devuelto se almacena después en la tercera variable local y posteriormente se imprime a través de la invocación a *System.Console.WriteLine*.

¿Cómo podemos realizar una comparación miembro a miembro de dos objetos? La respuesta es la clase base de todos los objetos del Framework .NET. La clase *System.Object* tiene un método llamado *Equals* diseñado para este propósito. Por ejemplo, el siguiente código ejecuta una comparación del contenido del objeto como podríamos esperar y devuelve un valor *true*:

```
using System;

class RelationalOps2App
{
    public static void Main()
    {
        Decimal test1 = new Decimal(42);
        Decimal test2 = new Decimal(42);

        Console.WriteLine("{0}", test1.Equals(test2));
    }
}
```

Fíjese que el ejemplo *RelationalOps1App* utilizaba una clase propia (*NumericTest*) y que este segundo ejemplo ha utilizado una clase .NET (*Decimal*). La razón de esto es que el método *Equals* debe ser redefinido para que realice la comparación miembro a miembro. Por lo tanto, si hubiésemos utilizado el método *Equals* en la clase *NumericTest*, no habría funcionado, al no haber redefinido el método. Sin embargo, al tener la clase *Decimal* el método *Equals* redefinido, funciona como esperamos.

Otra forma de manejar una comparación de objetos es mediante el mecanismo de *sobrecarga de operadores*. El sobrecargar un operador define la operación que tiene lugar entre objetos de un tipo específico. Por ejemplo, con objetos *string*, el operador *+* concatena las cadenas en lugar de realizar una operación de suma. Entraremos en la sobrecarga de operadores en el Capítulo 13.

Operadores de asignación

El valor en el lado izquierdo de un operador de asignación se conoce como *valor-izq*; el valor del lado derecho se llama *valor-der*. El *valor-der* puede ser cualquier constante, variable, número o expresión que se pueda resolver como un valor compatible con el *valor-izq*. Sin embargo, el *valor-izq* debe ser una variable de un tipo definido. La razón de esto es que un valor se ha de copiar desde la derecha a la izquierda. Por lo tanto, debe existir espacio reservado en memoria, que es el destino último del nuevo valor. Como ejemplo, podemos declarar *i=4*, porque *i* representa una localización física en memoria, o en la pila o en el montón, dependiendo el tipo real de la variable *i*. Sin embargo, no se puede ejecutar la instrucción *4=i*, porque 4 es un valor, no una variable de memoria de la cual se puede cambiar el contenido. Haciendo un aparte, técnicamente, la regla en C# es que el *valor-izq* puede ser una variable, propiedad o indizador. Para más datos sobre propiedades e indizadores, vuelva al Capítulo 7. Para dejar las cosas sencillas, nos atendremos a los ejemplos que utilizan variables encontrados en este capítulo.

Aunque la asignación numérica es bastante directa, las operaciones de asignación que implican objetos es una propuesta mucho más artificiosa. Recuerde que cuando trabaja con objetos no está tratando simplemente con elementos con memoria reservada en la pila que son fácilmente copiados y movidos de un sitio a otro. Cuando manipula objetos, en realidad sólo tiene una referencia a una entidad cuya memoria está reservada en el montón. Por lo tanto, cuando intenta asignar un objeto (o cualquier tipo referencia) a una variable, no copia datos del mismo modo que copia tipos valor. Simplemente está copian-do una referencia de un lugar a otro.

Digamos que tenemos dos objetos: *test1* y *test2*. Si declara *test1=test2*, *test1* no es una copia de *test2*. ¡Es la misma cosa! El objeto *test1* apunta a la misma posición de memoria que *test2*. Por lo tanto, cualquier cambio en el objeto *test1* afecta también al objeto *test2*. A continuación tenemos un programa que lo ilustra:

```
using System;
class Foo
{
    public int i;
}

class RefTest1App
{
    public static void Main()
    {
        Foo test1 = new Foo();
        test1.i = 1;

        Foo test2 = new Foo();
        test2.i = 2;

        Console.WriteLine("ANTES DE LA ASIGNACIÓN DE OBJETOS");
        Console.WriteLine("test1.i={0}", test1.i);
        Console.WriteLine("test2.i={0}", test2.i);
        Console.WriteLine("\n");
    }
}
```

```

test1 = test2;

Console.WriteLine("DESPUÉS DE LA ASIGNACIÓN DE OBJETOS");
Console.WriteLine("test1.i={0}", test1.i);
Console.WriteLine("test2.i={0}", test2.i);
Console.WriteLine("\n");

test1.i = 42;

Console.WriteLine("DESPUÉS DE CAMBIAR SÓLO EL MIEMBRO DE TEST1");
Console.WriteLine("test1.i={0}", test1.i);
Console.WriteLine("test2.i={0}", test2.i);
Console.WriteLine("\n");
}
}

```

Ejecute este código y obtendrá la siguiente salida:

```

ANTES DE LA ASIGNACIÓN DE OBJETOS
test1.i = 1
test2.i = 2

DESPUÉS DE LA ASIGNACIÓN DE OBJETOS
test1.i = 2
test2.i = 2

DESPUÉS DE CAMBIAR SÓLO EL MIEMBRO DE TEST1
test1.i = 42
test2.i = 42

```

Revisemos este ejemplo para ver qué ha sucedido en cada momento en el programa. *Foo* es una clase sencilla que define un miembro simple llamado *i*. Se crean dos instancias de esta clase —*test1* y *test2*— en el método *Main*, y en cada caso se fija el miembro *i* del nuevo objeto (a un valor de 1 y 2, respectivamente). En este momento, imprimimos los valores y vemos que se muestran los valores esperados, con *test1.i* con un valor 1 y *test2.i* con un valor 2. Aquí empieza la diversión. La siguiente línea asigna el objeto *test2* a *test1*. Los programadores Java que estén presentes saben lo que viene a continuación. Sin embargo, la mayoría de programadores C++ esperarían que el miembro *i* del objeto *test1* fuera igual al miembro del objeto *test2* (asumiendo esto porque la aplicación compilada debe tener algún tipo de operador copia inteligente implícito que esté funcionando). En realidad, esa es la sensación dada cuando imprimimos el valor de los miembros de los objetos. Sin embargo, la nueva relación entre estos objetos es mucho más profunda que eso. El programa asigna 42 a *test1.i* y de nuevo imprime los valores del miembro *i* de ambos objetos. ¿Qué? ¡Al cambiar el objeto *test1*, ha cambiado el objeto *test2* también! Esto se debe a que el objeto que anteriormente conocíamos como *test1* ya no existe. Con la asignación de *test1* a *test2*, el objeto *test1* se pierde, ya que no se le hace referencia en la aplicación, y finalmente es recogido por el recolector de basura (GC). Tanto el objeto *test1* como el objeto *test2* apuntan a la misma dirección de memoria del montón. Por lo tanto, un cambio hecho a cualquier variable lo verá el usuario de la otra variable.

Observe que en las últimas dos líneas de la salida que aunque el código fija el valor de *test1.i* sólo, el valor de *test2.i* también se ha visto afectado. Una vez más, esto se debe a que ambas variables apuntan a la misma dirección de memoria, el comportamiento que esperaría si fuera programador Java. Sin embargo, es un contraste extremo con lo que esperaría un programador C++, porque en C++ el hecho de copiar objetos significa exactamente eso —cada variable tiene su única copia de miembros de forma que las modificaciones en un objeto no impactan en el otro. Dado que esto es clave para entender cómo trabajar con objetos en C#, desviémonos un poco y veamos qué sucede cuando pasamos un objeto a un método:

```
using System;

class Foo
{
    public int i;
}

class RefTest2App
{
    public void ChangeValue(Foo f)
    {
        f.i = 42;
    }

    public static void Main()
    {
        RefTest2App app = new RefTest2App();

        Foo test = new Foo();
        test.i = 6;

        Console.WriteLine("ANTES DE INVOCAR EL MÉTODO");
        Console.WriteLine("test.i={0}", test.i);
        Console.WriteLine("\n");

        app.ChangeValue(test);

        Console.WriteLine("DESPUÉS DE INVOCAR EL MÉTODO");
        Console.WriteLine("test.i={0}", test.i);
        Console.WriteLine("\n");
    }
}
```

En la mayoría de lenguajes —excluyendo a Java—, este código resultaría en una copia del objeto *test* que se crearía en la pila local del método *RefTest2App.ChangeValue*. Si fuera esto el caso, el objeto *test* creado en el método *Main* nunca vería ningún cambio hecho al objeto *f* dentro del método *ChangeValue*. Sin embargo, una vez más, lo que está sucediendo aquí es que el método *Main* ha pasado una referencia a su objeto *test*, cuya memoria está reservada en el montón. Cuando el método *ChangeValue* manipula su variable local *f.i*, también está manipulando directamente el objeto *test* del método *Main*.

RESUMEN

Una aspecto clave de cualquier lenguaje de programación es la forma de manejar las operaciones de asignación, matemáticas, relacionales y lógicas para llevar a cabo el trabajo básico que requiere cualquier aplicación del mundo real. Estas operaciones se controlan en el código mediante operadores. Los factores que determinan los efectos de los operadores en el código incluyen la precedencia y la asociatividad por la derecha y por la izquierda. Además de proporcionar un conjunto de eficaces operadores predefinido, C# extiende estos operadores mediante las implementaciones de operadores definidos por el usuario, que trataremos en el Capítulo 13.

Control del flujo de programa

Las instrucciones que nos permiten controlar el flujo en una aplicación C# caen en una de estas tres categorías principales: instrucciones condicionales, instrucciones de iteración e instrucciones de salto. En todos estos casos se ejecuta una operación que produce un valor booleano que se utiliza para controlar el flujo de ejecución. En este capítulo aprenderá a utilizar cada uno de estos tipos de instrucciones para controlar el flujo de la estructura.

INSTRUCCIONES DE SELECCIÓN

Utilizará las instrucciones de selección para determinar qué código y cuándo se debería ejecutar. C# tiene dos instrucciones de selección: la instrucción *switch*, que se utiliza para ejecutar código basándose en un valor, y la instrucción *if*, que ejecuta el código basándose en una condición booleana. La más utilizada de estas instrucciones de selección es la instrucción *if*.

La instrucción *if*

La instrucción *if* ejecuta una o más instrucciones si la expresión que se está evaluando tiene como resultado *true*. A continuación mostramos la sintaxis de la instrucción *if* —los corchetes indican el uso opcional de la instrucción *else* (que describiremos en breve):

```
if (expresión)
    instrucción1
[else
    instrucción2]
```

Aquí, *expresión* es cualquier expresión que genera un resultado booleano. Si *expresión* genera un resultado *true*, el control se pasa a *instrucción1*. Si el resultado es *false* y existe un bloque *else*, el control se pasa a *instrucción2*. Debemos indicar que *instruc-*

ción1 e *instrucción2* pueden consistir en una instrucción única terminada en punto y coma (conocida como instrucción simple) o múltiples instrucciones entre llaves (una instrucción compuesta). El ejemplo siguiente muestra una instrucción compuesta que se ejecuta si *expresión1* produce un resultado *true*:

```
if (expresión1)
{
    instrucción1
    instrucción2
}
```

En el siguiente ejemplo, la aplicación pide que el usuario introduzca un número entre 1 y 10. A continuación se genera un número aleatorio y se le pregunta al usuario si el número que ha elegido coincide con el número aleatorio. Este sencillo ejemplo ilustra cómo utilizar la instrucción *if* en C#.

```
using System;

class IfTest1App
{
    const int MAX = 10;

    public static void Main()
    {
        Console.Write("Adivine un número entre 1 y {0}...", MAX);
        string inputString = Console.ReadLine();

        int userGuess = Convert.ToInt32(inputString);

        Random rnd = new Random();
        double correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write("El número correcto era {0} y usted introdujo {1}...",
                      correctNumber, userGuess);
        if (userGuess == correctNumber)    //¡Acertamos!
        {
            Console.WriteLine(";Felicitades!");
        }
        else    //¡Respuesta incorrecta!
        {
            Console.WriteLine(";Quizás la próxima vez!");
        }
    }
}
```

Cláusulas *else* múltiples

La cláusula *else* de la instrucción *if* nos permite especificar un trayecto de acción alternativo que se seguirá si la instrucción *if* tiene como resultado *false*. En el ejemplo de

adivinar un número, la aplicación realizaba una comparación sencilla entre el número que el usuario eligió y el número que se generó aleatoriamente. En este caso sólo existían dos posibilidades: el usuario había acertado o no. También puede combinar *if* y *else* para manejar situaciones en las que quiera comprobar más de dos condiciones. En el ejemplo siguiente se pregunta al usuario qué lenguaje está utilizando actualmente (excluyendo C#). Hemos incluido la posibilidad de elegir entre tres lenguajes, por lo que la instrucción *if* debe ser capaz de tratar con al menos cuatro posibles respuestas: los tres lenguajes y el caso en que el usuario elija un lenguaje desconocido. Veamos una forma de programar con una instrucción *if/else*:

```
using System;

class IfTest2App
{
    const string CPlusPlus = "C++";
    const string VisualBasic = "Visual Basic";
    const string Java = "Java";

    public static void Main()
    {
        Console.Write("¿Cuál es su lenguaje preferido" +
                      "(excluyendo C#)?");
        string inputString = Console.ReadLine();

        if (0 == String.Compare(inputString, CPlusPlus, true))
        {
            Console.WriteLine("\nNo tendrá problema si utiliza" +
                              "C#!");
        }
        else if (0 == String.Compare(inputString, VisualBasic, true))
        {
            Console.WriteLine("\nEncontrará muchas características
                              interesantes de VB en C#!");
        }
        else if (0 == String.Compare(inputString, Java, true))
        {
            Console.WriteLine("\n¡¡Le será fácil aprender" +
                              "C#!!");
        }
        else
        {
            Console.WriteLine("\nLo sentimos, no computable");
        }
    }
}
```

Debe reparar en la utilización del operador `==` para comparar 0 con el valor devuelto por *String.Compare*. Esto se debe a que *String.Compare* devolverá -1 si la primera cadena es menor que la segunda cadena; 1 si la primera cadena es mayor que la segunda, y 0 si las dos son idénticas. Sin embargo, esto ilustra algunos interesantes detalles que se describen en la siguiente sección sobre cómo C# fuerza la utilización de la instrucción *if*.

Cómo fuerza C# las reglas *if*

Un aspecto de la instrucción *if* que sorprende a los nuevos programadores C# es el hecho de que la expresión evaluada debe devolver un valor booleano. Esto contrasta con lenguajes como C++, donde se permite el uso de instrucción *if* para comprobar cualquier variable que tenga un valor distinto de 0. El siguiente ejemplo ilustra varios errores comunes que cometen los programadores C++ cuando intentan utilizar la instrucción *if* en C# por primera vez:

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class InvalidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo) //ERROR: intentamos convertir int en bool
        {
        }

        TestClass t = GetTestClass();
        if (t) //ERROR: intentamos convertir TestClass en bool.
        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i) //ERROR: intentamos convertir ITest en bool.

            {
                //métodos ITest
            }
        }
    }
}
```

Si intenta compilar este código, recibirá los siguientes errores del compilador C#:

```
invalidIf.cs(22,7): error CS0029: Cannot implicitly convert type
                    'int' to 'bool'
invalidIf.cs(27,7): error CS0029: Cannot implicitly convert type
                    'TestClass' to 'bool'
```

```
invalidIf.cs(31,14): warning CS0183:
The given expression is always of the provided ('ITest') type
invalidIf.cs(32,8): error CS0029: Cannot implicitly convert type
                        'ITest' to 'bool'
```

Como puede comprobar, el compilador se queja tres veces en respuesta a tres intentos de utilizar la instrucción *if* con un operador que no produce un valor booleano. El motivo de todo esto es que los diseñadores quieren ayudar a evitar código ambiguo y creen que el compilador debería imponer el propósito original de la instrucción *if*; esto es, controlar el flujo de control basándose en el resultado de una comprobación booleana. El ejemplo anterior se reescribe para que se compile sin errores. Cada línea que causaba un error del compilador en el programa anterior se ha modificado para contener una expresión que devuelva un resultado booleano, evitando así los avisos del compilador.

```
using System;

interface ITest
{
}

class TestClass : ITest
{
}

class ValidIfApp
{
    protected static TestClass GetTestClass()
    {
        return new TestClass();
    }

    public static void Main()
    {
        int foo = 1;
        if (foo > 0)
        {

        }

        TestClass t = GetTestClass();
        if (t != null)
        {
            Console.WriteLine("{0}", t);

            ITest i = t as ITest;
            if (i != null)
            {
                //métodos ITest.
            }
        }
    }
}
```

La instrucción *switch*

Utilizando la instrucción *switch*, puede especificar una expresión que devuelve un valor y uno o más fragmentos de código que se ejecutarán en función del resultado de la expresión. Es similar a la utilización de múltiples instrucciones *if/else*, pero aunque puede especificar múltiples instrucciones condicionales (posiblemente no relacionadas), una instrucción *switch* consiste en una única instrucción condicional seguida por todos los resultados que nuestro código está preparado para manejar. A continuación se muestra la sintaxis:

```
switch (expresión_switch)
{
    case expresión_constante:
        instrucción
        instrucción_de_salto
    :
    case expresión_constanteN:
        instrucciónN
    [por_defecto]
}
```

Hay dos reglas importantes que debemos tener en cuenta. Primera, la *expresión_switch* debe ser del tipo (o implícitamente convertible a) *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char* o *string* (o un *enum* basado en uno de estos tipos). Segunda, se debe proporcionar una *instrucción_de_salto* para cada instrucción *case*, excepto si la instrucción *case* es la última en el *switch*, que incluye la instrucción *break*. Como ésta funciona de forma distinta que en otros lenguajes, lo explicaré en más detalle más tarde en este capítulo en «No cierre de *case* en la instrucción *switch*».

Conceptualmente, la instrucción *switch* funciona de la misma forma que la instrucción *if*. En primer lugar, se evalúa la *expresión_switch* y el resultado se compara con cada *expresión_constante* o *etiqueta case* definida en las diferentes instrucciones *case*. Una vez que se encuentra una coincidencia, el control se pasa a la primera línea de código en esa instrucción *case*.

Además de permitirle definir diferentes instrucciones *case*, la instrucción *switch* permite la definición de una instrucción *por_defecto*. Ésta es similar a la *cláusula else* de una instrucción *if*. Tenga en cuenta que sólo puede haber una etiqueta por defecto para cada instrucción *switch*, y si no se encuentra una etiqueta *case* adecuada para la *expresión_switch*, el control se pasa a la primera línea de código que sigue a la llave de cierre de la instrucción *switch*. A continuación sigue un ejemplo —una clase *Payment* que utiliza una instrucción *switch* para determinar qué forma de pago se ha seleccionado:

```
using System;

enum Tenders : int
{
    Cash = 1,
```

```

        Visa,
        MasterCard,
        AmericanExpress
    };

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.Cash:
                Console.WriteLine("\nEfectivo -Aceptado");
                break;

            case (int)Tenders.Visa:
                Console.WriteLine("\nVisa -Aceptada");
                break;

            case (int)Tenders.MasterCard:
                Console.WriteLine("\nMastercard -Aceptada");
                break;

            case (int)Tenders.AmericanExpress:
                Console.WriteLine("\nAmerican Express -Aceptada");
                break;

            default:
                Console.WriteLine("\nLo sentimos -Medio de pago inválido");
                break;
        }
    }
}

class SwitchApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
    }
}

```

```

        payment.ProcessPayment();
    }
}

```

La ejecución de esta aplicación termina con la siguiente salida, ya que instanciamos la clase *Payment* pasándole un valor *Tenders.Visa*:

Visa -Aceptada.

Cómo combinar etiquetas *case*

En el ejemplo de *Payment*, utilizamos una etiqueta *case* para cada evaluación posible del campo *Payment.tender*. Sin embargo, ¿qué sucede si queremos combinar etiquetas *case*? Por ejemplo, podría querer mostrar una cuadro de diálogo para cualquiera de los tres tipos de tarjeta estimados como válidos en el *enumeradoTender*. En ese caso, podríamos colocar las etiquetas *case* una tras otra, como mostramos:

```

using System;

enum Tenders : int
{
    Cash = 1,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    public void ProcessPayment()
    {
        switch ((int) (this.tender))
        {
            case (int)Tenders.Cash:

```

```

        Console.WriteLine
            ("\nEfectivo -El medio de pago favorito de todos");
        break;

        case (int)Tenders.Visa:
        case (int)Tenders.MasterCard:
        case (int)Tenders.AmericanExpress:
            Console.WriteLine
                ("\nMostrar el cuadro de diálogo de Autorización de tarjeta de crédito");
            break;

        default:
            Console.WriteLine("\nLo sentimos -Medio de pago inválido");
            break;
    }
}

class CombiningCaseLabelsApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.MasterCard);
        payment.ProcessPayment();
    }
}

```

Si instancia la clase *Payment* con *Tenders.Visa*, *Tenders.MasterCard* o *Tenders.AmericanExpress*, conseguirá la siguiente salida:

Mostrar el cuadro de diálogo de Autorización de tarjeta de crédito.

No cierre de case en las instrucciones switch

A lo largo de la fase de diseño del C#, los diseñadores analizaban a fondo la relación entre «riesgo/beneficio» cuando decidían si se debería incluir o no una cierta característica en el lenguaje. La característica ‘no cierre de *case*’ es un ejemplo de característica que no pasó la prueba. Normalmente, en C#, una instrucción *case* se ejecuta cuando la *expresión_constante* coincide con la *expresión_switch*. La instrucción *switch* finaliza a continuación con una instrucción *break*. Si no cerramos un *case*, en ausencia de una instrucción *break*, provocamos que la siguiente instrucción *case* en la instrucción *switch* se ejecute.

Aunque no está permitido en C#, el no cierre de *case* se utiliza típicamente en situaciones en las que tenemos dos etiquetas *case* y la segunda etiqueta representa una operación que haremos en cualquier caso. Por ejemplo, una vez escribimos un editor de bases de datos en C++ que permitía a los usuarios finales crear sus tablas y campos gráficamente. Cada una de estas tablas se mostraba en forma de árbol en una interfaz de usuario similar al Explorador de Windows. Si el usuario final pulsaba el botón secundario del ratón en el árbol, queríamos que se mostrara un menú con opciones como Imprimir todas

las tablas y Crear nueva tabla. Si el usuario pulsaba el botón secundario sobre una tabla específica, queríamos que se mostrara un menú contextual para esa tabla. Sin embargo, ese menú también tenía que incluir todas las opciones de la vista de árbol. Conceptualmente, el código se parecía al siguiente:

```
//Menú creado dinámicamente en C++.
switch(itemSelected)
{
    case TABLE:
        //Añade las opciones de menú basadas en la tabla actual;
        //no se añade break de manera intencionada.

    case TREE_VIEW:
        //Añade las opciones de menú para la vista de árbol
        break;
}
//Mostrar menú.
```

La primera etiqueta *case* supone una combinación de las dos etiquetas sin que tengamos que duplicar código o insertar dos llamadas al mismo método. Sin embargo, los diseñadores del lenguaje C# decidieron que aunque esta característica puede ser cómoda, el beneficio no vale el riesgo que implica, ya que la mayoría de las veces una instrucción *break* no se especifica de manera intencionada, lo que produce defectos que son difíciles de rastrear. Para conseguir esto en C#, probablemente sería mejor utilizar una instrucción *if* en su lugar:

```
//Creación de un menú dinámicamente.
if (itemSelected == TABLE)
{
    //Añadir las opciones de menú basándonos en la tabla actual.
}
//Añadir opciones de menú para la vista de árbol
//Mostrar menú.
```

INSTRUCCIONES DE ITERACIÓN

En C#, las instrucciones *while*, *do/while*, *for* y *foreach* nos permiten ejecutar iteraciones o bucles. En cada caso, se ejecuta una instrucción simple o compuesta hasta que una expresión booleana produce un *true*, excepto en el caso de la instrucción *foreach*, que se utiliza para iterar a lo largo de una lista de objetos.

Instrucción *while*

La instrucción *while* tiene la siguiente estructura:

```
while (expresión_booleana)
    instrucción_interna
```

Si utilizamos el ejemplo de adivinación de números que vimos anteriormente en el capítulo, podríamos reescribirlo, como veremos a continuación, con una instrucción *while*, de forma que podríamos seguir jugando hasta que o bien adivináramos el número correcto o bien decidiéramos abandonar.

```
using System;

class WhileApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

    public static void Main()
    {
        Random rnd = new Random();
        double correctNumber;

        string inputString;
        int userGuess;

        bool correctGuess = false;
        bool userQuit = false;

        while (!correctGuess && !userQuit)
        {
            correctNumber = rnd.NextDouble() * MAX;
            correctNumber = Math.Round(correctNumber);

            Console.Write
                ("Adivine un número entre {0} y {1}...({2} para salir)",
                 MIN, MAX, QUIT_CHAR);
            inputString = Console.ReadLine();

            if (0 == string.Compare(inputString, QUIT_CHAR, true))
                userQuit = true;
            else
            {
                userGuess = Convert.ToInt32(inputString);
                correctGuess = (userGuess == correctNumber);

                Console.WriteLine
                    ("El número correcto era {0}\n",
                     correctNumber);
            }
        }

        if (correctGuess && !userQuit)
        {
            Console.WriteLine(";Felicitades!");
        }
        else
        {
            Console.WriteLine(";Quizás la próxima vez!");
        }
    }
}
```

```

    }
}

```

Si codificamos y ejecutamos esta aplicación, obtendremos una salida similar a la siguiente:

```

C:\>WhileApp
Adivine un número entre 1 y 10...(Q para salir)3
El número correcto era 5

Adivine un número entre 1 y 10...(Q para salir)5
El número correcto era 5
¡Felicidades!

C:\>WhileApp
Adivine un número entre 1 y 10...(Q para salir)q
¡Quizás la próxima vez!

```

La instrucción *do/while*

Si volvemos a observar la sintaxis de la instrucción *while*, veremos la posibilidad de que se produzca un problema. La *expresión_booleana* se evalúa antes de que se ejecute la *instrucción_interna*. Por este motivo, la aplicación en la sección anterior inicializaba las variables *correctGuess* y *userQuit* a *false* para garantizar que se entrase en el bucle *while*. Una vez dentro, estos valores se controlan a través del acierto en adivinar el número o del abandono por parte del usuario. Sin embargo, ¿qué sucede si queremos asegurarnos de que la *instrucción_interna* se ejecuta siempre al menos una vez sin tener que asignar valores de forma artificial a las variables? Para esto es para lo que utilizamos la instrucción *do/while*.

La instrucción *do/while* tiene la siguiente estructura:

```

do
    instrucción_interna
while (expresión_booleana)

```

Dado que la evaluación de la *expresión_booleana* de la instrucción *while* sucede después de la *instrucción_interna*, se nos garantiza que ésta se ejecutará al menos una vez. La aplicación de adivinación de números reescrita para utilizar la instrucción *do/while* se muestra a continuación:

```

using System;

class DoWhileApp
{
    const int MIN = 1;
    const int MAX = 10;
    const string QUIT_CHAR = "Q";

```

```

public static void Main()
{
    Random rnd = new Random();
    double correctNumber;

    string inputString;
    int userGuess = -1;

    bool userHasNotQuit = true;

    do
    {
        correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write
            ("Adivine un número entre {0} y {1}...({2} para salir)",
             MIN, MAX, QUIT_CHAR);
        inputString = Console.ReadLine();

        if (0 == string.Compare(inputString, QUIT_CHAR, true))
            userHasNotQuit = false;
        else
        {
            userGuess = inputString.ToInt32();
            Console.WriteLine
                ("El número correcto era {0}\n", correctNumber);
        }
    } while (userGuess != correctNumber && userHasNotQuit);

    if (userHasNotQuit && userGuess == correctNumber)
    {
        Console.WriteLine(";Felicitades!");
    }
    else //;respuesta incorrecta!
    {
        Console.WriteLine(";Quizás la próxima vez!");
    }
}
}

```

La funcionalidad de esta aplicación será la misma que en el ejemplo con *while*. La única diferencia es la forma en que se controla el bucle. En la práctica, verá que la instrucción *while* se utiliza más a menudo que la instrucción *do/while*. Sin embargo, dado que puede controlar fácilmente la entrada en el bucle mediante la inicialización de una variable booleana, el elegir una instrucción en lugar de la otra es una elección personal.

La instrucción *for*

Con mucho, la instrucción de iteración más común que verá y utilizará es la instrucción *for*. La instrucción *for* está formada por tres partes. Una se utiliza para llevar a cabo la inicialización al principio del bucle —esta parte se lleva a cabo sólo una vez. La segunda

parte es la comprobación condicional que determina si el bucle se va a ejecutar otra vez. Y la última parte, llamada «paso», es típicamente (aunque no necesariamente) utilizada para incrementar el contador que controla la continuidad del bucle —este contador es el que se chequea en la segunda parte. La instrucción *for* tiene la siguiente estructura:

```
for (inicialización; expresión_booleana; paso)
    instrucción_interna
```

Fíjese en que cualquiera de las tres partes (*inicialización*; *expresión_booleana*; *paso*) pueden estar vacías. Cuando *expresión_booleana* se evalúa a *false*, el control se pasa desde la línea superior del bucle a la siguiente línea que sigue a *instrucción_interna*. Por lo tanto, la instrucción *for* trabaja como una instrucción *while*, excepto que le da dos partes adicionales (*inicialización* y *paso*). A continuación tenemos un ejemplo de una instrucción *for* que muestra los caracteres ASCII imprimibles:

```
using System;

class ForTestApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i++)
        {
            Console.WriteLine("{0}={1}", i, (char)i);
        }
    }
}
```

El orden de los eventos para este bucle *for* es el siguiente:

1. Se reserva memoria para una variable de tipo valor (*i*) en la pila y se inicializa a 33. Tengamos en cuenta que esta variable estará fuera de ámbito una vez termine el bucle *for*.
2. La instrucción interna se ejecutará mientras que la variable *i* tenga un valor menor de 126. En este ejemplo utilizamos una instrucción compuesta. Sin embargo, dado que este bucle *for* consiste en una instrucción de una sola línea, podríamos escribir este código sin llaves y tendríamos el mismo resultado.
3. Después de cada iteración del bucle, la variable *i* incrementará su valor en 1.

Bucles anidados

En la *instrucción_interna* de un bucle *for* puede tener otros bucles *for*, a los que generalmente nos referimos como *bucles anidados*. Utilizando el ejemplo de la sección anterior, hemos añadido un bucle anidado que hace que la aplicación imprima tres caracteres por línea en vez de uno por línea:

```

using System;

class NestedForApp
{
    const int StartChar = 33;
    const int EndChar = 125;
    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar; i <= EndChar; i += CharactersPerLine)
        {
            for (int j = 0; j < CharactersPerLine; j++)
            {
                Console.Write("{0}={1}", i+j, (char)(i+j));
            }
            Console.WriteLine("");
        }
    }
}

```

Fíjese en que la variable *i* que se definió en el bucle exterior está todavía dentro de ámbito para el bucle interno. Sin embargo, la variable *j* no está disponible en el bucle exterior.

Cómo utilizar el operador coma

Una coma puede servir no sólo como separador en listas de parámetros de métodos, sino también como operador en una instrucción *for*. Tanto en las partes *inicialización* como *paso* de una instrucción *for*, el operador coma se puede utilizar para delimitar múltiples instrucciones que se procesan secuencialmente. En el siguiente ejemplo, hemos tomado el bucle anidado del ejemplo de la sección anterior y lo hemos convertido en un bucle *for* simple utilizando el operador coma:

```

using System;

class CommaOpApp
{
    const int StartChar = 33;
    const int EndChar = 125;

    const int CharactersPerLine = 3;

    static public void Main()
    {
        for (int i = StartChar, j = 1; i <= EndChar; i++, j++)
        {
            Console.Write("{0}={1}", i, (char)i);
            if (0 == (j % CharactersPerLine))
            {
                //Nueva línea si j es divisible por 3.
                Console.WriteLine("");
            }
        }
    }
}

```

```

    }
}
}

```

Utilizar el operador coma en la instrucción *for* puede ser más eficaz, pero puede también conducir a un código feo y difícil de mantener. Incluso añadiendo constantes, el siguiente código es un ejemplo del uso del operador coma de manera técnicamente correcta pero inapropiada:

```

using System;

class CommaOp2App
{
    const int StartChar = 33;
    const int EndChar = 125;

    const int CharsPerLine = 3;
    const int NewLine = 13;
    const int Space = 32;

    static public void Main()
    {
        for (int i = StartChar, extra = Space;
            i <= EndChar;
            i++, extra = ((0 == (i - (StartChar-1)) % CharsPerLine)
                ? NewLine : Space))
        {
            Console.WriteLine("{0}={1} {2}", i, (char)i, (char)extra);
        }
    }
}

```

La instrucción *foreach*

Durante años, los lenguajes como Visual Basic han tenido una instrucción especial específicamente diseñada para la iteración sobre arrays y colecciones. C# tiene esta construcción, la instrucción *foreach*, que tiene la siguiente forma:

foreach (*tipo en expresión*)
instrucción-interna

Observe la siguiente clase array:

```

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
    }
}

```

```

        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}

```

Partiendo de las diferentes instrucciones de iteración que ya ha visto, sabe que este array puede recorrerse con cualquiera de ellas. Sin embargo, para la mayoría de programadores Java y C++, la forma más lógica de escribir esta aplicación sería la siguiente:

```

using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}

class Foreach1App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        for (int i = 0; i < myArray.words.Count; i++)
        {
            Console.WriteLine("{0}", myArray.words[i]);
        }
    }
}

```

Pero esta aproximación está cargada de problemas potenciales:

- Si la inicialización de la variable de la instrucción *for* (*i*) no se realiza adecuadamente, no se recorrerá toda la lista.
- Si la expresión booleana de la instrucción *for* no es correcta, no se recorrerá toda la lista.
- Si el paso de la instrucción *for* es incorrecto, no se recorrerá toda la lista.
- Las colecciones y los arrays tienen distintos métodos y propiedades para acceder a su conteo.
- Las colecciones y arrays tienen distinta semántica para extraer un elemento específico.
- La instrucción embebida de un bucle *for* tiene que extraer el elemento en una variable de tipo correcto.

El código puede fallar de diferentes formas. Si utiliza la instrucción *foreach*, puede evitar estos problemas e iterar a través de cualquier colección o array de forma uniforme. Con la instrucción *foreach*, el código anterior se puede reescribir como sigue:

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
    }
}

class Foreach2App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        foreach (string word in myArray.words)
        {
            Console.WriteLine("{0}", word);
        }
    }
}
```

Fíjese lo intuitivo que es utilizar la instrucción *foreach*. Le garantiza conseguir cada elemento porque no tiene que gestionar manualmente el bucle y solicitar el conteo, y la instrucción ubica el elemento en la variable que defina. Únicamente necesita referirse a esa variable en la instrucción interna.

CÓMO BIFURCAR CON INSTRUCCIONES DE SALTO

Dentro de las instrucciones internas de cualquiera de las instrucciones de iteración que se han cubierto en las secciones previas, puede controlar el flujo de ejecución con una de varias instrucciones conocidas colectivamente como instrucciones de salto: *break*, *continue*, *goto* y *return*.

La instrucción *break*

Utilice la instrucción *break* para terminar el bucle actual o la instrucción condicional en los que aparece. El control se pasa a continuación a la línea de código que sigue a la

instrucción interna del bucle o de la instrucción condicional. Tiene la sintaxis más sencilla posible para las instrucciones, la instrucción *break* no tiene paréntesis o parámetros y tiene la siguiente forma en el lugar en el que queremos salir de un bucle o instrucción condicional:

`break`

En el siguiente ejemplo, la aplicación imprimirá cada número desde 1 hasta 100 que sea divisible por 6. Sin embargo, cuando el conteo llega a 66, la instrucción *break* hará que salgamos del bucle *for*:

```
using System;

class BreakTestApp
{
    public static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (0 == i % 6)
            {
                Console.WriteLine(i);
            }

            if (i == 66)
            {
                break;
            }
        }
    }
}
```

Saliendo de bucles infinitos

Otro uso de la instrucción *break* es el de crear un bucle infinito en el que el control se transfiere fuera del bucle sólo cuando se alcanza una instrucción *break*. El siguiente ejemplo ilustra una manera de escribir el juego de adivinación de números presentado con anterioridad en este capítulo con una instrucción *break* que se utiliza para abandonar el bucle una vez que el usuario introduzca la letra Q. Observe que hemos cambiado la instrucción *while* por *while(true)*, de forma que no terminará hasta que se encuentre una instrucción *break*.

```
using System;

class InfiniteLoopApp
{
    const int MIN = 1;
    const int MAX = 10;
```

```

const string QUIT_CHAR = "Q";

public static void Main()
{
    Random rnd = new Random();
    double correctNumber;

    string inputString;
    int userGuess;

    bool correctGuess = false;
    bool userQuit = false;

    while(true)
    {
        correctNumber = rnd.NextDouble() * MAX;
        correctNumber = Math.Round(correctNumber);

        Console.Write
            ("Adivine un número entre {0} y {1}...({2} para salir)",
             MIN, MAX, QUIT_CHAR);
        inputString = Console.ReadLine();

        if (0 == string.Compare(inputString, QUIT_CHAR, true))
        {
            userQuit = true;
            break;
        }
        else
        {
            userGuess = Convert.ToInt32(inputString);
            correctGuess = (userGuess == correctNumber);

            if ((correctGuess == (userGuess == correctNumber)))
            {
                break;
            }
            else
            {
                Console.WriteLine
                    ("El número correcto era {0}\n", correctNumber);
            }
        }
    }

    if (correctGuess && !userQuit)
    {
        Console.WriteLine(";Felicitades!");
    }
    else
    {
        Console.WriteLine(";Quizás la próxima vez!");
    }
}

```

Una última cosa: podríamos haber utilizado una instrucción *for* vacía de la forma *for(;;)* en vez de *while(true)*. Funcionan de la misma manera, por lo que se convierte en un asunto de preferencias personales.

La instrucción *continue*

Al igual que la instrucción *break*, la instrucción *continue* le permite alterar la ejecución del bucle. Sin embargo, en vez de terminar la instrucción interna del bucle actual, la instrucción *continue* detiene la iteración actual y devuelve el control al principio del bucle para la siguiente iteración. En el siguiente ejemplo tenemos un array de cadenas en el que queremos comprobar si existen duplicados. Una forma de conseguirlo es iterar a lo largo del array con un bucle anidado, comparando cada elemento con el resto. Sin embargo, no debemos comparar un elemento consigo mismo para evitar un número incorrecto de duplicados. Por lo tanto, si un índice del array (*i*) es igual que el otro índice del array (*j*), significa que tenemos el mismo elemento y que no queremos compararlos. En ese caso, utilizamos la instrucción *continue* para salir de la iteración actual y pasar el control de vuelta al principio del bucle.

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;

    public MyArray()
    {
        words = new ArrayList();
        words.Add("foo");
        words.Add("bar");
        words.Add("baz");
        words.Add("bar");
        words.Add("ba");
        words.Add("foo");
    }
}

class ContinueApp
{
    public static void Main()
    {
        MyArray myArray = new MyArray();
        ArrayList dupes = new ArrayList();

        Console.WriteLine("Procesando el array...");
        for (int i = 0; i < myArray.words.Count; i++)
        {
            for (int j = 0; j < myArray.words.Count; j++)
            {
                if (i == j) continue;
            }
        }
    }
}
```

```

        if (myArray.words[i] == myArray.words[j]
            && !dupes.Contains(j))
        {
            dupes.Add(i);
            Console.WriteLine("'{0}' aparece en las líneas {1} y {2}",
                               myArray.words[i],
                               i + 1,
                               j + 1);
        }
    }
}

Console.WriteLine("Se encontraron {0} duplicados",
                  ((dupes.Count > 0) ? dupes.Count.ToString() : "no"));
}
}

```

Observe que podríamos haber utilizado un bucle *foreach* para recorrer el array. Sin embargo, en este caso particular, el objetivo ha sido seguir el rastro al elemento en el que nos encontramos, por lo que utilizar un bucle *for* ha sido la mejor solución.

La infame instrucción *goto*

Probablemente, ninguna otra construcción en la historia de la programación es tan maligna como la instrucción *goto*. Por lo tanto, antes de entrar en la sintaxis y algunos usos de la instrucción *goto*, echemos un vistazo al porqué algunas personas están tan convencidas de no utilizar esta instrucción y los tipos de problemas que se pueden resolver mediante su utilización.

La instrucción *goto*: una historia (muy) breve

La instrucción *goto* fue lanzada a la desaprobación con la publicación de un artículo titulado «Go To Statement Considered Harmful», de Edsger W. Dijkstra, en 1968. En ese momento particular de la historia de la programación, se estaba llevando a cabo un furioso debate sobre el tema de la programación estructurada. Desgraciadamente, la atención se estaba centrando menos en los aspectos globales planteados por la programación estructurada que en un pequeño detalle: si instrucciones particulares, tales como *go to* (ahora normalmente se usa la palabra reservada *goto*), deberían estar presentes en los lenguajes de programación modernos. Como sucede muy a menudo en estos casos, mucha gente siguió el consejo de Dijkstra y fue hasta el extremo, concluyendo que toda utilización de *goto* era mala y que la utilización de *goto* debería evitarse a toda costa.

El problema de utilizar *goto* no es la palabra reservada por sí misma —más bien es el uso de *goto* en los lugares inadecuados. La instrucción *goto* puede ser una herramienta útil para estructurar el flujo de programa y se puede utilizar para escribir código más expresivo que el que resulta de otros mecanismos de ramificación e iteración. Uno de tales ejemplos es el problema del «bucle y medio», como lo llamó Dijkstra. Aquí está el flujo de programa tradicional para el problema del bucle y medio en pseudocódigo:

```

bucle
  leer un valor
  si valor == centinela, entonces salir
  procesar el valor
fin_bucle

```

La salida del bucle se cumple sólo mediante la ejecución de la instrucción *salir* de la mitad del bucle. Este ciclo *bucle/salir/fin_bucle*, sin embargo, puede ser bastante molesto para algunas personas, que actuando bajo la premisa de que todo uso de *goto* es malo, escribirían este código de la siguiente manera:

```

leer un valor
mientras valor != centinela
  procesar el valor
  leer un valor
fin_mientras

```

Desgraciadamente, como señala Eric S. Roberts, de la Universidad de Stanford, esta segunda aproximación tiene dos pegas mayores. Primero, requiere la duplicación de las instrucciones necesarias para leer un valor. Siempre que duplicamos código, tenemos un problema obvio de mantenimiento, ya que cualquier cambio en una instrucción tiene que hacerse en la otra. El segundo problema es más sutil y probablemente más inevitable. La clave para escribir código sólido que sea fácil de entender, y por lo tanto de mantener, es escribir código que se lea de manera natural. En cualquier descripción que no sea código sobre lo que el código intenta hacer, uno describiría la solución de la forma siguiente: Primero, tenemos que leer un valor. Si ese valor es un centinela, paro. Si no lo es, proceso ese valor y sigo hasta el siguiente valor. Por lo tanto, es la omisión del código de la instrucción *salir* lo que es realmente antiintuitivo, ya que esta aproximación invierte la manera natural de pensar en el problema. Ahora, veamos alguna situación en la que una instrucción *goto* puede ser la mejor manera de estructurar el control del flujo de programa.

Cómo utilizar la instrucción *goto*

La instrucción *goto* puede tomar una de las siguientes formas:

```

goto identificador
goto case expresión_constante
goto default

```

En la primera forma de la instrucción *goto*, el objetivo del *identificador* es una instrucción etiqueta. La instrucción etiqueta tiene la siguiente forma:

identificador:

Si esa etiqueta no existe en el método actual, se producirá un error de compilación. Otra regla importante a recordar es que la instrucción *goto* se puede utilizar para salir de un bucle anidado. Sin embargo, si la instrucción *goto* no está dentro del ámbito de la etiqueta, se producirá un error en tiempo de compilación. Debido a esto, no puede saltar dentro de un bucle anidado.

En el siguiente ejemplo, la aplicación está iterando a lo largo de un array simple, leyendo cada valor hasta que alcanza un valor centinela, con lo cual sale del bucle. Fíjese que la instrucción *goto* actúa exactamente como la instrucción *break*, ya que hace que el flujo del programa salga del bucle *foreach*.

```
using System;
using System.Collections;

class MyArray
{
    public ArrayList words;
    public const string TerminatingWord = "stop";

    public MyArray()
    {
        words = new ArrayList();

        for (int i = 1; i <= 5; i++) words.Add(i.ToString());
        words.Add(TerminatingWord);
        for (int i = 6; i <= 10; i++) words.Add(i.ToString());
    }
}

class Goto1App
{
    public static void Main()
    {
        MyArray myArray = new MyArray();

        Console.WriteLine("Procesando array...");

        foreach (string word in myArray.words)
        {
            if (word == MyArray.TerminatingWord) goto finished;
            Console.WriteLine(word);
        }

        finished:
        Console.WriteLine("Se ha terminado el procesamiento del
                           array");
    }
}
```

A la vista del uso de la instrucción *goto*, alguien podría argumentar que una instrucción *break* podía haberse utilizado de forma igualmente efectiva y no habría sido necesaria una etiqueta. Veremos otras formas de la instrucción *goto*, y verá que los problemas al alcance de la mano se pueden resolver sólo con la instrucción *goto*.

En la sección de la instrucción *switch*, tratamos el hecho de que C# no permite el no cerrar los *case*. Incluso aunque esto se permitiera, no resolvería el siguiente problema. Digamos que tenemos una clase *Payment* (reutilizada antes en este mismo capítulo) que aceptaba distintas formas de pago: Visa, American Express, MasterCard, efectivo y anulación de cargo (básicamente un crédito). Como Visa, American Express y MasterCard son todas tarjetas de crédito, podríamos combinarlas todas en una única etiqueta *case* y procesarlas todas de la misma manera. En caso de anulación de cargo, necesitaríamos invocar métodos específicos para ello, y en caso de compra en efectivo, sólo querríamos imprimir un recibo. También querríamos imprimir un recibo en todos los casos. ¿Cómo podríamos tener tres etiquetas de caso distintas pero hacer que los dos primeros casos —los de la tarjeta de crédito y la anulación de cargo— salten ambos a la etiqueta de efectivo cuando hayan terminado? Como verá en el siguiente código, este problema es un buen ejemplo de cuándo utilizar la etiqueta *goto*:

```
using System;

enum Tenders : int
{
    ChargeOff,
    Cash,
    Visa,
    MasterCard,
    AmericanExpress
};

class Payment
{
    public Payment(Tenders tender)
    {
        this.Tender = tender;
    }

    protected Tenders tender;
    public Tenders Tender
    {
        get
        {
            return this.tender;
        }
        set
        {
            this.tender = value;
        }
    }

    protected void ChargeOff()
    {
        Console.WriteLine("Anulando cargo");
    }

    protected bool ValidateCreditCard()
    {

```

```

        Console.WriteLine("Tarjeta aprobada");
        return true;
    }

    protected void ChargeCreditCard()
    {
        Console.WriteLine("Cargo a la tarjeta realizado");
    }

    protected void PrintReceipt()
    {
        Console.WriteLine("Gracias. Esperamos de nuevo su visita");
    }

    public void ProcessPayment()
    {
        switch ((int)(this.tender))
        {
            case (int)Tenders.ChargeOff:
                ChargeOff();
                goto case Tenders.Cash;

            case (int)Tenders.Visa:
            case (int)Tenders.MasterCard:
            case (int)Tenders.AmericanExpress:
                if (ValidateCreditCard())
                    ChargeCreditCard();
                goto case Tenders.Cash;

            case (int)Tenders.Cash:
                PrintReceipt();
                break;

            default:
                Console.WriteLine("\nLo sentimos -medio de pago inválido");
                break;
        }
    }
}

class GotoCaseApp
{
    public static void Main()
    {
        Payment payment = new Payment(Tenders.Visa);
        payment.ProcessPayment();
    }
}

```

En vez de resolver el problema de forma antiintuitiva, simplemente le diremos al compilador que cuando termine un caso de tarjeta de crédito o de anulación de cargo, queremos saltar a la etiqueta de efectivo. Una última cosa en la que nos tenemos que fijar es que si saltamos fuera de una etiqueta `case` en C#, no deberíamos utilizar una instrucción `break`, que provocaría un error de compilador por «código inalcanzable».

La última forma de la instrucción *goto* permite saltar a la etiqueta *default* en una instrucción *switch*, dándole por lo tanto otro mecanismo de escribir un bloque de código sencillo que se puede ejecutar como resultado de múltiples evaluaciones de la instrucción *switch*.

La instrucción *return*

La instrucción *return* tiene dos funciones. Especifica un valor que se va a devolver a quien invoca el código que se ejecuta actualmente (cuando no se ha definido que el código actual ha de devolver *void*) y causa una vuelta inmediata al invocador. La instrucción *return* se define con la siguiente sintaxis:

```
return [expresión_return]
```

Cuando el compilador encuentra la instrucción *return* de un método que especifica una *expresión_return*, evalúa si la *expresión_return* puede convertirse de forma implícita en una forma compatible con el valor de retorno definido en el método actual. El resultado de esta conversión es lo que se devuelve a quien invoca la función.

Cuando utilice la instrucción *return* con manejo de excepciones, se tendrá que asegurar de que conoce algunas reglas. Si la instrucción *return* está en un bloque *try* que contiene un bloque *finally* asociado, el control se pasa realmente a la primera línea del bloque *finally*, y cuando este bloque de código termina, el control seguirá hacia atrás en la cadena de esta forma hasta que el último bloque *finally* se haya ejecutado.

RESUMEN

Las instrucciones condicionales de C# le permiten controlar el flujo de programa. Las tres categorías distintas de instrucciones de flujo incluyen las instrucciones de selección, tales como *if* y *switch*, las instrucciones de iteración (*while*, *for* y *foreach*) y las distintas instrucciones de salto (*break*, *continue*, *goto* y *return*). La decisión de cuál es la mejor instrucción para utilizar basándose en el material de este capítulo le ayudará a escribir aplicaciones mejor estructuradas y mantenibles.

Capítulo 12

Manejo de errores con excepciones

Uno de los principales objetivos del Common Language Runtime (CLR) de .NET está dedicado a los errores en tiempo de ejecución, bien para evitarlos (mediante características como la gestión automática de memoria y recursos cuando se utiliza código gestionado) o al menos capturarlos en tiempo de compilación (mediante un sistema fuertemente tipado). Sin embargo, algunos errores sólo se pueden capturar en tiempo de ejecución y por lo tanto se debe utilizar un mecanismo consistente para tratar estos errores que sirva para todos los lenguajes que se ajustan a la Common Language Specification (CLS). A tal efecto, este capítulo se centra en el sistema de gestión de errores implementado por el CLR —el manejo de excepciones.

En este capítulo, en primer lugar aprenderá los mecanismos generales y la sintaxis básica del manejo de excepciones. Una vez que tenga el conocimiento básico, verá cómo el manejo de excepciones se compara con los métodos predominantes hoy para el manejo de errores y descubrirá las ventajas que tiene el manejo de excepciones sobre estas otras técnicas. Luego nos adentraremos en los aspectos específicos de manejo de excepciones de .NET, tales como la clase *Exception*, y cómo derivar sus propias clases de excepción. Finalmente, la última parte de este capítulo tratará el tema del diseño adecuado de su sistema para que utilice manejo de excepciones.

VISIÓN GENERAL DEL MANEJO DE EXCEPCIONES

Las excepciones son condiciones de error que surgen cuando el flujo normal que sigue el código —esto es, una serie de invocaciones a método en la pila de invocaciones— es poco práctico o imprudente. Es imperativo entender aquí la diferencia entre una excepción y un evento esperado (tal y como sucede al alcanzar el final de un archivo). Si tiene un método que está leyendo secuencialmente un archivo, sabe que en algún momento se va a alcanzar el final de éste. Por lo tanto, este evento es apenas *excepcional* por su

naturaleza y con seguridad no evita que la aplicación no pueda continuar. Sin embargo, si está intentando leer un archivo y el sistema operativo le avisa de que hay un error de disco, esto es realmente una situación excepcional y definitivamente afectará el flujo normal de los intentos de sus métodos de seguir leyendo el archivo.

La mayoría de las excepciones también implica otro problema: el *contexto*. Veamos un ejemplo. Si asumimos que está escribiendo código fuertemente cohesivo —código en que un método es responsable de una acción—, su (pseudo)código podría parecerse a este:

```
public void Foo()
{
    File file = OpenFile(String fileName);
    while (!file.IsEOF())
    {
        String record = file.ReadRecord();
    }
    CloseFile();
}

public void OpenFile(String fileName)
{
    //Intenta bloquear y abrir un archivo
}
```

Fíjese que si el método *OpenFile* falla, no puede manejar el error. Esto es porque sólo es responsable de abrir archivos. No puede determinar si la incapacidad de abrir el archivo especificado constituye un error catastrófico o una molestia menor. Por lo tanto, *OpenFile* no puede manejar la condición de error porque se dice que el método no está en el contexto adecuado.

Esta es la razón más importante para la existencia del manejo de excepciones: un método determina que una condición de excepción se ha alcanzado y sucede que no se encuentra en el contexto correcto para tratar el error. Avisa al entorno de ejecución que hay un error. El entorno de ejecución recorre en orden inverso la pila de invocaciones hasta que encuentra un método que puede tratar de forma apropiada la condición de error. Obviamente, esto se convierte en algo mucho más importante si el código en ejecución tiene cinco niveles de profundidad de invocación alcanzando una condición de error en el método que se encuentra en el quinto nivel de profundidad siendo el método en el primer nivel de profundidad el único que puede tratar correctamente el error. Veamos la sintaxis utilizada para el manejo de excepciones.

SINTAXIS BÁSICA DE MANEJO DE EXCEPCIONES

El manejo de excepciones consiste en sólo cuatro palabras reservadas: *try*, *catch*, *throw* y *finally*. La forma en que funcionan las palabras clave es sencilla y directa. Cuando un método fracasa en su objetivo y no puede continuar —esto es, cuando detecta una situación excepcional—, devuelve una excepción al método que lo ha invocado mediante la utilización de la palabra *throw*. El método que invoca, si asumimos que tiene el contexto suficiente para tratar la excepción, recibe a continuación esta excepción mediante la

palabra reservada *catch* y decide qué curso de acción tomar. En las siguientes secciones veremos la semántica de lenguaje, que gobierna cómo lanzar y capturar excepciones, así como algunos fragmentos de código que ilustrarán cómo funciona.

Cómo lanzar una excepción

Cuando un método necesita notificar al método que invoca que ha sucedido un error, utiliza la palabra reservada *throw* de la siguiente forma:

```
throw instrucción
throw instrucciónopt
```

Entraremos en las diferentes formas en que podemos lanzar excepciones un poco más tarde. Por ahora es suficiente con darse cuenta de que cuando lanza una excepción, se requiere que se lance un objeto de tipo *System.Exception* (o una clase derivada). A continuación tenemos un ejemplo de un método que ha determinado que ha sucedido un error irreparable y que necesita lanzar una excepción al método que lo invoca. Observe cómo se instancia un nuevo objeto excepción y luego se lanza ese objeto recién creado hacia atrás en la pila de invocaciones.

```
public void SomeMethod()
{
    //Se determina que ha sucedido algún error
    throw new Exception();
}
```

Cómo capturar una excepción

Obviamente, dado que un método puede lanzar una excepción, debe haber una parte recíproca en esta ecuación, donde algo tiene que capturar la excepción lanzada. La palabra reservada *catch* define un bloque de código cuya función es procesar una excepción de un tipo dado cuando ésta se captura. El código de este bloque es llamado *manejador de excepción*.

Una cosa que debemos tener presente es que no todos los métodos tienen que tratar con cualquier posible excepción lanzada, especialmente porque un método podría no tener el contexto necesario para hacer nada con la información de error. Después de todo, la clave del manejo de excepciones es que los errores sean manejados por el código que tiene el contexto suficiente para hacerlo —vea «Cómo manejar errores en el contexto correcto», más adelante en este capítulo. Por ahora, tratemos con situaciones en las que un método va a capturar cualquier excepción lanzada desde el método que está siendo invocado. Utilizamos dos palabras clave para capturar una excepción: *try* y *catch*.

Para capturar una excepción, necesita empaquetar el código que intenta ejecutar en un bloque *try* y luego, en un bloque *catch*, especificar qué tipos de excepciones puede manejar para el código ubicado dentro de ese bloque *try*. Todas las instrucciones en el

bloque *try* se procesarán en el orden habitual, excepto si se lanza una excepción en alguno de los métodos que se invoquen. Si sucede eso, el control se pasa a la primera línea del bloque *catch* apropiado. Por «bloque *catch* apropiado» queremos referirnos al bloque que se ha definido para capturar el tipo de excepción que se ha lanzado. A continuación tenemos un ejemplo de un método (*Foo*) que llama y captura una excepción que lanza otro método (*Bar*):

```
public void Foo()
{
    try
    {
        Bar();
    }
    catch(System.Exception e)
    {
        //Hacer algo con condición de error.
    }
}
```

Ahora su siguiente pregunta podría ser: «¿Qué sucede si *Bar* lanza una excepción y *Foo* no la captura?». (Esto podría ser también el caso de una invocación a *Bar* sin que ésta se encuentre en un bloque *try*). El resultado depende del diseño de la aplicación. Cuando se lanza una excepción, el control se pasa hacia atrás en la pila de invocaciones hasta que se encuentra un método que tiene un bloque *catch* para el tipo de excepción lanzada. Si no se localiza un método con un bloque *catch* apropiado, la aplicación anula su ejecución. Por lo tanto, si un método llama a otro método que lanza una excepción, el diseño del sistema deber ser tal que algún método de la pila de invocaciones debe capturar la excepción.

Cómo relanzar una excepción

Habrás veces, después de que un método haya capturado una excepción y realizado todo lo que pueda en su contexto, en las que a continuación se *relance* la excepción hacia atrás en la pila de invocaciones. Esto se hace de manera muy sencilla mediante la palabra reservada *throw*. A continuación se muestra un ejemplo de cómo hacerlo:

```
using System;

class RethrowApp
{
    static public void Main()
    {
        RethrowApp rethrow = new RethrowApp();

        try
        {
            rethrow.Foo();
        }
    }
}
```

```

        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception)
        {
            //Manejar el error.
            throw;
        }
    }

    public void Bar()
    {
        throw new Exception("lanzada por Rethrow.Bar");
    }
}

```

En este ejemplo, *Main* invoca a *Foo*, que invoca a *Bar*. *Bar* lanza una excepción que captura *Foo*. En este momento, *Foo* hace algún preprocesamiento y luego relanza la excepción hacia atrás en la pila de invocaciones, hacia *Main*, utilizando la palabra reservada *throw*.

Poniendo orden con *finally*

Un asunto complicado con el manejo de excepciones es el asegurar que un fragmento del código siempre se ejecuta con independencia de si se captura o no una excepción. Por ejemplo, suponga que reserva memoria para un recurso, tal como un dispositivo físico o un archivo de datos. Luego suponga que abre este dispositivo e invoca un método que lanza una excepción. Independientemente de si su método puede continuar trabajando con el recurso, todavía necesita liberar la memoria o cerrar ese recurso. Es en este caso cuando la palabra reservada *finally* se utiliza, como podemos ver:

```

using System;

public class ThrowException1App
{
    public static void ThrowException()
    {
        throw new Exception();
    }

    public static void Main()
    {
    }
}

```

```

    {
        try
        {
            Console.WriteLine("try...");
        }
        catch(Exception e)
        {
            Console.WriteLine("catch...");
        }
        finally
        {
            Console.WriteLine("finally");
        }
    }
}

```

Como puede observar, la existencia de la palabra *finally* evita que tenga que poner este código, que siempre se ha de ejecutar en el bloque *catch* y después de los bloques *try/catch*. Ahora, a pesar de que se lance una excepción, el código del bloque *finally* se ejecutará.

COMPARACIÓN DE TÉCNICAS DE MANEJO DE ERRORES

Ahora que hemos visto las bases del lanzamiento y captura de excepciones, pasaremos unos minutos comparando las diferentes aproximaciones para manejar errores que se utilizan en los lenguajes de programación.

La aproximación estándar para el manejo de errores ha sido típicamente devolver un código de error al mensaje que invoca. El método que invoca será el responsable de descifrar el valor devuelto y actuar en consecuencia. El valor devuelto puede ser tan sencillo como un tipo básico de C o C++, o puede ser un puntero a un objeto más robusto que contiene toda la información necesaria para analizar y entender completamente el error. Las técnicas de manejo de errores de diseño más elaboradas implican un subsistema de errores completo en el que el método invocado indica la condición de error al subsistema y luego devuelve un código de error al que invoca. El invocador, a continuación, invoca una función global exportada por el subsistema de error para determinar la causa del último error registrado por éste. Puede encontrar un ejemplo de esta aproximación en el SDK de Microsoft Open Database Connectivity (ODBC). Sin embargo, con independencia de la semántica exacta, el concepto básico permanece igual: el método invocador, de alguna forma, invoca un método e inspecciona el valor devuelto para comprobar el éxito o fallo relativo del método invocado. Esta aproximación, si bien ha sido el estándar durante muchos años, presenta un conjunto de defectos importantes debido a una serie de aspectos. Las siguientes secciones describen algunas de las formas en que el manejo de excepciones proporciona unos beneficios tremendos frente a los códigos de retorno.

Beneficios del manejo de excepciones frente a los códigos de retorno

Cuando utilizamos códigos de retorno, el método invocado devuelve un código de error y la condición de error es tratada por el método invocador. Dado que el manejo del error ocurre fuera del ámbito del método invocado, no hay garantías de que el invocador compruebe el código de error devuelto. Como ejemplo, digamos que escribe una clase llamada *CommaDelimitedFile* que recubre la funcionalidad de leer y escribir en archivos estándares delimitados por comas. Parte de lo que tendría que hacer su clase sería exponer los métodos incluidos para abrir y leer datos del archivo. Si utilizáramos el método antiguo de código de retorno para informar de errores, estos métodos devolverían algún tipo de variable que tendría que ser comprobada por el invocador para comprobar el éxito del método invocado. Si el usuario de su clase llamara al método *CommaDelimitedFile.Open* y luego intentara invocar al método *CommaDelimitedFile.Read* sin comprobar si la invocación a *Open* terminó con éxito, podría —y probablemente lo haría en caso de que hiciéramos una demo a nuestro cliente más importante— obtener resultados indeseables. Sin embargo, si el método *Open* de la clase lanzara una excepción, el invocador estaría forzado a tratar con el hecho de que el método *Open* ha fallado. Esto se debe a que cada vez que un método lanza una excepción, el control se devuelve a la pila de invocaciones hasta que es capturada. A continuación mostramos un ejemplo de cómo podría ser este código:

```
using System;

class ThrowException2App
{
    class CommaDelimitedFile
    {
        protected string fileName;

        public void Open(string fileName)
        {
            this.fileName = fileName;
            //Intento de abrir un archivo
            //y lanzamiento de excepción por condición de error
            throw new Exception("apertura fallida");
        }

        public bool Read(string record)
        {
            //Código para leer un archivo
            return false; //EOF
        }
    }

    public static void Main()
    {
        try
        {
            Console.WriteLine("intentando abrir un archivo");
```

```

        CommaDelimitedFile file = new CommaDelimitedFile();
        file.Open("c:\\test.csv");

        string record = "";

        Console.WriteLine("leyendo del archivo");

        while (file.Read(record) == true)
        {
            Console.WriteLine(record);
        }

        Console.WriteLine("terminada la lectura del archivo");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

```

En este ejemplo, si el método *CommaDelimitedFile.Open* o el método *CommaDelimitedFile.Read* lanzan una excepción, el método invocador se ve forzado a tratar con ello. Si el método invocador no captura la excepción y ningún otro método en el código en ejecución intenta capturar una excepción de este tipo, la aplicación anulará su ejecución. Preste atención especial al hecho de que dado que el método *Open* está ubicado en un bloque, no se intentaría una lectura inválida (utilizando nuestro ejemplo en el que el método *Open* ha lanzado una excepción). Esto se debe a que el control de programa se pasaría desde la invocación a *Open* en el bloque *try* a la primera línea del bloque *catch*. Por lo tanto, uno de los mayores beneficios del manejo de excepciones sobre los códigos de retorno es que las excepciones son, desde el punto de vista del programa, más difíciles de ignorar.

Cómo manejar errores en el contexto adecuado

Un principio general de la buena programación es la práctica de una *cohesión firme*, que se refiere al objetivo o propósito de un método dado. Los métodos que demuestran cohesión firme son aquellos que llevan a cabo una tarea sencilla. El principal beneficio de utilizar cohesión firme al programar es que cuando un método realiza una única acción es probablemente más portable y se puede utilizar en distintos escenarios. Un método que ejecuta una acción sencilla es con certeza más fácil de depurar y mantener. Sin embargo, el código que está firmemente cohesionado causa un problema importante cuando se trata del manejo de errores. Veamos un ejemplo que ilustra el problema y cómo lo resuelve el manejo de excepciones.

En este ejemplo se utiliza una clase (*AccessDatabase*) para generar y manipular bases de datos Microsoft Access. Digamos que esta clase tiene un método estático que se llama *GenerateDatabase*. Dado que este método se utilizaría para crear nuevas bases de datos Access, tendría que realizar varias tareas para crear la base de datos. Por ejemplo, tendría

que crear el archivo físico de base de datos, crear las tablas especificadas (incluyendo cualquier fila y columna que necesite su aplicación) y definir cualesquiera índices y relaciones. El método *GenerateDatabase* podría incluso tener que crear algunos usuarios y permisos predeterminados.

El problema de diseño desde el punto de vista del programa es el siguiente: si sucede un error en el método *CreateIndexes*, ¿qué método lo manejaría y cómo? Obviamente, en algún momento, el método que originalmente invocó el método *GenerateDatabase* tendría que manejar el error, pero ¿cómo podría hacerlo? No tendría ni idea de cómo manejar un error que se produjo en una invocación de algún método que se encuentra a unos cuantos niveles de profundidad más allá en el código. Como hemos visto, se dice que el método que invoca no está en el contexto correcto para manejar el error. En otras palabras, el único método que podría proporcionar lógicamente cualquier información significativa del error sobre el error mismo es el método que falló. Dicho esto, si utilizáramos los códigos de error en nuestra clase *AccessDatabase*, cada método que se ejecutara tendría que comprobar los errores en el código que *podrían* devolverse en cualquier otro método. Un problema obvio asociado es que el método invocador potencialmente tendría que manejar un número ingente de códigos de error. Además, el mantenimiento sería difícil. Cada vez que se añadiera una condición de error a cualquier método en la ruta de ejecución del código, cualquier otra instancia en la aplicación en que un método lo invocara, tendría que actualizarse para manejar el nuevo código de error. No es necesario decir que esto no es una proposición económica en términos del costo total de propiedad (TCO).

El manejo de excepciones resuelve todos estos problemas permitiendo que el método invocador capture un tipo dado de excepción. En el ejemplo que estamos utilizando, si se derivara una clase llamada *AccessDatabaseException* de *Exception*, se podría utilizar para cualesquiera tipos de errores que sucedan dentro de cualquier método *AccessDatabase*. (Trataremos la clase *Exception* y derivaremos nuestras propias clases de excepción más tarde en este capítulo en «Cómo utilizar la clase *System.Exception*»). Dicho esto, en caso de que el método *CreateIndexes* fallara, construiríamos una excepción del tipo *AccessDatabaseException*. El método invocador capturaría esa excepción y sería capaz de inspeccionar el objeto *Exception* para descifrar exactamente qué falló. Por lo tanto, en vez de gestionar cualquier tipo posible de código de retorno que pudiera devolver *GenerateDatabase* y cualquiera de los métodos que invocara el método invocador, se podría asegurar que si *cualquiera* de los métodos en la cadena de invocaciones fallara, se devolvería la información de error adecuada. El manejo de excepciones proporciona una ventaja adicional: dado que la información de error está contenida en una clase, se pueden añadir nuevas condiciones de error y el método invocador se mantendrá sin cambios. ¿Y no era la extensibilidad —ser capaz de construir algo y luego añadirle cosas sin cambiar o echar a perder el código existente— una de las premisas originales de la programación orientada a objetos con las que empezamos? Por estos motivos, el concepto de capturar y tratar errores en el contexto adecuado es la ventaja más significativa de utilizar el manejo de excepciones.

Cómo mejorar la legibilidad del código

Cuando se utiliza código de manejo de excepciones, la legibilidad del código mejora enormemente. Esto se asocia directamente con una reducción de los costes en términos de mantenibilidad del código. La manera en que se manejan los códigos de retorno frente a la sintaxis de la gestión de excepciones es el motivo de esta mejora. Si solía devolver códigos de retorno con el método *AccessDatabase.GenerateDatabase* mencionado anteriormente, se necesitaría un código similar al siguiente para manejar las condiciones de error:

```
public bool GenerateDatabase()
{
    if (CreatePhysicalDatabase())
    {
        if (CreateTables())
        {
            if (CreateIndexes())
            {
                return true;
            }
            else
            {
                //Manejar el error.
                return false;
            }
        }
        else
        {
            //Manejar el error.
            return false;
        }
    }
    else
    {
        //Manejar el error.
        return false;
    }
}
```

Añada algunas otras validaciones al código precedente y terminará con una cantidad tremenda de código de validación de error mezclado con su lógica de negocio. Si indenta su código 4 espacios por bloque, el primer carácter de una línea de código podría no aparecer hasta la columna 20 o posterior. Nada de esto es un desastre para el código en sí, pero hace el código más difícil de leer y de mantener, y la conclusión que podemos sacar es que el código que es difícil de mantener es un terreno abonado para los defectos. Veamos cómo sería el ejemplo anterior si se utilizara gestión de excepciones:

```
//Código invocador.
try
{
    AccessDatabase accessDb = new AccessDatabase();
    accessDb.GenerateDatabase();
}
```

```

    }
    catch(Exception e)
    {
        //Analizar la excepción capturada.
    }

    //Definición del método AccessDatabase.GenerateDatabase.
    public void GenerateDatabase()
    {
        CreatePhysicalDatabase();
        CreateTables();
        CreateIndexes();
    }

```

Observe lo limpio y elegante que resulta la segunda solución. Esto se debe a que el código de detección y recuperación de errores ya no está mezclado con la lógica del método invocador. Dado que el manejo de excepciones ha hecho este código más directo, el mantener el código se ha convertido en algo mucho más fácil.

Cómo lanzar excepciones desde los constructores

Una de las mayores ventajas que tienen las excepciones sobre otras técnicas de manejo de errores está en el área de construcción de objetos. Dado que un constructor no puede devolver valores, simplemente no hay ningún medio fácil e intuitivo de indicar al método que invoca al constructor que hay un error durante la construcción del objeto. Las excepciones, sin embargo, se pueden utilizar, ya que el método invocador sólo necesita envolver la construcción del objeto en un bloque *try*, como se muestra en el siguiente código:

```

try
{
    //Si el objeto AccessDatabase falla al construir
    //adecuadamente y lanza una excepción, ahora se capturará
    AccessDatabase accessDb = new AccessDatabase();
}
catch(Exception e)
{
    //Inspeccionar la excepción capturada.
}

```

LA CLASE *SYSTEM.EXCEPTION*

Como mencionamos anteriormente, todas las excepciones que se lancen deberán ser del tipo (o derivadas de) *System.Exception*. En realidad, la clase *System.Exception* es la clase base de varias clases de excepción que se pueden utilizar en su código C#. La mayoría de las clases que heredan de *System.Exception* no añaden ninguna funcionalidad a la clase base. Entonces, ¿por qué nos molesta C# con clases derivadas si éstas no van a diferir significativamente de su clase base? La razón es que un bloque sencillo *try* puede tener múltiples bloques *catch* indicando cada uno de ellos un tipo específico de excepción.

(Veremos esto en breve). Esto permite que el código maneje las distintas excepciones de forma aplicada a un tipo de excepción específico.

Cómo construir un objeto *Exception*

Como indicamos a continuación, hay cuatro constructores diferentes para la clase *System.Exception*:

```
public Exception ();
public Exception(String);
protected Exception(SerializationInfo, StreamingContext);
public Exception(String, Exception);
```

El primer constructor de la lista anterior es el constructor por defecto. No tiene parámetros y simplemente toma de forma predeterminada las variables miembro. La excepción se lanza típicamente de la siguiente manera:

```
//Se alcanza la condición de error.
throw new Exception();
```

El segundo constructor de excepción toma como parámetro un valor *String* que identifica un mensaje de error y es la forma que ha visto en la mayoría de ejemplos en este capítulo. Este mensaje es recuperado por el código capturando la excepción a través de la propiedad *System.Exception.Message*. A continuación mostramos un ejemplo sencillo de ambos extremos de la propagación de excepciones:

```
using System;

class ThrowException3App
{
    class FileOps
    {
        public void FileOpen(String fileName)
        {
            //...
            throw new Exception("Oh, problemas");
        }

        public void FileRead()
        {
        }
    }

    public static void Main()
    {
        //Código que captura la excepción.
        try
        {
            FileOps fileOps = new FileOps();
```

```

        fileOps.FileOpen("c:\\test.txt");
        fileOps.FileRead();
    }
    catch(System.Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
}

```

El tercer constructor inicializa una instancia de la clase *Exception* con datos serializados.

Finalmente, el último constructor le permite especificar no sólo un mensaje de error, sino también lo que es conocido como *excepción interna*. La razón para esto es que cuando manejamos excepciones, a veces querrá *manejar* la excepción parcialmente en un nivel de la pila de invocaciones antes de pasarla hacia arriba a través de ésta. Veamos un ejemplo de cómo podría utilizar esta característica.

Digamos que para facilitar la carga en su clase cliente, decide lanzar sólo un tipo de excepción. De esta forma, el cliente sólo tiene que capturar una excepción y hacer referencia a la propiedad interna *InnerException* de la excepción. Esto tiene el beneficio añadido de que si decide modificar un método dado para lanzar un nuevo tipo de excepción después de haber escrito el código del cliente, éste sólo se tendrá que actualizar si se quiere hacer algo específico con esta excepción.

En el siguiente ejemplo, fíjese que el código del cliente captura la excepción de más alto nivel *System.Exception* e imprime un mensaje contenido en el objeto excepción interno a la excepción. Si en fechas posteriores el método *DoWork* lanza otras clases de excepciones —siempre y cuando lo haga como excepciones internas de un objeto *System.Exception*—, el código de cliente seguirá funcionando.

```

using System;
using System.Globalization;

class FooLib
{
    protected bool IsValidParam(string value)
    {
        bool success = false;

        if (value.Length == 3)
        {
            char c1 = value[0];
            if (Char.IsNumber(c1))
            {
                char c2 = value[2];
                if (Char.IsNumber(c2))
                {
                    if (value[1] == '.')
                        success = true;
                }
            }
        }
    }
}

```

```

        return success;
    }

    public void DoWork(string value)
    {
        if (!IsValidParam(value)) throw new Exception
            ("", new FormatException("Parámetro especificado inválido"));
        Console.WriteLine("Trabajo realizado con '{0}'", value);
    }
}

class FooLibClientApp
{
    public static void Main(string[] args)
    {
        FooLib lib = new FooLib();
        try
        {
            lib.DoWork(args[0]);
        }
        catch (Exception e)
        {
            Exception inner = e.InnerException;
            Console.WriteLine(inner.Message);
        }
    }
}

```

Cómo utilizar la propiedad *StackTrace*

Otra propiedad interesante de la clase *System.Exception* es la propiedad *StackTrace*. La propiedad *StackTrace* le permite determinar —en cualquier punto dado en el que tenga un objeto *System.Exception* válido— cómo es la pila de invocaciones actual. Eche un vistazo al siguiente código:

```

using System;

class StackTraceTestApp
{
    public void Open(String fileName)
    {
        Lock(fileName);
        //...
    }
    public void Lock(String fileName)
    {
        //Se lanza la condición de error.
        throw new Exception("fallo al bloquear archivo");
    }
    public static void Main()
    {
        StackTraceTestApp test = new StackTraceTestApp();

        try

```

```

    {
        test.Open("c:\\test.txt");
        //Utilizar el archivo.
    }
    catch(Exception e)
    {
        Console.WriteLine(e.StackTrace);
    }
}

```

En este ejemplo se imprime lo siguiente:

```
at StackTraceTest.Main()
```

Por lo tanto, la propiedad *StackTrace* devuelve la pila de invocaciones en el momento en que se captura la excepción, lo que puede ser útil para escenarios de registro de actividad y depuración.

Cómo capturar múltiples tipos de excepciones

En varias situaciones, podría querer un bloque *try* para capturar distintos tipos de excepciones. Por ejemplo, podría tener un método cuya documentación indica que puede lanzar varios tipos de excepciones diferentes, o podría tener varias invocaciones a métodos en un único bloque *try*, donde la documentación de cada método invocado indicara que es capaz de lanzar un tipo diferente de excepción. Esto se maneja añadiendo un bloque *catch* distinto para cada tipo de excepción que necesitara manejar el código:

```

try
{
    Foo(); //Puede lanzar la excepción FooException.
    Bar(); //Puede lanzar la excepción BarException.
}
catch(FooException e)
{
    //Manejar el error.
}
catch(BarException e)
{
    //Manejar el error.
}
catch(Exception e)
{
}

```

Cada tipo de excepción se puede manejar con un bloque *catch* distinto (junto con el código de manejo de errores). Sin embargo, el hecho de que la clase base de las excepciones es manejada la última es un detalle extremadamente importante aquí. Obviamente, dado que todas las excepciones se derivan de *System.Exception*, si colocáramos este bloque *catch* el primero, los otros bloques nunca se alcanzarían. En ese caso, el siguiente código sería rechazado por el compilador:

```

try
{
    Foo(); //Puede lanzar la excepción FooException.
    Bar(); //Puede lanzar la excepción BarException.
}
catch(Exception e)
{
    /***ERROR -ÉSTE NO COMPILARÁ
}
catch(FooException e)
{
    //Manejar el error.
}
catch(BarException e)
{
    //Manejar el error.
}

```

Cómo derivar sus propias clases *Exception*

Como hemos indicado, podría haber veces en que quisiéramos proporcionar información extra o dar formato a una excepción antes de que se envíe al código cliente. También podríamos proporcionar una clase base para esa situación, de forma que nuestra clase pueda publicar que sólo lanza un tipo de excepción. De esta forma, el cliente sólo necesita preocuparse por capturar esta clase base.

Otro ejemplo más en que podríamos querer derivar nuestra propia clase *Exception* es si quisiéramos llevar a cabo alguna acción —tal como registrar el evento o enviar un correo a alguien en el departamento de soporte— cada vez que se lanzara una excepción. En este caso, derivaríamos nuestra propia clase *Exception* y colocaríamos el código necesario en el constructor de clase de la siguiente manera:

```

using System;

public class TestException : Exception
{
    //Probablemente tendría métodos y propiedades extra
    //aquí que aumentan la excepción .NET de la que deriva.

    //Constructores de la clase base Exception.
    public TestException()
        :base(){}
    public TestException(String message)
        :base(message){}
    public TestException(String message, Exception innerException)
        :base(message, innerException){}
}
public class DerivedExceptionTestApp
{
    public static void ThrowException()
    {
        throw new TestException("condición de error");
    }
}

```

```

    }
    public static void Main()
    {
        try
        {
            ThrowException();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}

```

CÓMO DERIVAR SUS PROPIAS CLASES *EXCEPTION*

Aunque no es una regla estricta, es una buena práctica de programación —y consistente con la mayoría del código C# que verá— nombrar sus clases de excepción de forma que el nombre termine con la palabra «Exception». Por ejemplo, si quisiera derivar una clase *Exception* para una clase llamada *MyFancyGraphics*, podría nombrar la clase *MyFancyGraphicsException*.

Otra regla general cuando cree o derive sus propias clases de excepción es implementar los tres constructores de la clase *System.Exception*. Una vez más, esto no es absolutamente necesario, pero mejora la consistencia con otro código C# que su cliente pudiera estar utilizando.

Este código generará la siguiente salida. Observe que el método *ToString* resulta de la combinación de propiedades que se están mostrando: la representación textual del nombre de clase de la excepción, la cadena de mensaje que se pasa al constructor de la excepción y el *StackTrace*.

```

TestException: error condition
at DerivedExceptionTestApp.Main()

```

CÓMO DISEÑAR SU CÓDIGO CON MANEJO DE EXCEPCIONES

Hasta ahora, hemos cubierto los conceptos básicos de la utilización del manejo de excepciones y la semántica relacionada con el lanzamiento y captura de éstas. Ahora veamos una faceta igualmente importante del manejo de excepciones: cómo entender la forma de diseñar su sistema teniendo en mente el manejo de excepciones. Suponga que tiene tres métodos: *Foo*, *Bar* y *Baz*. *Foo* invoca a *Bar*, que a su vez invoca a *Baz*. Si *Baz* publica el hecho de lanzar una excepción, ¿tiene que capturar *Bar* esa excepción incluso si no puede o no quiere hacer nada con ella? ¿Cómo se debería dividir el código con relación a los bloques *try* y *catch*?

Consideraciones de diseño con el bloque *try*

Sabe cómo capturar una excepción que un método invocado podría lanzar, y sabe que el control escala la pila de invocaciones hasta que encuentra un bloque *catch* apropiado. Por esto, la cuestión es: ¿debería un bloque *try* capturar toda excepción posible que un método incluido dentro de él pudiera lanzar? La respuesta es no, para obtener los mayores beneficios del uso del manejo de excepciones en sus aplicaciones: programar poco y reducir los costes de mantenimiento. El siguiente ejemplo ilustra un programa en que un *catch* es vuelto a lanzar para ser manejado por otro bloque *catch*.

```
using System;

class WhenNotToCatchApp
{
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Bar()
    {
        try
        {
            Baz();
        }
        catch(Exception e)
        {
            //Bar debería capturar esta excepción porque
            //no hace nada excepto relanzarlo.

            throw;
        }
    }

    public void Baz()
    {
        throw new Exception("Excepción lanzada originalmente por Baz");
    }

    public static void Main()
    {
        WhenNotToCatchApp test = new WhenNotToCatchApp();
        test.Foo(); //Este método finalmente imprimirá
                   //el mensaje de error.
    }
}
```

En este ejemplo, *Foo* captura la excepción que *Baz* lanza, aun cuando no hace nada, excepto relanzarla hacia arriba, hasta *Bar*. *Bar* luego captura la excepción relanzada y hace algo con la información —en ese caso, mostrar la propiedad de mensaje de error del objeto *Exception*. A continuación indicamos algunas razones por las que los métodos que relanzan las excepciones no deberían en principio capturarlas:

- Dado que el método no hace nada con la excepción, se quedaría con un bloque *catch*, que en el mejor de los casos es superfluo. Obviamente, nunca es una buena idea que exista código cuando no tiene absolutamente ninguna función.
- Si el tipo de excepción que está siendo lanzado por *Baz* cambiara, tendría que volver y cambiar tanto el bloque *catch* de *Bar* como el bloque *catch* de *Foo*. ¿Por qué ponerse en una situación en la que tendría que alterar código que ni siquiera hace nada?

Dado que el CLR automáticamente escala la pila de invocaciones hasta que un método captura la excepción, los métodos intermedios pueden pasar por alto las excepciones que no pueden procesar. Asegúrese en su diseño de que sólo *algunos* métodos capturan la excepción, porque, como se mencionó antes, si una excepción se lanza y no se encuentra ningún bloque *catch* en la pila de invocaciones actual, la aplicación anulará su ejecución.

Consideraciones de diseño con el bloque *catch*

El único código que debería aparecer en un bloque *catch* es el código que procesará, al menos parcialmente la excepción capturada. Por ejemplo, a veces un método capturará una excepción, hará lo que pueda para procesarla y luego la relanzará para que pueda haber un manejo posterior de la misma. El siguiente ejemplo ilustra esto. *Foo* ha invocado a *Bar*, que hará algún trabajo con la base de datos en representación de *Foo*. Dado que se están utilizando control mediante consolidación en la base de datos (*commit*) y transacciones, *Bar* tiene que capturar cualquier error que suceda y dar marcha atrás a los cambios que no se hayan llegado a consolidar en la base de datos antes de devolver la excepción a *Foo*.

```
using System;

class WhenToCatchApp
{
    public void Foo()
    {
        try
        {
            Bar();
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }

    public void Bar()
    {

```

```

    try
    {
        //Invocar a un método para fijar el "límite del commit"
        Console.WriteLine("fijando límites del commit");

        //Invocar a Baz para salvar los datos.
        Console.WriteLine("invocando a Baz para almacenar los datos");
        Baz();

        Console.WriteLine("haciendo commit de los datos");
    }
    catch(Exception)
    {
        //En este caso, Bar capturaría la excepción
        //porque va a hacer algo significativo
        //(restaurar o consolidar los cambios en la base de datos).

        Console.WriteLine("deshaciendo los cambios que no se han" +
                           "realizado y relanzando la excepción");

        throw;
    }
}

public void Baz()
{
    throw new Exception("fallo de bd en Baz");
}

public static void Main()
{
    WhenToCatchApp test = new WhenToCatchApp();
    test.Foo(); //Este método imprimirá finalmente
               //el mensaje de error.
}

```

Bar tenía que capturar la excepción para realizar su propio procesamiento de errores. Cuando haya terminado, relanzará la excepción hacia arriba en la pila de invocaciones, donde *Foo* la capturará y será capaz de hacer su propio procesamiento de errores. Esta habilidad de cada nivel de la pila de excepciones de hacer exactamente la cantidad de procesamiento de la que es capaz, dado su contexto, es un ejemplo de lo que hace el manejo de excepciones tan importante para el diseño de su sistema.

RESUMEN

La sintaxis de manejo de excepciones de C# es sencilla y directa, e implementar el manejo de excepciones en su aplicación es tan fácil como diseñar sus métodos de antemano. Un objetivo primario del CLR .NET es ayudarle a evitar los errores en tiempo de ejecución mediante un sistema fuertemente tipado y un manejo elegante de los errores que ocurran en tiempo de ejecución. La utilización de código de manejo de excepciones en sus aplicaciones las hará más robustas, merecedoras de confianza y, en último término, más fáciles de mantener.

Sobrecarga de operadores y conversiones definidas por el usuario

En el Capítulo 7, «Propiedades, arrays e indizadores», aprendió cómo utilizar el operador [] con una clase para, a través de un programa, indizar un objeto como si fuera un array. En este capítulo veremos dos características de C# relacionadas de forma muy próxima que proporcionan la habilidad de crear interfaces de estructura y de clase que se utilizarán de forma más fácil e intuitiva: sobrecarga de operadores y conversiones definidas por el usuario. Empezaremos con una visión general de la sobrecarga de operadores en términos de los beneficios que le proporciona y luego veremos la sintaxis actual para redefinir los comportamientos por defecto de operadores, así como ejemplos realistas de aplicaciones en que sobrecargaremos el operador + para agregar múltiples objetos *Invoice*. Después de eso, veremos un listado de qué operadores unitarios y binarios son sobrecargables, así como algunas restricciones implicadas. El tratamiento de la sobrecarga de operadores terminará con algunas guías de diseño que se deben tener en cuenta cuando decida si sobrecargará un operador en sus clases. Una vez terminemos con la sobrecarga de operadores, aprenderá un nuevo concepto, las conversiones definidas por usuario. Una vez más, empezaremos con aproximaciones generales, explicando lo básico de esta característica, y luego nos adentraremos en una clase que ilustra cómo puede utilizar las conversiones para permitir la conversión de tipo de un *struct* o *class* en un *struct*, *class* u otro tipo básico C#.

SOBRECARGA DE OPERADORES

La sobrecarga de operadores permite que los operadores C# existentes se redefinan para que se puedan utilizar con tipos definidos por el usuario. La sobrecarga de operadores se ha llamado «azúcar sintáctico», debido al hecho de que el operador sobrecargado es

simplemente otra forma de invocar a un método. También se ha dicho que la característica no añade nada fundamental al lenguaje. Aunque esto es técnicamente cierto, la sobrecarga de operadores realmente ayuda en uno de los aspectos más importantes de la programación orientada a objetos: la abstracción.

Suponga que quiere agregar una colección de facturas para un cliente particular. Mediante el uso de la sobrecarga de operadores, puede escribir código similar al siguiente, en el que el operador `+=` se sobrecarga:

```
Invoice summaryInvoice = new Invoice();
foreach (Invoice invoice in customer.GetInvoices())
{
    summaryInvoice += invoice;
}
```

Los beneficios de tal código incluyen una sintaxis realmente natural y el hecho de que al cliente se le abstraiga de tener que entender los detalles de implementación sobre cómo se están agregando las facturas. Dicho de forma sencilla, la sobrecarga de operadores ayuda a la creación de software menos caro de escribir y mantener.

Sintaxis y ejemplo

Como he dicho, la sobrecarga de operadores es un medio de invocar un método. Para redefinir un operador para una clase sólo necesita seguir el siguiente patrón, donde *op* es el operador que está sobrecargando:

```
public static valret operatorop (objeto1 [, objeto2])
```

Tenga en mente los siguientes hechos cuando utilice sobrecarga de operadores:

- Todos los métodos sobrecargados deben definirse como *public* y *static*.
- Técnicamente, *valret* (el valor de retorno) puede ser de cualquier tipo. Sin embargo, es práctica común devolver el tipo para el que el método está siendo definido, con la excepción de los operadores *true* y *false*, que siempre deberían devolver un valor booleano.
- El número de parámetros pasados (*objeto1*, *objeto2*) depende del tipo de operador que se esté sobrecargando. Si se está sobrecargando un operador unitario (un operador que tiene un único operando), habrá un único parámetro. Si se está sobrecargando un operador binario (un operador que toma dos operandos), se le pasarán dos parámetros.
- En el caso de un operador unitario, el parámetro del método ha de ser del mismo tipo que el de la clase o struct en que se encuentre. En otras palabras, si redefine el operador unitario `!` para una clase llamada *Foo*, ese método debe tomar como su único parámetro una variable de tipo *Foo*.
- Si el operador que se está sobrecargando es un operador binario, el primer parámetro tiene que ser del mismo tipo que la clase en la que se encuentra y el segundo parámetro puede ser de cualquier tipo.

En el pseudocódigo de la sección previa utilicé el operador `+=` con una clase *Invoice*. Por razones que pronto entenderá, realmente no se pueden sobrecargar estos operadores compuestos. Puede sobrecargar el operador «base»; en este caso, el `+`. A continuación se muestra la sintaxis utilizada para definir el método *operator+* de la clase *Invoice*:

```
public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
{
    //Crea un nuevo objeto.
    //Suma el contenido deseado desde
    //invoice1 al nuevo objeto Invoice.
    //Suma el contenido deseado desde
    //invoice2 al nuevo objeto Invoice.
    //Devuelve el nuevo objeto Invoice.
}
```

Veamos ahora un ejemplo más sustancial, uno con dos clases principales: *Invoice* e *InvoiceDetailLine*. La clase *Invoice* tiene una variable miembro de tipo *ArrayList* que representa una colección de todas las líneas de detalle de una factura. Para permitir la agregación de líneas de detalle para múltiples facturas hemos sobrecargado el operador `+` (vea el método *operator+* debajo para más detalles). El método *Invoice.operator+* crea un nuevo objeto *Invoice* e itera a lo largo del array que contiene las líneas de detalle de las facturas en ambos objetos *Invoice* (los operandos), añadiendo cada línea de detalle al nuevo objeto *Invoice*. Este objeto *Invoice* es devuelto al llamante a continuación. Obviamente, en un módulo del mundo real para facturación esto sería mucho más complejo, pero la idea aquí es mostrar de manera algo realista cómo se podría utilizar la sobrecarga de operadores.

```
using System;
using System.Collections;

class InvoiceDetailLine
{
    double lineTotal;
    public double LineTotal
    {
        get
        {
            return this.lineTotal;
        }
    }

    public InvoiceDetailLine(double LineTotal)
    {
        this.lineTotal = LineTotal;
    }
}

class Invoice
{
    public ArrayList DetailLines;

    public Invoice()
```

```

    {
        DetailLines = new ArrayList();
    }

    public void PrintInvoice()
    {
        Console.WriteLine("\nNúm. Línea \tTotal");

        int i = 1;
        double total = 0;
        foreach (InvoiceDetailLine detailLine in DetailLines)
        {
            Console.WriteLine("{0}\t\t{1}", i++, detailLine.LineTotal);
            total += detailLine.LineTotal;
        }

        Console.WriteLine("====\t\t==");
        Console.WriteLine("Total\t\t{1}", i++, total);
    }

    public static Invoice operator+ (Invoice invoice1, Invoice invoice2)
    {
        Invoice returnInvoice = new Invoice();

        foreach (InvoiceDetailLine detailLine in invoice1.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }

        foreach (InvoiceDetailLine detailLine in invoice2.DetailLines)
        {
            returnInvoice.DetailLines.Add(detailLine);
        }

        return returnInvoice;
    }
}

class InvoiceAddApp
{
    public static void Main()
    {
        Invoice i1 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i1.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice i2 = new Invoice();
        for (int i = 0; i < 2; i++)
        {
            i2.DetailLines.Add(new InvoiceDetailLine(i + 1));
        }

        Invoice summaryInvoice = i1 + i2;

        summaryInvoice.PrintInvoice();
    }
}

```

Operadores que se pueden sobrecargar

Sólo los siguientes operadores unitarios y binarios se pueden sobrecargar.

Operadores unitarios: `+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`

Operadores binarios: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, `<=`

NOTA La coma se utiliza aquí para separar los distintos operadores sobrecargables. El operador coma, que se utiliza en la instrucción *for* y en la invocación a métodos, no se puede sobrecargar.

Restricciones en la sobrecarga de operadores

No es posible sobrecargar el operador de asignación `=`. Sin embargo, cuando sobrecarga un operador binario, su equivalente para la asignación compuesta está implícitamente sobrecargado. Por ejemplo, si sobrecarga el operador `+`, el operador `+=` está implícitamente sobrecargado, debido al hecho de que se invocará el método *operator+*.

Los operadores `[]` no se pueden sobrecargar. Sin embargo, como se vio en el Capítulo 7, la indización de objetos definida por usuario se puede realizar mediante el uso de indizadores.

Los paréntesis que se utilizan para las conversiones de tipo tampoco se pueden sobrecargar. En su lugar debería utilizar operadores de conversión, que también se conocen como conversiones definidas por usuario, el objeto de la segunda mitad de este capítulo.

Los operadores que actualmente no están definidos en el lenguaje C# no se pueden sobrecargar. No puede, por ejemplo, definir `**` como un mecanismo para tener potencias, ya que C# no tiene un operador `**` definido. Además, tampoco se puede modificar la sintaxis de un operador. No se puede cambiar el operador binario `*` para que tome tres parámetros, cuando, por definición, su sintaxis necesita dos. Finalmente, no se puede alterar la precedencia de un operador. Las reglas de precedencia son estáticas —vea el Capítulo 10, «Expresiones y operadores», para saber más de estas reglas.

Guías de diseño

Ha visto qué es la sobrecarga de operadores y cómo utilizarla en C#, así que examinemos un aspecto a menudo ignorado de esta útil característica: las guías de diseño. Debe mantenerse alejado de la tendencia natural de querer utilizar una nueva característica por el mero hecho de utilizarla. Este fenómeno es a veces conocido como «la solución que busca al problema». Siempre es una buena aproximación al diseño recordar el dicho «El código es más leído que escrito». Tenga la clase del cliente en mente cuando esté de-

terminando si procede y cuándo sobrecargar un operador o un conjunto de operadores. Esto es una regla general: debería sobrecargar un operador sólo si hace que la interfaz de clase sea más intuitiva de utilizar. Por ejemplo, tiene sentido común sumar facturas juntas.

Además, no olvide que tiene que pensar como si fuera un cliente de su clase. Digamos que está escribiendo una clase *Invoice* y que quiere que el cliente sea capaz de descontar de la factura. Usted y yo podríamos pensar que se añadirá un elemento de línea de crédito a la factura, pero el punto de encapsulación que se busca es que sus clientes no tienen por qué conocer los detalles de implementación exactos de la clase. Por lo tanto, sobrecargar el operador `*` —como se muestra aquí— podría ser una buena idea, porque serviría para hacer la interfaz de clase *Invoice* más natural e intuitiva:

```
invoice *= .95; //5% de descuento.
```

CONVERSIONES DEFINIDAS POR EL USUARIO

Anteriormente mencioné que los paréntesis utilizados para la conversión de tipos no se pueden sobrecargar y que las conversiones definidas por el usuario se pueden utilizar en su lugar. En resumen, las conversiones definidas por el usuario le permiten declarar conversiones sobre estructuras o clases tales de forma que una *class* o *struct* se puedan convertir a otras estructuras, clases o tipos básicos C#. ¿Por qué y cuándo querría hacer esto? Digamos que necesitara utilizar las escalas térmicas de Celsius y Fahrenheit en su aplicación de forma que pudiera convertir fácilmente entre las dos. Creando conversiones definidas por el usuario, podría utilizar la siguiente sintaxis:

```
Fahrenheit f = 98.6F;  
Celsius c = (Celsius)f; //Conversión definida por el usuario.
```

Aunque el ejemplo anterior no proporciona ningún beneficio frente a la sintaxis del siguiente ejemplo, es más intuitiva de escribir y más fácil de leer:

```
Fahrenheit f = 98.6F;  
Celsius c = f.ConvertToCelsius();
```

Sintaxis y ejemplo

La sintaxis de la conversión definida por el usuario utiliza la palabra reservada *operator* para declarar conversiones definidas por el usuario:

```
public static implicit operator conv-tipo-sal (conv-tipo-ent operando)  
public static explicit operator conv-tipo-sal (conv-tipo-ent operando)
```

Hay sólo un par de reglas que afectan a la sintaxis de la definición de conversiones:

- Cualquier método de conversión para un *struct* o *class* —pueden definirse tantos como desee— deben ser *static*.
- Las conversiones se deben definir como *implicit* o *explicit*. La palabra reservada *implicit* significa que no es necesaria la conversión de tipo por parte del cliente y ocurrirá automáticamente. Por el contrario, la palabra reservada *explicit* significa que el cliente ha de hacer una conversión de tipo sobre el valor al tipo deseado.
- Todas las conversiones han de tomar (como parámetro) el tipo sobre el que se está definiendo la conversión o debe devolver ese tipo.
- Al igual que la sobrecarga de operadores, la palabra reservada *operator* se utiliza en la signatura del método de conversión pero sin ningún operador añadido.

La primera vez que lea estas reglas, probablemente no tenga ni idea de qué hacer, así que veremos un ejemplo para aclarar las ideas. En este ejemplo tenemos dos estructuras (*Celsius* y *Fahrenheit*) que permiten que el cliente convierta un valor de tipo *float* a cualquier escala térmica. Primero presentaremos la estructura *Celsius* y haremos algunos comentarios sobre ella, y luego verá la aplicación operativa completa.

```
struct Celsius
{
    public Celsius(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}
```

La primera decisión que se puede observar es la decisión de utilizar una estructura en lugar de una clase. No teníamos ninguna razón real para hacerlo, más que el hecho de que la utilización de clases es más costosa que la de las estructuras —en términos de cómo se gestiona la memoria de las clases—, y una clase no es necesaria realmente aquí porque la estructura *Celsius* no necesita ninguna característica específica de las clases C#, como por ejemplo la herencia.

A continuación, fíjese que hemos declarado un constructor que toma un número en coma flotante como su único parámetro. Este valor se almacena en una variable miembro llamada *Temp*. Ahora observe el operador de conversión definido inmediatamente después del constructor de la estructura. Este es el método que será invocado cuando el cliente intente hacer una conversión de tipo de un número en coma flotante a *Celsius*, o

utilizar un número en coma flotante en un lugar, como un método, donde se espera una estructura *Celsius*. Este método no tiene que hacer mucho, y en realidad es un código formulista que se puede utilizar en las conversiones más básicas. Aquí simplemente instanciamos una estructura *Celsius* y luego devolvemos esta estructura. Esa invocación a `return` es la que hará que se invoque el último método definido en la estructura. Como puede ver, el método simplemente proporciona la fórmula matemática para convertir desde un valor Fahrenheit a uno Celsius.

A continuación tenemos la aplicación completa, incluyendo una estructura *Fahrenheit*:

```
using System;

struct Celsius
{
    public Celsius(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Celsius(float temp)
    {
        Celsius c;
        c = new Celsius(temp);
        return(c);
    }

    public static implicit operator float(Celsius c)
    {
        return((((c.temp - 32) / 9) * 5));
    }

    public float temp;
}

struct Fahrenheit
{
    public Fahrenheit(float temp)
    {
        this.temp = temp;
    }

    public static implicit operator Fahrenheit(float temp)
    {
        Fahrenheit f;
        f = new Fahrenheit(temp);
        return(f);
    }

    public static implicit operator float(Fahrenheit f)
    {
        return((((f.temp * 9) / 5) + 32));
    }

    public float temp;
```

```

}

class Temp1App
{
    public static void Main()
    {
        float t;

        t=98.6F;
        Console.Write("Conversión de {0} a grados centígrados = ", t);
        Console.WriteLine((Celsius)t);

        t=0F;
        Console.Write("Conversión de {0} a grados Fahrenheit = ", t);
        Console.WriteLine((Fahrenheit)t);
    }
}

```

Si compila este código y ejecuta la aplicación, obtendrá la siguiente salida:

```

Conversión de 98.6 a grados centígrados = 37
Conversión de 0 a grados Fahrenheit = 32

```

Esto funciona bastante bien, y el poder escribir *(Celsius)98.6F* es ciertamente más intuitivo que algunos métodos estáticos de clase. Pero fíjese que sólo se pueden pasar valores de tipo *float* a estos métodos de conversión. Para la aplicación anterior, lo siguiente no se compilará:

```

Celsius c = new Celsius(55);
Console.WriteLine((Fahrenheit)c);

```

Además, al no haber un método de conversión *Celsius* que tome una estructura *Fahrenheit* (o viceversa), el código ha de asumir que el valor que se está pasando es un valor que necesita conversión. En otras palabras, si invoco *(Celsius)98.6F*, recibiré el valor 37. Sin embargo, si al valor luego le volvemos a aplicar el método de conversión, el método de conversión no tiene forma de saber que el valor ya ha sido convertido y en realidad representa lógicamente una temperatura *Celsius* válida —para el método de conversión sólo es un número en coma flotante. Como resultado, el valor se vuelve a convertir de nuevo. Por lo tanto, necesitamos modificar la aplicación de forma que cada estructura pueda tomar como parámetro válido la otra estructura.

Cuando consideramos cómo hacer esto originalmente, mientras lo pensábamos, nos preocupaba lo difícil que sería la tarea. En realidad, veremos que es extremadamente fácil. Aquí está el código revisado con los resultados correspondientes:

```

using System;

class Temperature
{
    public Temperature(float Temp)
    {

```

```

        this.temp = Temp;
    }
    protected float temp;
    public float Temp
    {
        get
        {
            return this.temp;
        }
    }
}

class Celsius : Temperature
{
    public Celsius(float Temp)
        : base(Temp){}

    public static implicit operator Celsius(float Temp)
    {
        return new Celsius(Temp);
    }

    public static implicit operator Celsius(Fahrenheit F)
    {
        return new Celsius(F.Temp);
    }

    public static implicit operator float(Celsius C)
    {
        return((((C.temp - 32) / 9) * 5));
    }
}

class Fahrenheit : Temperature
{
    public Fahrenheit(float Temp)
        : base(Temp){}

    public static implicit operator Fahrenheit(float Temp)
    {
        return new Fahrenheit(Temp);
    }

    public static implicit operator Fahrenheit(Celsius C)
    {
        return new Fahrenheit(C.Temp);
    }

    public static implicit operator float(Fahrenheit F)
    {
        return((((F.temp * 9) / 5) + 32));
    }
}

class Temp2App
{
    public static void DisplayTemp(Celsius Temp)

```

```

    {
        Console.WriteLine("Conversión de {1} grados {0} a grados Fahrenheit = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Fahrenheit)Temp);
    }

    public static void DisplayTemp(Fahrenheit Temp)
    {
        Console.WriteLine("Conversión de {1} grados {0} a grados centígrados = ",
            Temp.ToString(), Temp.Temp);
        Console.WriteLine((Celsius)Temp);
    }

    public static void Main()
    {
        Fahrenheit f = new Fahrenheit(98.6F);
        DisplayTemp(f);

        Celsius c = new Celsius(0F);
        DisplayTemp(c);
    }
}

```

La primera cosa que podemos notar es que se han cambiado los tipos *Celsius* y *Fahrenheit* de *struct* a *class*. Se hizo así para tener dos ejemplos: uno utilizando *struct* y otro utilizando *class*. Pero una razón más práctica para hacerlo así ha sido compartir la variable *temp* haciendo que las clases *Celsius* y *Fahrenheit* deriven de la misma clase base *Temperature*. Podemos utilizar ahora también el método heredado *ToString* (de *System.Object*) en la salida de la aplicación.

La otra diferencia que debemos observar es el añadido de una conversión para cada escala térmica que toma como parámetro un valor de la otra escala térmica. Observe cómo se parece el código entre los dos métodos de conversión *Celsius*:

```

public static implicit operator Celsius(float temp)
{
    Celsius c;
    c = new Celsius(temp);
    return(c);
}

public static implicit operator Celsius(Fahrenheit f)
{
    Celsius c;
    c = new Celsius(f.temp);
    return(c);
}

```

Las únicas tareas que tuvimos que hacer de forma diferente fueron cambiar el parámetro que se pasaba y recuperar la temperatura del objeto pasado, en vez de tener un valor directamente en el código de tipo *float*. Por esto indicamos con anterioridad lo fácil y formulistas que son los métodos de conversión una vez que se conocen las bases.

RESUMEN

La sobrecarga de operadores y las conversiones definidas por el usuario son útiles para crear interfaces intuitivas para sus clases. Cuando utilice operadores sobrecargados, tenga en cuenta las restricciones asociadas mientras diseña sus clases. Por ejemplo, aunque no se puede sobrecargar el operador de asignación `=`, cuando se sobrecarga un operador binario, su asignación compuesta está implícitamente sobrecargada. Siga las guías de diseño para decidir cuándo utilizar cada característica. Tenga los clientes de la clase en cuenta cuando determine si se sobrecarga o no un operador o un conjunto de operadores. Con un poquito de intuición sobre cómo utilizarán los clientes sus clases, podrá utilizar estas eficaces características para definir sus clases de forma que ciertas operaciones se puedan llevar a cabo con una sintaxis más natural.

Delegados y manejadores de eventos

Otra innovación útil del lenguaje C# es algo llamado delegado, que básicamente realiza la misma función que los punteros a funciones en C++. Sin embargo, los delegados son objetos de memoria gestionada y de tipo seguro. Esto significa que el entorno de ejecución garantiza que un delegado apunta a un método válido, lo que adicionalmente implica que posee todos los beneficios de los punteros a funciones sin ninguno de los peligros asociados, tales como direcciones de memoria inválidas o que un delegado corrompa la memoria de otros objetos. En este capítulo, veremos los delegados, cómo se comparan con las interfaces, la sintaxis utilizada para definirlos y los diferentes problemas, para los que fueron diseñados, que resuelven. También veremos varios ejemplos de utilización de delegados, tanto con métodos de devolución de llamada como de manejo de eventos asíncronos.

En el Capítulo 9, «Interfaces», vimos cómo se definían e implementaban las interfaces en C#. Como recordará, desde un punto de vista conceptual, las interfaces son sencillamente contratos entre dos trozos diferentes de código. Sin embargo, las interfaces se asemejan a clases en el hecho de que se definen en tiempo de compilación y pueden incluir métodos, propiedades, indizadores y eventos. Los delegados, por otro lado, hacen referencia sólo a métodos simples que se definen en tiempo de ejecución. Los delegados tienen dos usos principales en la programación C#: devoluciones de llamada y manejo de eventos. Empecemos hablando de los métodos de devolución de llamada.

DELEGADOS COMO MÉTODOS DE DEVOLUCIÓN DE LLAMADA

Los métodos de devolución de llamada, utilizados extensivamente en la programación para Microsoft Windows, se utilizan cuando se necesita pasar un puntero a función a otra función que posteriormente le devolverá la llamada (a través del puntero pasado). Un ejemplo sería la función *EnumWindows* de la API Win32. Esta función enumera todas las

ventanas de primer nivel en la pantalla, invocando la función proporcionada por cada ventana. La devolución de llamada sirve a muchos propósitos, aunque los siguientes son los más comunes:

- **Procesamiento asíncrono.** Los métodos de devolución de llamada se utilizan en procesamiento asíncrono cuando el código que se invoca tardará mucho tiempo en procesar la llamada. El escenario, típicamente, funciona así: el código cliente hace una invocación a un método, pasándole el método de devolución de llamada. El método que se invoca arranca un hilo de ejecución y devuelve el control inmediatamente. A continuación, el nuevo hilo de ejecución hace la mayoría del trabajo, invocando la función de devolución de llamada cuando sea necesario. Esto tiene el beneficio obvio de permitir que el cliente continúe el procesamiento sin tener que quedarse bloqueado por una llamada síncrona potencialmente larga.
- **Insertar código de cliente en el trayecto del código de una clase.** Otro uso común de los métodos de devolución de llamada es cuando una clase permite que el cliente especifique un método que se invocará para realizar algún procesamiento personalizado. Veamos un ejemplo en Windows para ilustrar esto. Si utilizamos la clase *Listbox* en Windows, puede especificar que los elementos se ordenen ascendente o descendientemente. Además de otras opciones básicas de ordenación, la clase *Listbox* no puede realmente darle libertad para hacer muchas más cosas y seguir siendo una clase genérica. Por lo tanto, la clase *Listbox* le permite especificar una función de devolución de llamada para ordenar. De esta forma, cuando *Listbox* ordene los elementos, invocará la función de devolución de llamada y su código podrá entonces hacer la ordenación personalizada que necesite.

Veamos ahora un ejemplo de definición y utilización de un delegado. En este ejemplo, tenemos una clase gestora de base de datos que realiza el seguimiento de todas las conexiones activas en la base de datos y que proporciona un método para enumerar esas conexiones. Si asumimos que el gestor de bases de datos está en un servidor remoto, podría ser una buena decisión de diseño hacer el método asíncrono y permitir que el cliente proporcione un método de devolución de llamada. Fíjese que para una aplicación real, normalmente se crearía la aplicación como aplicación multihilo para hacerla realmente asíncrona. Sin embargo, para hacer que el ejemplo sea sencillo —y porque todavía no hemos cubierto la ejecución multihilo—, no entraremos en ello.

En primer lugar, definiremos dos clases principales: *DBManager* y *DBConnection*.

```
class DBConnection
{
    :
}

class DBManager
{
    static DBConnection[] activeConnections;
    :
```

```

public delegate void EnumConnectionsCallback(DBConnection connection);
public static void EnumConnections(EnumConnectionsCallback callback)
{
    foreach (DBConnection connection in activeConnections)
    {
        callback(connection);
    }
}

```

El método *EnumConnectionsCallback* es el delegado y está definido colocando la palabra reservada *delegate* delante de la signatura del método. Puede ver que este delegado se ha definido indicando que devuelve *void* y que toma un parámetro simple: un objeto *DBConnection*. El método *EnumConnections* se define a continuación, de forma que toma un método *EnumConnectionsCallback* como su único parámetro. Para invocar el método *DBManager.EnumConnections*, sólo tenemos que pasarle un delegado *DBManager.EnumConnectionsCallback* instanciado.

Para hacer esto, cree un nuevo delegado con *new*, pasándole el nombre del método que tiene la misma signatura que el delegado como nombre. A continuación se muestra un ejemplo:

```

DBManager.EnumConnectionsCallback myCallback =
    new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

DBManager.EnumConnections(myCallback);

```

Observe que también puede combinarlo en una simple invocación de la siguiente forma:

```

DBManager.EnumConnections(new
    DBManager.EnumConnectionsCallback(ActiveConnectionsCallback));

```

Esto es todo lo referente a la sintaxis básica de los delegados. Veamos a continuación la aplicación de ejemplo completa:

```

using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }
    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
    }
}

```

```

        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] =
                new DBConnection("DBConnection" + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class DelegatelApp
{
    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine("Invocado método de devolución de llamada para"
            + connection.name);
    }

    public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnectionsCallback myCallback =
            new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

        DBManager.EnumConnections(myCallback);
    }
}

```

Si compilamos y ejecutamos esta aplicación, conseguimos la siguiente salida:

```

Invocado método de devolución de llamada para DBConnection 1
Invocado método de devolución de llamada para DBConnection 2
Invocado método de devolución de llamada para DBConnection 3
Invocado método de devolución de llamada para DBConnection 4
Invocado método de devolución de llamada para DBConnection 5

```

CÓMO DEFINIR DELEGADOS COMO MIEMBROS ESTÁTICOS

Dado que suena un poco raro que los clientes tengan que instanciar el delegado cada vez que éste se vaya a utilizar, C# permite definir el método que se vaya a utilizar en la creación del delegado como miembro de clase estático. A continuación puede ver el ejemplo de la sección anterior modificado para utilizar ese formato. Observe que el delegado se define en el ejemplo como un miembro de clase estático llamado *myCallback*, y observe también que se puede utilizar en el método *Main* sin necesidad de que el cliente lo instancie.

```
using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] =
                new DBConnection("DBConnection" + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}
```

```

    }

class Delegate2App
{
    public static DBManager.EnumConnectionsCallback myCallback =
        new DBManager.EnumConnectionsCallback(ActiveConnectionsCallback);

    public static void ActiveConnectionsCallback(DBConnection connection)
    {
        Console.WriteLine ("Invocado método de devolución de llamada para"
            + connection.name);
    }

    public static void Main()
    {
        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(myCallback);
    }
}

```

NOTA Dado que la convención de nomenclatura estándar para los delegados es anexar la palabra *Callback* al método que toma el delegado como parámetro, es fácil equivocarse y utilizar el nombre del método en vez del nombre del delegado. En ese caso, tendrá errores un tanto confusos en tiempo de compilación que afirman que ha utilizado un método cuando se esperaba una clase. Si tiene este error, recuerde que el problema real es que ha especificado un método en lugar de un delegado.

CÓMO CREAR DELEGADOS SÓLO CUANDO SE NECESITEN

En los dos ejemplos que hemos visto hasta ahora, el delegado se crea con independencia de si se utiliza o no. Esto funciona adecuadamente en esos ejemplos porque sabíamos que siempre se invocarían. Sin embargo, cuando usted defina sus propios delegados, es importante tener en cuenta cuándo crearlos. Digamos, por ejemplo, que la creación de un delegado en particular es una operación costosa en tiempo y no es algo que quiera hacer de forma gratuita. En estas situaciones en que sabe que el cliente no invocará un método de devolución de llamada dado, puede posponer la creación del delegado hasta que sea realmente necesario mediante el recubrimiento de su instanciación en una propiedad. Para ilustrar cómo hacer esto, a continuación se muestra una modificación de la clase *DBManager* que utiliza una propiedad de sólo lectura —porque sólo está presente un método de obtención— para instanciar el delegado. El delegado no se creará hasta que se haga referencia a esta propiedad.

```

using System;

class DBConnection
{
    public DBConnection(string name)
    {
        this.name = name;
    }

    protected string Name;
    public string name
    {
        get
        {
            return this.Name;
        }
        set
        {
            this.Name = value;
        }
    }
}

class DBManager
{
    static DBConnection[] activeConnections;
    public void AddConnections()
    {
        activeConnections = new DBConnection[5];
        for (int i = 0; i < 5; i++)
        {
            activeConnections[i] =
                new DBConnection("DBConnection" + (i + 1));
        }
    }

    public delegate void EnumConnectionsCallback(DBConnection connection);
    public static void EnumConnections(EnumConnectionsCallback callback)
    {
        foreach (DBConnection connection in activeConnections)
        {
            callback(connection);
        }
    }
}

class Delegate3App
{
    public DBManager.EnumConnectionsCallback myCallback
    {
        get
        {
            return new DBManager.EnumConnectionsCallback
                (ActiveConnectionsCallback);
        }
    }

    public static void ActiveConnectionsCallback(DBConnection connection)
    {

```

```

        Console.WriteLine ("Invocado método de devolución de llamada para" +
                           connection.name);
    }

    public static void Main()
    {
        Delegate3App app = new Delegate3App();

        DBManager dbMgr = new DBManager();
        dbMgr.AddConnections();

        DBManager.EnumConnections(app.myCallback);
    }
}

```

COMPOSICIÓN DE DELEGADOS

La habilidad de componer delegados —mediante la creación de un único delegado a partir de múltiples delegados— es una de esas características que al principio no parece muy cómoda, pero que si alguna vez necesita, agradecerá que el equipo de diseño de C# pensara en ella. Veamos algunos ejemplos en los que la composición de delegados es útil. En el primer ejemplo, tiene un sistema de distribución y una clase que itera a través de los sectores para una sede dada, invocando un método de devolución de llamada para cada sector que tiene un valor «en reserva» menor de 50. En un ejemplo de distribución más realista, la fórmula tendría en cuenta no sólo «en reserva», sino también «en pedido» o «en tránsito» con relación a los tiempos totales que tarda en completarse el proceso, y restaría el nivel de existencias de seguridad. Pero hagámoslo sencillo: si el valor de un sector en reserva es menor de 50, se ha producido una excepción.

Para complicarlo, queremos que se invoquen dos métodos distintos si un determinado sector está por debajo del nivel de existencias: queremos que se registre el evento y luego queremos enviar un correo electrónico al responsable de compras. Por lo tanto, echemos un vistazo a cómo se crea, mediante programación, un delegado compuesto sencillo a partir de múltiples delegados:

```

using System;
using System.Threading;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }

    protected string Sku;
    public string sku
    {

```

```

        get
        {
            return this.Sku;
        }
        set
        {
            this.Sku = value;
        }
    }

    protected int OnHand;
    public int onhand
    {
        get
        {
            return this.OnHand;
        }
        set
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;

    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Sede" + (i + 1));

            Thread.Sleep(10); //La aleatoriedad toma como semilla el
                               //tiempo

            parts[i] = part;
            Console.WriteLine("Sumando a sede '{0}' el disponible = {1}",
                              part.sku, part.onhand);
        }
    }

    public delegate void OutOfStockExceptionMethod(Part par);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcesando inventario...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                ("Disponible {0} ({1}) por debajo del límite {2}",
                 part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

```

```

    }
}

class CompositeDelegateApp
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tregistrando evento...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\tenviando correo al responsable de compras...");
    }

    public static void Main()
    {
        InventoryManager mgr = new InventoryManager();

        InventoryManager.OutOfStockExceptionMethod LogEventCallback =
            new InventoryManager.OutOfStockExceptionMethod(LogEvent);

        InventoryManager.OutOfStockExceptionMethod
            EmailPurchasingMgrCallback = new
                InventoryManager.OutOfStockExceptionMethod
                    (EmailPurchasingMgr);

        InventoryManager.OutOfStockExceptionMethod
            OnHandExceptionEventsCallback =
                EmailPurchasingMgrCallback + LogEventCallback;

        mgr.ProcessInventory(OnHandExceptionEventsCallback);
    }
}

```

Si ejecutamos esta aplicación, se producen resultados similares a los siguientes:

```

Sumando a sede 'Sede 1' el disponible = 16
Sumando a sede 'Sede 2' el disponible = 98
Sumando a sede 'Sede 3' el disponible = 65
Sumando a sede 'Sede 4' el disponible = 22
Sumando a sede 'Sede 5' el disponible = 70

Procesando el inventario...
Disponible 1 (16) por debajo del límite 50
    registrando evento...
    enviando correo al responsable de compras...
Disponible 4 (22) por debajo del límite 50
    registrando evento...
    enviando correo al responsable de compras...

```

Por lo tanto, utilizando esta característica del lenguaje, podemos discernir de forma dinámica qué métodos conforman un método de devolución de llamada, agregando esos métodos en un simple delegado y pasando el delegado compuesto como si fuera uno

único. El entorno de ejecución automáticamente controlará que todos los métodos se invoquen en secuencia. Además, puede eliminar los delegados que desee del compuesto utilizando el operador menos.

Sin embargo, el hecho de que estos métodos se invoquen en orden secuencial nos lleva a una pregunta importante: ¿por qué no puede simplemente encadenar los métodos juntos haciendo que cada método de forma sucesiva invoque al otro? En el ejemplo de esta sección, donde sólo tenemos dos métodos y ambos se invocan emparejados, podríamos hacerlo. Pero compliquemos aún más el ejemplo. Digamos que tenemos varias sedes de tiendas de forma que cada sede dicta qué métodos se invocarán. Por ejemplo, Location1 podría ser el almacén, así que querríamos que se registrara el evento y se enviara un correo electrónico al responsable de compras, mientras que un sector que está por debajo de la cantidad mínima en reserva para cualquier otra sede resultaría en un evento que se registraría y un correo electrónico enviado al responsable de la tienda.

Podemos afrontar estos requisitos mediante la creación de un delegado compuesto basado en la sede que se está procesando. Sin delegados, tendríamos que escribir un método que no sólo tendría que determinar qué métodos invocar, sino también tendría que llevar un seguimiento de qué métodos se han invocado y cuáles se han de invocar durante la secuencia de invocaciones. Como puede ver en el siguiente código, los delegados hacen esta operación, que es potencialmente compleja, muy sencilla.

```
using System;

class Part
{
    public Part(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.OnHand = (int)d;
    }

    protected string Sku;
    public string sku
    {
        get
        {
            return this.Sku;
        }
        set
        {
            this.Sku = value;
        }
    }

    protected int OnHand;
    public int onhand
    {
        get
```

```

        {
            return this.OnHand;
        }
        set
        {
            this.OnHand = value;
        }
    }
}

class InventoryManager
{
    protected const int MIN_ONHAND = 50;

    public Part[] parts;
    public InventoryManager()
    {
        parts = new Part[5];
        for (int i = 0; i < 5; i++)
        {
            Part part = new Part("Sede" + (i + 1));
            parts[i] = part;
            Console.WriteLine
                ("Sumando el disponible de sede '{0}' = {1}",
                 part.sku, part.onhand);
        }
    }

    public delegate void OutOfStockExceptionMethod(Part part);
    public void ProcessInventory(OutOfStockExceptionMethod exception)
    {
        Console.WriteLine("\nProcesando el inventario...");
        foreach (Part part in parts)
        {
            if (part.onhand < MIN_ONHAND)
            {
                Console.WriteLine
                    ("Disponible {0} ({1}) por debajo del límite {2}",
                     part.sku, part.onhand, MIN_ONHAND);

                exception(part);
            }
        }
    }
}

class CompositeDelegate2App
{
    public static void LogEvent(Part part)
    {
        Console.WriteLine("\tregistrando evento...");
    }

    public static void EmailPurchasingMgr(Part part)
    {
        Console.WriteLine("\tenviando un correo al responsable de compras...");
    }
}

```

```

public static void EmailStoreMgr(Part part)
{
    Console.WriteLine("\tenviando un correo al responsable del almacén...");
}

public static void Main()
{
    InventoryManager mgr = new InventoryManager();

    InventoryManager.OutOfStockExceptionMethod[] exceptionMethods
        = new InventoryManager.OutOfStockExceptionMethod[3];

    exceptionMethods[0] = new
        InventoryManager.OutOfStockExceptionMethod
            (LogEvent);
    exceptionMethods[1] = new
        InventoryManager.OutOfStockExceptionMethod
            (EmailPurchasingMgr);
    exceptionMethods[2] = new
        InventoryManager.OutOfStockExceptionMethod
            (EmailStoreMgr);

    int location = 1;

    InventoryManager.OutOfStockExceptionMethod compositeDelegate;

    if (location == 2)
    {
        compositeDelegate =
            exceptionMethods[0] + exceptionMethods[1];
    }
    else
    {
        compositeDelegate =
            exceptionMethods[0] + exceptionMethods[2];
    }

    mgr.ProcessInventory(compositeDelegate);
}
}

```

Ahora, la compilación y ejecución de esta aplicación devolverá diferentes resultados basándose en los distintos valores que asigne a la variable *location*.

CÓMO DEFINIR EVENTOS CON DELEGADOS

Casi todas las aplicaciones *Windows* tienen alguna clase de necesidad de procesamiento de eventos asíncrono. Algunos de estos eventos son genéricos, tales como el envío desde *Windows* de mensajes a la cola de mensajes de las aplicaciones cuando el usuario ha interactuado de alguna forma con la aplicación. Algunos problemas son más específicos de un dominio, tales como la impresión de una factura para un pedido que se ha actualizado.

Los eventos en C# siguen el patrón de diseño de publicación —suscripción en el cual una clase publica un evento que puede «lanzar» y cualquier número de clases se pueden suscribir a ese evento. Una vez que se lanza el evento, el entorno de ejecución se encarga de notificar a cada suscriptor que ha sucedido el evento.

El método invocado como resultado del lanzamiento de un evento se define mediante un delegado. Aun así, ha de tener en cuenta algunas reglas estrictas que afectan a un delegado que se utiliza de esta forma. Primero, el delegado tiene que definirse de forma que tome dos parámetros. En segundo lugar, los parámetros siempre representan dos objetos: el objeto que lanzó el evento (el que publica) y el objeto de información del evento. Adicionalmente, este segundo objeto debe derivarse de la clase *EventArgs* del Framework .NET.

Digamos que queremos controlar los cambios en los niveles de inventario. Podríamos crear una clase *InventoryManager* que se utilizase siempre para actualizar el inventario. La clase *InventoryManager* publicaría un evento que sería lanzado cada vez que el inventario se cambiara mediante acciones tales como la recepción de inventario, las ventas y las actualizaciones físicas del inventario. Luego, cualquier clase que necesitase mantenerse actualizada se suscribiría al evento. A continuación vemos cómo se codificaría esto en C# mediante la utilización de delegados y eventos:

```
using System;

class InventoryChangeEventArgs : EventArgs
{
    public InventoryChangeEventArgs(string sku, int change)
    {
        this.sku = sku;
        this.change = change;
    }

    string sku;
    public string Sku
    {
        get
        {
            return sku;
        }
    }

    int change;
    public int Change
    {
        get
        {
            return change;
        }
    }
}

class InventoryManager //Editor.
{
    public delegate void InventoryChangeEventHandler
        (object source, InventoryChangeEventArgs e);
```

```

public event InventoryChangeEventHandler OnInventoryChangeHandler;
public void UpdateInventory(string sku, int change)
{
    if (0 == change)
        return;    //No actualizar o cambio a nulo.

    //Código para actualizar la base de datos iría aquí.

    InventoryChangeEventArgs e = new
        InventoryChangeEventArgs(sku, change);

    if (OnInventoryChangeHandler != null)
        OnInventoryChangeHandler(this, e);
}

class InventoryWatcher //Suscriptor.
{
    public InventoryWatcher(InventoryManager inventoryManager)
    {
        this.inventoryManager = inventoryManager;
        inventoryManager.OnInventoryChangeHandler += new
            InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
    }
    void OnInventoryChange(object source, InventoryChangeEventArgs e)
    {
        int change = e.Change;
        Console.WriteLine("Sede '{0}' es {1} en {2} unidades",
            e.Sku,
            change > 0 ? "incremento" : "decremento",
            Math.Abs(e.Change));
    }
    InventoryManager inventoryManager;
}

class Events1App
{
    public static void Main()
    {
        InventoryManager inventoryManager =
            new InventoryManager();

        InventoryWatcher inventoryWatch =
            new InventoryWatcher(inventoryManager);

        inventoryManager.UpdateInventory("111 006 116", -2);
        inventoryManager.UpdateInventory("111 005 383", 5);
    }
}

```

Veamos los dos primeros miembros de la clase *InventoryManager*:

```

public delegate void InventoryChangeEventHandler
    (object source, InventoryChangeEventArgs e);
public event InventoryChangeEventHandler OnInventoryChangeHandler;

```

La primera línea de código es un delegado, que, como ya sabe, es una definición de una signature de método. Como mencionamos antes, todos los delegados que se van a utilizar como eventos tienen que definirse de forma que tomen dos parámetros: un objeto que publica (en este caso *source*) y un objeto de información de evento (un objeto derivado de *EventArgs*). La segunda línea utiliza la palabra reservada *event*, un tipo de miembro con el que se especifica el delegado y el método (o métodos) que se invocarán cuando se lance el evento.

El último método en la clase *InventoryManager* es el método *UpdateInventory*, que es invocado en el momento en que el inventario se cambia. Como puede ver, este método crea un objeto del tipo *InventoryChangeEventArgs*. Este objeto se pasa a todos los suscriptores y se utiliza para describir el evento que tuvo lugar.

Ahora vea las dos líneas de código siguientes:

```
if (OnInventoryChangeHandler != null)
    OnInventoryChangeHandler(this, e);
```

La instrucción condicional *if* comprueba si el evento tiene algún suscriptor asociado con el método *OnInventoryChangeHandler*. Si lo tiene —en otras palabras, *OnInventoryChangeHandler* no es *null*—, se lanza el evento. Esto es todo lo que hay desde el punto de vista de quien publica. Veamos ahora el código del suscriptor.

El suscriptor en este caso es la clase llamada *InventoryWatcher*. Todo lo que necesita hacer es llevar a cabo dos tareas sencillas. Primero, se añade a sí mismo como suscriptor mediante la instanciación de un nuevo delegado de tipo *InventoryManager.InventoryChangeEventHandler* y añade ese delegado al evento *InventoryManager.OnInventoryChangeHandler*. Preste especial atención a la sintaxis utilizada —se utiliza el operador de asignación compuesta *+=* para añadirse a sí mismo a la lista de suscriptores para así no borrar ningún suscriptor previo.

```
inventoryManager.OnInventoryChangeHandler
+= new InventoryManager.InventoryChangeEventHandler(OnInventoryChange);
```

El único parámetro que se tiene que proporcionar aquí es el nombre del método que se invocará en caso de que se lance el evento.

La única tarea adicional que el suscriptor tiene que hacer es implementar su manejador de evento. En este caso, el manejador de evento es *InventoryWatcher.OnInventoryChange*, que imprime un mensaje indicando el número de sección y el cambio en el inventario.

Finalmente, el código que ejecuta esta aplicación instancia las clases *InventoryManager* e *InventoryWatcher*, y cada vez que el método *InventoryManager.UpdateInventory* se invoca, se lanza un evento automáticamente que hace que se invoque el método *InventoryWatcher.OnInventoryChanged*.

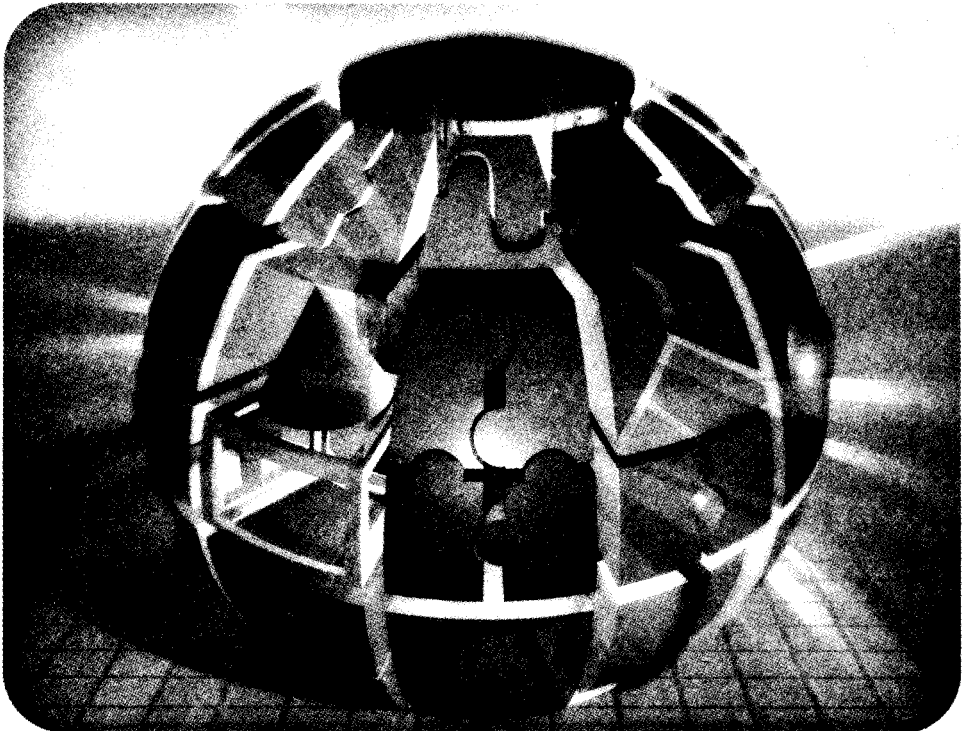
RESUMEN

Los delegados son objetos de memoria gestionada y de tipo seguro que tienen el mismo propósito que los punteros a funciones en C++. Los delegados difieren de las clases e

interfaces en que en vez de ser definidos en tiempo de compilación, se refieren a métodos sencillos que se definen en tiempo de ejecución. Los delegados se utilizan comúnmente para llevar a cabo procesamiento asíncrono y para insertar código de cliente en el código que ejecuta una clase. Los delegados se pueden utilizar para un gran número de propósitos generales, incluyendo su utilización como métodos de devolución de llamada, definir métodos estáticos y para definir eventos.

Parte IV

C# AVANZADO



Programación multihilo

Hablando en términos técnicos, los hilos de ejecución o subprocesos (*threads*) no son específicos de C#; por ello, la mayoría de los libros de C# evitan tratar este tema. Aunque hemos intentado ceñirnos lo más posible a C#, el tema general de la ejecución multihilo debería ser familiar para la mayoría de los programadores al aprender este nuevo lenguaje. Ciertamente, no podemos cubrir el rango completo de aspectos relativos a la ejecución multihilo en un capítulo, pero cubriremos los aspectos más elementales, e incluso algunos aspectos de nivel intermedio a avanzado sobre los procesos de cancelación, planificación y gestión del ciclo de vida de los hilos. También discutiremos la sincronización de hilos de ejecución con las clases *System.Monitor* y *System.Mutex* y con la palabra reservada *lock* en C#.

CONCEPTOS BÁSICOS SOBRE HILOS

Usar una aplicación con un único hilo de ejecución es algo parecido a comprar en un supermercado con un único cajero. Emplear un único cajero es más barato para el dueño del supermercado y el cajero puede manejar niveles de tráfico relativamente bajos, pero una vez que la tienda anuncia un día de ofertas y que las colas en el cajero comienzan a ser cada vez más largas, hay clientes que comienzan a enfadarse. Lo que nos encontramos es el escenario tradicional de un cuello de botella: muchos datos y un conducto demasiado pequeño. La solución pasa, por supuesto, por contratar más cajeros.

Y el principio es el mismo cuando hablamos de hilos de ejecución y aplicaciones. El uso del multihilo permite que una aplicación divida sus tareas de modo que puedan trabajar de manera simultánea pero independiente entre sí para hacer el uso más eficiente posible del tiempo, tanto del procesador como del usuario. Sin embargo, si el concepto de programación multihilo le es nuevo, tenga en cuenta que hay un pero: *la programación multihilo no es siempre la elección correcta para todas las aplicaciones, e incluso en determinadas ocasiones, puede ralentizar la aplicación.* De este modo, es de extrema importancia que, conjuntamente con el aprendizaje de la sintaxis para hacer uso de la programación multihilo, deberemos entender en qué contexto usarla de manera efectiva.

Para lograrlo, hemos incluido una sección al final de este capítulo («Guía para el uso de hilos») para ayudarle a tomar una decisión sobre cuándo es la opción correcta el crear múltiples hilos de ejecución en sus aplicaciones. Por ahora, vamos a ver cómo están implementados los hilos de ejecución en el .NET Framework.

Hilos y multitarea

Un hilo de ejecución es una unidad de procesamiento, y la multitarea consiste en la ejecución simultánea de múltiples hilos. La multitarea posee dos ramas principales: multitarea cooperativa y multitarea preferente. Algunas versiones muy antiguas de Windows permitían multitarea cooperativa, lo que significaba que cada hilo de ejecución era responsable de renunciar al control del procesador de tal modo que éste pudiera procesar otros hilos. Para aquellos de nosotros con experiencia en otros sistemas operativos —en mi caso, IBM System/38 (CPF) y OS/2—, seguro que todos tenemos nuestra propia experiencia del día en que colgamos el ordenador porque no colocamos una llamada a *PeekMessage* en nuestro código para permitir que la CPU atendiera a otros hilos de ejecución en el sistema. «¡Que tengo que hacer qué!», era nuestra respuesta típica.

*Sin embargo, Microsoft Windows NT —y luego Windows 95, Windows 98 y Windows 2000— permite el mismo tipo de multitarea preferente que OS/2. Con la multitarea preferente, el procesador es responsable de dar a cada hilo de ejecución una cierta cantidad de tiempo en la que ejecutarse —una *fracción o rodaja de tiempo* (*timeslice*). El procesador procede entonces a seleccionar alternativamente los diferentes hilos, dándole a cada uno su fracción de tiempo, y el programador no tiene que preocuparse de cuándo ceder el control para que el resto de los hilos de ejecución puedan ejecutarse. Dado que .NET sólo podrá ejecutarse en sistemas operativos con multitarea preferente, nos centramos en estos sistemas.

Por cierto, incluso con multitarea preferente, si nuestro programa se está ejecutando en una máquina con un único procesador, realmente no tenemos múltiples hilos de ejecución ejecutándose al mismo tiempo. Comoquiera que el procesador está alternando entre procesos en intervalos que duran milisegundos, de alguna manera «sentimos» que lo está haciendo. Si queremos ejecutar múltiples hilos de ejecución concurrentemente de verdad, necesitaremos desarrollar y ejecutar nuestro código en una máquina con múltiples procesadores.

Alternancia de contexto (*Context switching*)

La *alternancia de contexto* forma parte de la utilización de hilos de ejecución y es un concepto complicado de asimilar para algunos, así que permitidnos hacer un pequeño resumen aquí en la sección «Conceptos básicos sobre hilos».

El procesador utiliza un temporizador hardware para determinar cuándo ha terminado una fracción de tiempo para un hilo de ejecución dado. Cuando el temporizador lanza la interrupción, el procesador salva en la pila el estado de los registros del hilo de ejecución en curso. Lo siguiente que hace el procesador es mover esos datos a una estructura de

datos denominada estructura *CONTEXT*. Cuando el procesador quiere retornar a un hilo de ejecución previamente ejecutado, deshace la operación anterior y recupera los valores existentes en la estructura *CONTEXT*, depositándolos de nuevo en los registros. El conjunto de todas estas operaciones se denomina alternancia de contexto.

UNA APLICACIÓN MULTIHILLO EN C#

Antes de examinar algunas de las diferentes maneras de utilizar hilos de ejecución en C#, veamos lo sencillo que es crear un hilo de ejecución secundario en C#. Después de discutir el ejemplo, veremos en más detalle el espacio de nombres *System.Threading* y, específicamente, la clase *Thread*. En este ejemplo, estamos creando un segundo hilo de ejecución desde el método *Main*. El método asociado con el segundo hilo de ejecución produce un mensaje que indica que el segundo hilo de ejecución se ha invocado.

```
using System;
using System.Threading;

class SimpleThreadApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("Hilo de ejecución secundario iniciado");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Principal -Creando hilo de ejecución secundario");

        Thread t = new Thread(worker);
        t.Start();

        Console.WriteLine
            ("Principal -Hilo de ejecución secundario invocado");
    }
}
```

Si compila y ejecuta esta aplicación, observará que el mensaje que proviene del método *Main* imprime primero el mensaje del hilo de ejecución secundario, demostrando de este modo que el hilo de ejecución está efectivamente funcionando de manera asíncrona. Vamos a diseccionar lo que está ocurriendo aquí.

El primer detalle a mencionar es la inclusión de la instrucción *using* para el espacio de nombres *System.Threading*. Entraremos en más detalle en ese espacio de nombres en breve. Por ahora es suficiente el entender que dicho espacio de nombres contiene las distintas clases necesarias para usar los hilos de ejecución de manera efectiva en el entorno .NET. Observemos ahora la primera línea del método *Main*:

```
ThreadStart WorkerThreadMethod = new ThreadStart(WorkerThreadMethod);
```

Cada vez que vea esta estructura, puede estar seguro de que *x* es un delegado o —como aprendimos en el Capítulo 14, «Delegados y manejadores de eventos»— una definición de un nombre de método:

```
x NombreVariable = new x(NombreMétodo);
```

Así sabemos que *ThreadStart* es un delegado. Pero no un delegado cualquiera. Específicamente, es el delegado que se debe utilizar cuando creamos un nuevo hilo de ejecución, y se usa para especificar el método que queremos invocar para inicializar el hilo de ejecución. De ahí en adelante, instanciamos un objeto *Thread*, cuyo constructor lleva como único argumento un delegado *ThreadStart*, de esta manera:

```
Thread t = new Thread(worker);
```

Después de esto, llamamos al método *Start* perteneciente al objeto *Thread*, lo que resulta en una llamada a *WorkerThreadMethod*.

¡Y ya está! Tres líneas de código para preparar y ejecutar el hilo de ejecución y el método del hilo de ejecución en sí, y ¡ya estamos en marcha! Veamos ahora más en detalle el espacio de nombres *System.Threading* y las clases que permiten que todo esto suceda.

CÓMO TRABAJAR CON HILOS

Toda la creación y la gestión de hilos de ejecución se logra a través del uso de la clase *System.Threading.Thread*, así que empezaremos por ahí.

AppDomain

En .NET, los hilos de ejecución se ejecutan en algo llamado *AppDomain*. En ocasiones habrá escuchado que *AppDomain* es análogo a un proceso Win32 en el hecho de que ofrece muchos de sus mismos beneficios, incluyendo tolerancia a fallos y la capacidad de ser arrancado y detenido de manera independiente. Es una buena comparación, pero la comparación se rompe en el momento en que hablamos de hilos. En Win32, un hilo de ejecución está confinado a un único proceso, como ya vimos cuando mencionamos antes la alternancia de contextos. Un hilo de ejecución en un proceso no puede invocar a un método en un hilo de ejecución que pertenezca a otro proceso. En .NET, sin embargo, los hilos de ejecución pueden romper los límites de *AppDomain*, y un método en un hilo de ejecución puede invocar a un método existente en otro *AppDomain*. Por lo tanto, hay una definición mejor de *AppDomain*: un proceso lógico dentro de un proceso físico.

La clase *Thread*

La mayor parte de toda la operativa con hilos de ejecución la realizaremos usando la clase *Thread*. Esta sección trata sobre cómo usar la clase *Thread* para ejecutar tareas básicas que utilicen hilos.

Cómo crear hilos de ejecución y objetos *Thread*

Puede instanciar un objeto *Thread* de dos maneras diferentes. Ya hemos visto una: crear un nuevo hilo de ejecución, y en ese proceso, obtener un objeto *Thread* con el que manipularlo. La otra manera de obtener un objeto *Thread* es invocando el método estático *Thread.CurrentThread* para el hilo de ejecución que se está ejecutando en ese momento.

Cómo gestionar el ciclo de vida de un hilo

Hay muchas tareas diferentes que pueden ser necesarias para gestionar la actividad o la vida de un hilo. Podemos gestionar todas esas tareas usando los diferentes métodos de *Thread*. Por ejemplo, es bastante común el detener un hilo de ejecución durante una cantidad determinada de tiempo. Para hacerlo, podemos invocar el método *Thread.Sleep*. Este método tiene un único argumento que representa la cantidad de tiempo, en milisegundos, que queremos detener el hilo de ejecución. Observe que el método *Thread.Sleep* es un método estático y no se puede invocar instanciando un objeto *Thread*. Hay una razón excelente para esto: no se nos permite invocar *Thread.Sleep* en ningún otro hilo de ejecución, con excepción del que estamos ejecutando en este mismo momento. El método estático *Thread.Sleep* llama al método estático *CurrentThread*, que procede a detener ese hilo de ejecución durante la cantidad de tiempo especificada. Aquí tenemos un ejemplo:

```
using System;
using System.Threading;

class ThreadSleepApp
{
    public static void WorkerThreadMethod()
    {
        Console.WriteLine("Hilo de ejecución secundario iniciado");

        int sleepTime = 5000;

        Console.WriteLine("\tDurmiendo durante {0} segundos", sleepTime
                           / 1000);
        Thread.Sleep(sleepTime); //Duerme durante cinco segundos.
        Console.WriteLine("\tActivándose");
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);
```

```

        Console.WriteLine("Principal -Creando hilo de ejecución secundario");

        Thread t = new Thread(worker);
        t.Start();

        Console.WriteLine
        ("Principal -Solicitada la inicialización del hilo de ejecución
        secundario");
    }
}

```

Hay dos maneras más de invocar el método *Thread.Sleep*. Primero, invocando *Thread.Sleep* con el valor 0, podemos hacer que el hilo de ejecución actual renuncie a la parte no utilizada de la fracción de tiempo que tiene asignada. Pasar un valor *Timeout.Infinite* hace que el hilo de ejecución se detenga de manera indefinida hasta que este estado sea interrumpido por otro hilo de ejecución invocando al método *Thread.Interrupt* del hilo de ejecución detenido.

La segunda manera de suspender temporalmente la ejecución de un hilo de ejecución es utilizar el método *Thread.Suspend*. Hay diferencias significativas entre las dos técnicas. Primero, el método *Thread.Suspend* puede ser invocado en el hilo de ejecución en ejecución actualmente o desde otro hilo de ejecución. Segundo, una vez que un hilo de ejecución está suspendido de este modo, sólo otro hilo de ejecución puede reactivarlo, utilizando el método *Thread.Resume*. Observe que una vez que un hilo de ejecución suspende a otro hilo de ejecución, el primer hilo de ejecución no está bloqueado. La llamada devuelve el control al método invocador inmediatamente. También, e independientemente de cuántas veces sea invocado el método *Thread.Suspend* para un hilo de ejecución determinado, una única llamada a *Thread.Resume* hará que el hilo de ejecución reanude su ejecución.

Eliminación de hilos de ejecución

Si surge la necesidad de destruir un hilo, podemos conseguirlo con una llamada al método *Thread.Abort*. El entorno de ejecución fuerza la eliminación del hilo de ejecución al lanzar una excepción *ThreadAbortException*. Incluso si el método intenta capturar la *ThreadAbortException*, el entorno de ejecución no le dejará. Sin embargo, el entorno de ejecución sí ejecutará el código en el bloque *finally* del hilo de ejecución cancelado, si está presente. El siguiente código ilustra lo que pretendemos mostrar. El método *Main* hace una pausa de dos segundos para asegurar que el entorno de ejecución ha tenido tiempo de iniciar el hilo de ejecución secundario. Una vez iniciado, el hilo de ejecución secundario comienza a contar hasta diez, parando un segundo entre cada número. Cuando el método *Main* reanuda su ejecución después de su parada de dos segundos, cancela el hilo de ejecución secundario. Observe que el bloque *finally* se ejecuta después de la cancelación del hilo de ejecución.

```

using System;
using System.Threading;

```

```

class ThreadAbortApp
{
    public static void WorkerThreadMethod()
    {
        try
        {
            Console.WriteLine("Hilo de ejecución secundario iniciado");

            Console.WriteLine
                ("Hilo de ejecución secundario -Contando lentamente
                 hasta 10");
            for (int i = 0; i < 10; i++)
            {
                Thread.Sleep(500);
                Console.Write("{0}...", i);
            }

            Console.WriteLine("Hilo de ejecución secundario terminado");
        }
        catch(ThreadAbortException e)
        {
        }
        finally
        {
            Console.WriteLine
                ("Hilo de ejecución secundario -No puedo
                 capturar la excepción pero puedo limpiar");
        }
    }

    public static void Main()
    {
        ThreadStart worker = new ThreadStart(WorkerThreadMethod);

        Console.WriteLine("Principal -Creando hilo de ejecución
                           secundario");

        Thread t = new Thread(worker);
        t.Start();

        //Darle al hilo de ejecución secundario tiempo para empezar.
        Console.WriteLine("Principal -Inactiva durante dos segundos");
        Thread.Sleep(2000);

        Console.WriteLine("\nPrincipal -Abortando el hilo de ejecución
                           secundario");
        t.Abort();
    }
}

```

Cuando compile y ejecute esta aplicación, obtenemos el siguiente resultado:

```

Principal -Creando hilo de ejecución secundario
Principal -Inactiva durante dos segundos
Hilo de ejecución secundario iniciado
Hilo de ejecución secundario -Contando lentamente hasta 10

```

```
0...1...2...3...
```

```
Principal -Abortando el hilo de ejecución secundario
```

```
Hilo de ejecución secundario -No puedo capturar la excepción pero puedo limpiar
```

Debe observar también que cuando se invoca el método *Thread.Abort*, el hilo de ejecución no detendrá su ejecución inmediatamente. El entorno de ejecución espera hasta que el hilo haya alcanzado lo que la documentación llama un «punto seguro». Por lo tanto, si su código depende de algo que debe ocurrir después de que se anule el hilo de ejecución y debe estar seguro de que se ha detenido, puede usar el método *Thread.Join*. Esta es una llamada síncrona, lo que significa que no devolverá el control hasta que el hilo de ejecución se haya detenido. Por último, aunque tenga un objeto válido *Thread*, no puede hacer nada útil con él en términos de código a ejecutar.

Planificación de los hilos

Cuando el procesador alterna entre hilos de ejecución, una vez que la fracción de tiempo asignada a un hilo de ejecución en concreto ha terminado, el proceso de determinar cuál es el siguiente hilo de ejecución a ejecutar es cualquier cosa, salvo arbitrario. Cada hilo de ejecución tiene asociado un nivel de prioridad que informa al procesador de cómo se debería planificar con relación al resto de los hilos de ejecución en el sistema. Este nivel de prioridad es por defecto *Normal* —más sobre esto en breve— para hilos de ejecución que son creados dentro del entorno de ejecución. Los hilos que se crean fuera del entorno de ejecución mantienen su prioridad original. Puede usar la propiedad *Thread.Priority* para ver y fijar este valor. El método de asignación de la propiedad *Thread.Priority* toma un valor del tipo *Thread.ThreadPriority*, que es un *enum* que posee estos posibles valores: *Highest* (Máxima), *AboveNormal* (Por encima de lo normal), *Normal*, *BelowNormal* (Por debajo de lo normal) y *Lowest* (Mínima).

Para ilustrar cómo pueden afectar las prioridades incluso al más simple de los códigos, eche un vistazo al siguiente ejemplo, en el que un hilo de ejecución secundaria cuenta de 1 a 10 y otro cuenta de 11 a 20. Observe el bucle anidado dentro de cada método *WorkerThread*. Cada bucle representa trabajo que el hilo de ejecución realizaría dentro de una aplicación real. Comoquiera que estos métodos realmente no hacen nada, si no tuviéramos esos bucles, resultaría que ¡cada hilo de ejecución terminaría su trabajo en su primera fracción de tiempo!

```
using System;
using System.Threading;

class ThreadSchedule1App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Hilo de ejecución secundario iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario -Contando lentamente de 1 a 10");
```

```

    for (int i = 1; i < 11; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            //Código para imitar trabajo real.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Hilo de ejecución secundario finalizado");
}

public static void WorkerThreadMethod2()
{
    Console.WriteLine("Hilo de ejecución secundario iniciado");

    Console.WriteLine
        ("Hilo de ejecución secundario -Contando lentamente
         de 11 a 20");
    for (int i = 11; i < 20; i++)
    {
        for (int j = 0; j < 100; j++)
        {
            Console.Write(".");
            //Código para imitar trabajo real.
            int a;
            a = 15;
        }
        Console.Write("{0}", i);
    }

    Console.WriteLine("Hilo de ejecución secundario finalizado");
}

public static void Main()
{
    ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
    ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

    Console.WriteLine("Principal -Creando hilos de ejecución
                      secundarios");

    Thread t1 = new Thread(worker1);
    Thread t2 = new Thread(worker2);

    t1.Start();
    t2.Start();
}
}

```

Al ejecutar esta aplicación obtenemos el siguiente resultado. Observe que, en aras a ser breves, hemos truncado la salida —la mayor parte de los puntos están borrados.

```
Principal -Creando hilos de ejecución secundarios
Hilo de ejecución secundario iniciado
Hilo de ejecución secundario iniciado
Hilo de ejecución secundario -Contando lentamente de 1 a 10
Hilo de ejecución secundario -Contando lentamente de 11 a 20
.....1.....11.....2.....12.....3.....13
```

Como puede ver, ambos hilos de ejecución obtienen el mismo tiempo del procesador. Ahora vamos a alterar la propiedad *Priority* para cada hilo de ejecución como en el código siguiente —le hemos dado al primer hilo de ejecución la prioridad más alta permitida y al segundo hilo de ejecución la más baja— y veremos un resultado completamente diferente.

```
using System;
using System.Threading;

class ThreadSchedule2App
{
    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Hilo de ejecución secundario iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario -Contando lentamente de 1 a 10");
        for (int i = 1; i < 11; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                //Código para imitar trabajo real.
                int a;
                a = 15;
            }
            Console.Write("{0}", i);
        }

        Console.WriteLine("Hilo de ejecución secundario finalizado");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Hilo de ejecución secundario iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario -Contando lentamente de 11 a 20");
        for (int i = 11; i < 20; i++)
        {
            for (int j = 0; j < 100; j++)
            {
                Console.Write(".");
                //Código para imitar trabajo real.
                int a;
                a = 15;
            }
        }
    }
}
```

```

        Console.Write("{0}", i);
    }

    Console.WriteLine("Hilo de ejecución secundario finalizado");
}

public static void Main()
{
    ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
    ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

    Console.WriteLine("Principal -Creando hilos de ejecución
                      secundarios");

    Thread t1 = new Thread(worker1);
    Thread t2 = new Thread(worker2);

    t1.Priority = ThreadPriority.Highest;
    t2.Priority = ThreadPriority.Lowest;

    t1.Start();
    t2.Start();
}
}

```

Los resultados (abreviados) deberían ser como los mostrados a continuación. Observe que el segundo hilo de ejecución tiene una prioridad tan baja que no recibe ningún ciclo de reloj hasta que el primer hilo de ejecución ha terminado su labor.

```

Principal -Creando hilos de ejecución secundarios
Hilo de ejecución secundario iniciado
Hilo de ejecución secundario iniciado
Hilo de ejecución secundario -Contando lentamente de 1 a 10
Hilo de ejecución secundario -Contando lentamente de 11 a 20
.....1.....2.....3.....4.....5.....6.....7.....8.....9.....10.....
11.....12.....13.....14.....15.....16.....17.....18.....19.....20

```

Recuerde que cuando le comunicamos al procesador la prioridad que queremos que tenga un hilo de ejecución determinado, es el sistema operativo el que finalmente utiliza ese valor como parte del algoritmo planificador que aplica al procesador. En .NET, este algoritmo está basado en el nivel de prioridad que acabamos de fijar (con la propiedad *Thread.Priority*), así como en los valores *prioridad de clase* (*priority class*) y *aceleración dinámica* (*dynamic boost*) del proceso. Todos estos valores se usan para crear un valor numérico (0-31 en un procesador Intel) que representa la prioridad del hilo de ejecución. El hilo de ejecución que posea el valor más alto es el hilo de ejecución con la prioridad más alta.

Un último apunte sobre planificación de hilos: utilícelo con precaución. Digamos que tenemos una aplicación con interfaz gráfica de usuario y un par de hilos de ejecución secundarios realizando algún tipo de trabajo asíncrono. Si fijamos las prioridades de esos hilos de ejecución secundarios demasiado altas, la interfaz de usuario puede volverse inmanejablemente lenta, porque el hilo de ejecución principal en el que reside la interfaz gráfica de usuario que se está ejecutando recibe menos ciclos de CPU. A menos que

tengamos una razón poderosa para planificar un hilo de ejecución específico con prioridad alta, es mejor mantener la prioridad por defecto de los hilos de ejecución en *Normal*.

NOTA Cuando tenemos una situación en la que varios hilos de ejecución se están ejecutando con la misma prioridad, todos ellos obtienen la misma cantidad de tiempo de procesador. Esta situación se denomina planificación por algoritmo de asignación por cuantos de tiempo (*round robin*).

SEGURIDAD Y SINCRONIZACIÓN EN LOS HILOS DE EJECUCIÓN

Cuando programamos para un entorno monohilo, es común el escribir métodos de tal modo que en determinados puntos del código el objeto está temporalmente en un estado no válido. Obviamente, si solamente accede al objeto en cuestión un hilo de ejecución, estamos garantizando que cada método se completará antes de que se invoque cualquier otro método —implicando que el objeto está siempre en un estado válido para cualquiera de los clientes del objeto. Sin embargo, cuando aparecen los hilos de ejecución múltiples, se nos pueden presentar fácilmente situaciones en las que el procesador salta a otro hilo de ejecución mientras que nuestro objeto está en un estado no válido. Si ese hilo de ejecución intenta a su vez utilizar el mismo objeto, los resultados son bastante impredecibles. Por lo tanto, el término «seguridad en los hilos» significa que los miembros de un objeto siempre mantienen un valor válido cuando son usados de manera concurrente por hilos de ejecución múltiples.

Así que ¿cómo prevenimos estos estados impredecibles? Realmente, como suele ser habitual en la programación, hay diversas maneras de prevenir este bien conocido problema. En esta sección mencionaremos el método más común: sincronización. Usando la sincronización, especificaremos *secciones críticas* del código en las que sólo puede acceder un hilo de ejecución cada vez, garantizando por lo tanto que en cualquier estado temporal no válido nuestro objeto no será visto por los clientes. Veremos varios métodos de definir secciones críticas, incluyendo las clases *.NET Monitor* y *Mutex*, así como la instrucción *lock* de C#.

Cómo proteger el código usando la clase *Monitor*

La clase *System.Monitor* permite acceder de manera secuencial a bloques de código por medio de bloqueos y señales. Por ejemplo, tenemos un método que actualiza una base de datos y que no puede ser ejecutado por dos o más hilos de ejecución al mismo tiempo. Si el trabajo que realiza este método exige mucho tiempo y tenemos múltiples hilos, de los cuales cualquiera puede invocar este método, podemos tener un serio problema entre manos. Aquí es donde interviene la clase *Monitor*. Echemos un vistazo al siguiente ejemplo de sincronización. Aquí tenemos dos hilos, que invocarán ambos al método *Database.SaveData*.

```

using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Console.WriteLine("Database.SaveData -Iniciado");

        Console.WriteLine("Database.SaveData -Ejecutándose");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData -Finalizado");
    }
}

class ThreadMonitor1App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Hilo de ejecución secundario #1 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #1 -Invocando Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Hilo de ejecución secundario #1 -Retornado
                           desde Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Hilo de ejecución secundario #2 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #2 -Invocando Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Hilo de ejecución secundario #2 -Retornado
                           desde Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Principal -Creando hilos de ejecución
                           secundarios");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);
    }
}

```

```

        t1.Start();
        t2.Start();
    }
}

```

Quando compilamos y ejecutemos esta aplicación, veremos que el resultado de la salida será una mezcla de oes y equis, mostrando que el método *Database.SaveData* se está ejecutando de manera concurrente por ambos hilos de ejecución. (Observe que de nuevo hemos abreviado la salida).

Principal -Creando hilos de ejecución secundarios

Hilo de ejecución secundario #1 -Iniciado

Hilo de ejecución secundario #2 -Iniciado

Hilo de ejecución secundario #1 -Invocando Database.SaveData

Hilo de ejecución secundario #2 -Invocando Database.SaveData

Database.SaveData -Iniciado

Database.SaveData -Iniciado

Database.SaveData -Ejecutándose

Database.SaveData -Ejecutándose

XX

Database.SaveData -Finalizado

Database.SaveData -Finalizado

Hilo de ejecución secundario #1 -Retornado desde Output

Hilo de ejecución secundario #2 -Retornado desde Output

Obviamente, si el método `Database.SaveData` necesitaba terminar de actualizar múltiples tablas antes de ser invocado por otro hilo de ejecución, tendríamos un problema serio.

Para incorporar la clase *Monitor* en este ejemplo, usamos dos de sus métodos estáticos. El primer método se denomina *Enter*. Cuando se ejecuta, este método intenta obtener un bloqueo exclusivo del objeto. Si otro hilo de ejecución lo tiene ya bloqueado, el método será inaccesible hasta que dicho bloqueo haya sido liberado. Observe que no hay una operación de empaquetado (*boxing*) implícita, así que sólo podemos proporcionar un tipo referencia a este método. El método *Monitor.Exit* se invoca entonces para liberar el bloqueo. Aquí podemos ver el ejemplo reescrito para forzar la serialización del acceso al método *Database.SaveData*:

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        Monitor.Enter(this);

        Console.WriteLine("Database.SaveData -Iniciado");

        Console.WriteLine("Database.SaveData -Ejecutándose");
    }
}
```

```

        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine(text);
        }

        Console.WriteLine("\nDatabase.SaveData -Finalizado");

        Monitor.Exit(this);
    }
}

class ThreadMonitor2App
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Hilo de ejecución secundario #1 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #1 -Invocando Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Hilo de ejecución secundario #1 -Retornado
                           desde Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Hilo de ejecución secundario #2 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #2 -Invocando Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Hilo de ejecución secundario #2 -Retornado
                           desde Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Principal -Creando hilos de ejecución
                           secundarios");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}

```

Observe en este resultado de salida que aunque el segundo hilo de ejecución invoca el método *Database.SaveData*, el método *Monitor.Enter* previene que se acceda a él hasta que el primer hilo de ejecución haya liberado su bloqueo:

```
Principal -Creando hilos de ejecución secundarios

Hilo de ejecución secundario #1 -Iniciado
Hilo de ejecución secundario #2 -Iniciado

Hilo de ejecución secundario #1 -Invocando Database.SaveData
Hilo de ejecución secundario #2 -Invocando Database.SaveData

Database.SaveData -Iniciado
Database.SaveData -Ejecutándose
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Database.SaveData -Finalizado

Database.SaveData -Iniciado
Hilo de ejecución secundario #1 -Retornado desde Output
Database.SaveData -Ejecutándose
oooooooooooooooooooooooooooooooooooooooooooooooooooo
Database.SaveData -Finalizado

Hilo de ejecución secundario #2 -Retornado desde Output
```

Utilización de bloqueos de monitor con la instrucción *lock* de C#

Aunque la instrucción *lock* de C# no permite todas las funcionalidades proporcionadas por la clase *Monitor*, permite obtener y liberar un bloqueo de monitor. Para utilizar la instrucción *lock*, simplemente especificaremos la instrucción *lock* con el código que va a ser objeto de acceso en serie entre llaves. Las llaves indican el punto inicial y final del código que está siendo protegido, por lo que no es necesaria una instrucción *unlock*. El siguiente código producirá el mismo tipo de salida sincronizada que proporcionaban los ejemplos anteriores:

```
using System;
using System.Threading;

class Database
{
    public void SaveData(string text)
    {
        lock(this)
        {
            Console.WriteLine("Database.SaveData -Iniciado");

            Console.WriteLine("Database.SaveData -Ejecutándose");
            for (int i = 0; i < 100; i++)
            {
```

```

        Console.Write(text);
    }
    Console.WriteLine("\nDatabase.SaveData -Finalizado");
}
}

class ThreadLockApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()
    {
        Console.WriteLine("Hilo de ejecución secundario #1 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #1 -Invocando Database.SaveData");
        db.SaveData("x");

        Console.WriteLine("Hilo de ejecución secundario #1 -Retornado
                           desde Output");
    }

    public static void WorkerThreadMethod2()
    {
        Console.WriteLine("Hilo de ejecución secundario #2 -Iniciado");

        Console.WriteLine
            ("Hilo de ejecución secundario #2 -Invocando Database.SaveData");
        db.SaveData("o");

        Console.WriteLine("Hilo de ejecución secundario #2 -Retornado
                           desde Output");
    }

    public static void Main()
    {
        ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
        ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

        Console.WriteLine("Principal -Creando hilos de ejecución
                           secundarios");

        Thread t1 = new Thread(worker1);
        Thread t2 = new Thread(worker2);

        t1.Start();
        t2.Start();
    }
}

```

Cómo sincronizar código usando la clase *Mutex*

La clase *Mutex* —definida en el espacio de nombres *System.Threading*— es una representación de la primitiva del sistema Win32 del mismo nombre. Podemos usar una ex-

clusión mutua (*mutex*) para serializar el acceso al código exactamente igual que podemos usar un monitor, pero las exclusiones mutuas son mucho más lentas, debido a que proporcionan mayor flexibilidad. El término *mutex* proviene de *mutually exclusive* (exclusión mutua), y al igual que solamente un hilo de ejecución puede obtener a la vez un bloqueo de monitor para un objeto dado, sólo un hilo de ejecución puede obtener cada vez una exclusión mutua dada.

Podemos crear una exclusión mutua en C# con los tres constructores siguientes:

```

Mutex()
Mutex(bool PropietarioInicial)
Mutex(bool PropietarioInicial, string NombreDeMutex)

```

El primer constructor crea una exclusión mutua sin nombre concreto y hace al hilo de ejecución en curso propietario de la exclusión mutua. Por lo tanto, la exclusión mutua está bloqueada por el hilo de ejecución en curso. El segundo constructor utiliza tan sólo un indicador booleano que especifica si el hilo de ejecución que crea la exclusión mutua quiere poseerla (bloquearla). Y el tercer constructor permite especificar si el hilo de ejecución en curso es dueño de la exclusión mutua y especificar el nombre de la exclusión mutua. Incorporemos una exclusión mutua para acceder secuencialmente al método *Database.SaveData*:

```

using System;
using System.Threading;

class Database
{
    Mutex mutex = new Mutex(false);

    public void SaveData(string text)
    {
        mutex.WaitOne();

        Console.WriteLine("Database.SaveData -Iniciado");

        Console.WriteLine("Database.SaveData -Ejecutándose");
        for (int i = 0; i < 100; i++)
        {
            Console.Write(text);
        }

        Console.WriteLine("\nDatabase.SaveData -Finalizado");

        mutex.Close();
    }
}

class ThreadMutexApp
{
    public static Database db = new Database();

    public static void WorkerThreadMethod1()

```

```

{
    Console.WriteLine("Hilo de ejecución secundario #1 -Iniciado");

    Console.WriteLine
        ("Hilo de ejecución secundario #1 -Invocando Database.SaveData");
    db.SaveData("x");

    Console.WriteLine("Hilo de ejecución secundario #1 -Retornado
        desde Output");
}

public static void WorkerThreadMethod2()
{
    Console.WriteLine("Hilo de ejecución secundario #2 -Iniciado");

    Console.WriteLine
        ("Hilo de ejecución secundario #2 -Invocando Database.SaveData");
    db.SaveData("o");

    Console.WriteLine("Hilo de ejecución secundario #2 -Retornado
        desde Output");
}

public static void Main()
{
    ThreadStart worker1 = new ThreadStart(WorkerThreadMethod1);
    ThreadStart worker2 = new ThreadStart(WorkerThreadMethod2);

    Console.WriteLine("Principal -Creando hilos de ejecución
        secundarios");

    Thread t1 = new Thread(worker1);
    Thread t2 = new Thread(worker2);

    t1.Start();
    t2.Start();
}
}

```

Ahora, la clase *Database* define un campo *Mutex*. No queremos que el hilo de ejecución tome el control de la exclusión mutua inmediatamente, porque no tendríamos manera de acceder al método *SaveData*. La primera línea del método *SaveData* nos muestra cómo debemos intentar adquirir la exclusión mutua —con el método *Mutex.WaitOne*. Al final del método tenemos una llamada al método *Close*, que libera la exclusión mutua.

El método *WaitOne* está asimismo sobrecargado para proporcionar más flexibilidad en términos de permitirnos definir cuánto tiempo esperará a la exclusión mutua el hilo de ejecución para quedar disponible. Estas son las sobrecargas:

```

WaitOne()
WaitOne(TimeSpan tiempo, bool salirDelContexto)
WaitOne(int milisegundos, bool salirDelContexto)

```

La diferencia básica entre estas sobrecargas es que la primera versión —usada en el ejemplo— esperará indefinidamente, mientras que la segunda y tercera versión *esperarán* durante la cantidad de tiempo que les especifiquemos, expresada con un valor *TimeSpan* o con un valor *int*.

Seguridad en los hilos de ejecución y las clases .NET

Una pregunta que nos encontramos con cierta asiduidad en los grupos de noticias y las listas de correo es si las clases *System.** de .NET son seguras con relación a los hilos de ejecución. La respuesta es «No, y además no deberían serlo». Podemos perjudicar seriamente el rendimiento de un sistema si todas las clases obligan a acceder a sus funcionalidades de forma secuencial. Por ejemplo, imaginemos que intentamos hacer uso de una de las clases de recolección que tuviera un bloqueo de monitor cada vez que invocáramos a su método *Add*. Digamos ahora que instanciamos un objeto colección y le añadimos mil objetos. El rendimiento será lamentable —hasta el punto de hacer el sistema inútil.

GUÍAS PARA USAR HILOS DE EJECUCIÓN

¿Cuándo tenemos que utilizar hilos, y cuándo es mejor, si es que hay alguna situación así, donde haya que evitarlos como a la peste? En esta sección describiremos tanto escenarios en los que los hilos de ejecución pueden ser de extrema utilidad para nuestras aplicaciones como algunas situaciones en las que sería recomendable evitar usar múltiples hilos de ejecución.

Cuándo usar hilos

Deberíamos usar hilos de ejecución cuando estamos buscando aumentar la concurrencia de nuestros programas, simplificar nuestro diseño o mejorar la utilización de nuestro tiempo de CPU, como describimos en las siguientes secciones.

Incrementar la concurrencia

Muy a menudo, las aplicaciones necesitan terminar más de una tarea a la vez. Por ejemplo, en una ocasión desarrollamos un sistema de recuperación de documentos para bancos que accedían a datos que estaban en discos ópticos que a su vez estaban en almacenes de discos ópticos. Imaginemos las cantidades masivas de datos de las que estamos hablando, imagínese un almacén con un único dispositivo de lectura y 50 servidores de discos proporcionando gigabytes de datos. En ocasiones tardabas unos 5 o 10 segundos en localizar un disco y encontrar el documento solicitado. No hace falta señalar que no sería exactamente la definición de productividad si nuestra aplicación impidiera que el

usuario siguiera operando mientras se hacía todo eso. Por lo tanto, para que se ocupara de las solicitudes de usuario, lanzamos otro hilo de ejecución que hiciera todo el trabajo físico de recuperar los datos, permitiendo que el usuario siguiera trabajando. Este hilo de ejecución avisaría al hilo de ejecución principal cuando estuviera finalmente cargado el documento. Esto es un excelente ejemplo de cómo podemos mantener actividades independientes —cargar un documento y manejar la interfaz de usuario— que pueden ser manejadas con dos hilos de ejecución diferentes.

Simplificar el diseño

Una manera clásica de simplificar el diseño de sistemas complejos es usar colas y procesamiento asíncrono. Utilizando dicho diseño, tendríamos colas preparadas para manejar los diferentes eventos que se den en nuestro sistema. En lugar de tener métodos que invocamos directamente, los objetos se crean y colocan en colas donde se manejarán. En el otro extremo de dichas colas tenemos programas servidores con múltiples hilos de ejecución que están preparados para «escuchar» mensajes provenientes de dichas colas. La ventaja de este tipo de diseño simplificado es que proporciona un entorno fiable, robusto y fácilmente escalable para el desarrollo de sistemas.

Mejor utilización del tiempo de CPU

Muchas veces, nuestra aplicación no está haciendo trabajo real mientras que disfruta de su fracción de tiempo. En el ejemplo del sistema de recuperación de documentos que mencionamos antes, un hilo de ejecución estaba esperando hasta que el almacén de discos cargaba el disco en cuestión. Obviamente, esta espera era un asunto estrictamente relativo al hardware y no demandaba uso de CPU. Otros ejemplos de tiempos de espera incluyen cuándo estamos imprimiendo un documento o esperando respuesta de nuestro disco duro o CD-ROM. En ambos casos, la CPU no está siendo utilizada. Estos casos son ejemplos perfectos de procesos que pueden ser enviados a hilos de ejecución secundarios.

Cuándo no usar hilos de ejecución

Es un error común entre los principiantes en el uso de los hilos de ejecución el intentar usarlos sistemáticamente en todas las aplicaciones. ¡Esta situación puede ser mucho peor que no poder usarlos en absoluto! Como pasa con cualquier otra herramienta en nuestro arsenal de programadores, deberíamos usar los hilos de ejecución solamente cuando el caso lo requiera. Deberíamos evitar usar múltiples hilos de ejecución en nuestras aplicaciones en al menos los siguientes casos (descritas en las siguientes secciones): cuando los costes superan los beneficios; cuando no tenemos claro cuál de los dos casos (con hilos de ejecución o sin ellos) da mejor resultado, y cuando no podemos encontrar una buena razón para usarlos.

Los costes superan a los beneficios

Como vimos en «Seguridad y sincronización en hilos de ejecución», escribir aplicaciones multihilo lleva un cierto tiempo y esfuerzo en el diseño del código. Habrá ocasiones en las que los leves beneficios que obtengamos del uso de múltiples hilos de ejecución no justifican el tiempo extra que necesitaremos para hacer nuestro código a prueba de problemas con los hilos.

No hemos evaluado ambos casos

Si somos nuevos en la programación multihilo, puede sorprendernos el hecho de que a menudo el esfuerzo extra requerido para la creación y planificación de hilos de ejecución por la CPU *pueden hacer que una aplicación monohilo sea más rápida!* Todo depende de qué es lo que estemos haciendo y de si realmente estamos dividiendo tareas independientes en hilos. Por ejemplo, si tenemos que leer tres archivos desde el disco, lanzar tres hilos de ejecución no mejorará nuestra situación porque todas ellas utilizan el mismo disco duro. Por lo tanto, deberemos asegurarnos siempre de medir los rendimientos de prototipos monohilo y con múltiples hilos de ejecución de nuestro sistema antes de pasar por el tiempo y coste extra que implica diseñar una solución multihilo, dado que en términos de rendimiento nos puede salir el tiro por la culata.

No hay una buena razón para hacerlo

El uso de múltiples hilos de ejecución *no* debe ser nuestra opción por defecto. Debido a las inherentes complejidades de la escritura de aplicaciones multihilo, deberíamos hacer uso de código monohilo, a menos que tengamos una buena razón para no hacerlo.

RESUMEN

El uso de múltiples hilos de ejecución permite a las aplicaciones dividir sus tareas de modo que trabajen de manera independientemente unas de otras con el fin de hacer el mejor uso posible del tiempo del procesador. Sin embargo, el añadir múltiples hilos de ejecución a una aplicación no es la elección correcta en todas las situaciones, y puede, en determinadas ocasiones, hacer que la aplicación vaya más lenta. La creación y gestión de hilos de ejecución en C# se hace por medio de la clase *System.Threading.Thread*. Un importante concepto relativo a la creación y uso de hilos de ejecución es la seguridad de hilos. Seguridad de hilos de ejecución implica que los miembros de un objeto siempre mantienen un estado válido cuando se usan de manera concurrente por hilos de ejecución múltiples. Es importante que junto con la sintaxis del uso de múltiples hilos de ejecución, también entendamos cuándo usarla: para aumentar la concurrencia, para simplificar nuestros diseños y para optimizar el uso del tiempo de CPU.

Cómo obtener información sobre metadatos con Reflection

En el Capítulo 2, «Presentando Microsoft .NET», mencionamos el hecho de que el compilador genera un ejecutable portable Win32, compuesto fundamentalmente de MSIL y metadatos. Una funcionalidad muy potente de .NET es que permite escribir código para acceder a los metadatos de una aplicación vía un proceso denominado *reflexión**. Por ponerlo claramente, ‘reflection’ es la capacidad de descubrir la información de un tipo determinado en tiempo de ejecución. Este capítulo describe la API Reflection y cómo podemos usarla para iterar a través de los módulos y tipos de un ensamblaje y para recuperar las diferentes características definidas en tiempo de diseño para un tipo. Veremos también varios usos avanzados de la ‘reflection’, como la posibilidad de invocar dinámicamente métodos y usar la información de tipo (por medio de enlace en tiempo de ejecución), e incluso ¡crear y ejecutar código MSIL en tiempo de ejecución!

LA JERARQUÍA DE LA API REFLECTION

La API Reflection de .NET es realmente una amalgama de clases —parte de las cuales se muestran en la Figura 16.1— que están definidas en el espacio de nombres *System.Reflection*. Estas clases nos permiten desplazarnos a través de información del ensamblaje y del tipo. Podemos posicionarnos y desplazarnos desde cualquier parte de esta jerarquía, dependiendo de las necesidades específicas de diseño de nuestra aplicación.

Observe que estas clases abarcan una gran cantidad de funcionalidad. En lugar de enumerar cada método y cada campo para cada clase, presentaremos una visión general de las clases principales y mostraremos un ejemplo que ilustra la funcionalidad que probablemente sea la que necesitemos incorporar a nuestras aplicaciones.

* *Reflection* en el original. Usaremos la acepción inglesa en lo sucesivo, al ser ésta el nombre del espacio de nombres y de la API (*N. de los t.*).

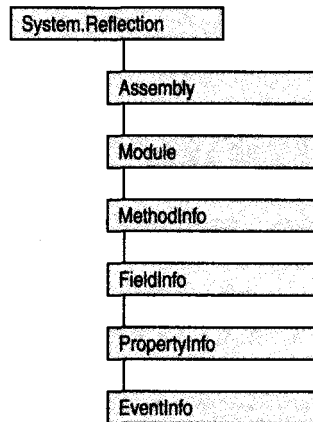


Figura 16.1. Vista parcial de la jerarquía de clases de `System.Reflection`.

LA CLASE *TYPE*

En el corazón de 'reflection' reside la clase *System.Type*. La clase *System.Type* es una clase abstracta que representa un tipo en el sistema común de tipos (CTS) y nos permite hacer solicitudes por nombre de tipo, el módulo y espacio de nombres que incluyen el tipo, y si el tipo es un tipo valor o un tipo referencia.

Obteniendo el tipo de una instancia

El ejemplo siguiente muestra cómo obtener un objeto *Type* para un *int* instanciado:

```

using System;
using System.Reflection;

class TypeObjectFromInstanceApp
{
    public static void Main(string[] args)
    {
        int i = 6;
        Type t = i.GetType();
        Console.WriteLine(t.Name);
    }
}

```

Obteniendo el tipo de un nombre

Aparte de recuperar el objeto *Type* de una variable en concreto, podemos igualmente crear un objeto *Type* a partir de un nombre de tipo. En otras palabras, no necesitamos tener una instancia del tipo. Veamos un ejemplo para el tipo *System.Int32*:

```
using System;
using System.Reflection;

class TypeObjectFromNameApp
{
    public static void Main(string[] args)
    {
        Type t = Type.GetType("System.Int32");
        Console.WriteLine(t.Name);
    }
}
```

Observe que no podemos usar los alias de C# cuando invocamos el método *Type.GetType*, porque este método se utiliza en todos los lenguajes. Por lo tanto, no podemos usar el tipo específico de C# *int* en lugar de *System.Int32*.

Interrogando a los tipos

La clase *System.Type* también le permite consultar un tipo acerca de casi cualquiera de sus atributos, incluyendo el modificador de acceso a tipos, si el tipo está anidado, sus propiedades COM, etc. El siguiente código ilustra esto usando varios tipos estándar y algunos tipos como ejemplo:

```
using System;
using System.Reflection;

interface DemoInterface
{
}

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

public class DemoBaseClass
{
}

public class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class QueryTypesApp
{
}
```

```

public static void QueryType(string typeName)
{
    try
    {
        Type type = Type.GetType(typeName);
        Console.WriteLine("Nombre de tipo: {0}", type.FullName);
        Console.WriteLine("\tHasElementType = {0}",
            type.HasElementType);
        Console.WriteLine("\tIsAbstract = {0}", type.IsAbstract);
        Console.WriteLine("\tIsAnsiClass = {0}", type.IsAnsiClass);
        Console.WriteLine("\tIsArray = {0}", type.IsArray);
        Console.WriteLine("\tIsAutoClass = {0}", type.IsAutoClass);
        Console.WriteLine("\tIsAutoLayout = {0}",
            type.IsAutoLayout);
        Console.WriteLine("\tIsByRef = {0}", type.IsByRef);
        Console.WriteLine("\tIsClass = {0}", type.IsClass);
        Console.WriteLine("\tIsCOMObject = {0}", type.IsCOMObject);
        Console.WriteLine("\tIsContextful = {0}",
            type.IsContextful);
        Console.WriteLine("\tIsEnum = {0}", type.IsEnum);
        Console.WriteLine("\tIsExplicitLayout = {0}",
            type.IsExplicitLayout);
        Console.WriteLine("\tIsImport = {0}", type.IsImport);
        Console.WriteLine("\tIsInterface = {0}",
            type.IsInterface);
        Console.WriteLine("\tIsLayoutSequential = {0}",
            type.IsLayoutSequential);
        Console.WriteLine("\tIsMarshalByRef = {0}",
            type.IsMarshalByRef);
        Console.WriteLine("\tIsNestedAssembly = {0}",
            type.IsNestedAssembly);
        Console.WriteLine("\tIsNestedFamANDAssem = {0}",
            type.IsNestedFamANDAssem);
        Console.WriteLine("\tIsNestedFamily = {0}",
            type.IsNestedFamily);
        Console.WriteLine("\tIsNestedFamORAssem = {0}",
            type.IsNestedFamORAssem);
        Console.WriteLine("\tIsNestedPrivate = {0}",
            type.IsNestedPrivate);
        Console.WriteLine("\tIsNestedPublic = {0}",
            type.IsNestedPublic);
        Console.WriteLine("\tIsNotPublic = {0}",
            type.IsNotPublic);
        Console.WriteLine("\tIsPointer = {0}",
            type.IsPointer);
        Console.WriteLine("\tIsPrimitive = {0}",
            type.IsPrimitive);
        Console.WriteLine("\tIsPublic = {0}",
            type.IsPublic);
        Console.WriteLine("\tIsSealed = {0}",
            type.IsSealed);
        Console.WriteLine("\tIsSerializable = {0}",
            type.IsSerializable);
        Console.WriteLine("\tIsServicedComponent = {0}",
            type.IsServicedComponent);
        Console.WriteLine("\tIsSpecialName = {0}",
            type.IsSpecialName);
    }
}

```

```
        Console.WriteLine("\tIsUnicodeClass = {0}",
                           type.IsUnicodeClass);
        Console.WriteLine("\tIsValueType = {0}",
                           type.IsValueType);
    }
    catch (System.NullReferenceException)
    {
        Console.WriteLine("{0} no es un tipo válido",
                           typeName);
    }
}

public static void Main(string[] args)
{
    QueryType("System.Int32");
    QueryType("System.Int64");
    QueryType("System.Type");

    QueryType("DemoAttr");
    QueryType("DemoEnum");
    QueryType("DemoBaseClass");
    QueryType("DemoDerivedClass");
    QueryType("DemoStruct");
}
```

CÓMO TRABAJAR CON ENSAMBLAJES Y MÓDULOS

Trataremos los ensamblajes (*assemblies*) con mucho más detalle en el Capítulo 18, «Cómo trabajar con ensamblajes». Para el propósito que nos atañe ahora, basta con saber que los ensamblajes son archivos físicos que consisten en múltiples archivos PE (Portable Executable) .NET. La principal ventaja de los ensamblajes es que permite agrupar funcionalidades semánticamente para una gestión de versiones e implantación más sencillos. La representación en tiempo de ejecución en .NET de un ensamblaje (y la raíz de la jerarquía de objetos 'reflection') es la clase *Assembly*.

Podemos hacer muchas cosas con la clase *Assembly*. Aquí tenemos unos cuantos ejemplos de algunas que veremos en breve:

- Iterar a través de los tipos de un ensamblaje.
- Listar los módulos de un ensamblaje.
- Determinar información de identificación, como el nombre y la localización físicas del ensamblaje.
- Inspeccionar la versión e información de seguridad.
- Recuperar el punto de entrada al ensamblaje.

Iterar a través de los tipos de un ensamblaje

Para iterar a través de todos los tipos pertenecientes a un ensamblaje dado, tan sólo necesitamos instanciar un objeto *Assembly* y luego solicitar el array *Types* para ese ensamblaje. Aquí tenemos un ejemplo:

```
using System;
using System.Diagnostics;
using System.Reflection;

class DemoAttr : System.Attribute
{
}

enum DemoEnum
{
}

class DemoBaseClass
{
}

class DemoDerivedClass : DemoBaseClass
{
}

class DemoStruct
{
}

class GetTypesApp
{
    protected static string GetAssemblyName(string[] args)
    {
        string assemblyName;

        if (0 == args.Length)
        {
            Process p = Process.GetCurrentProcess();
            assemblyName = p.ProcessName + ".exe";
        }
        else
            assemblyName = args[0];

        return assemblyName;
    }

    public static void Main(string[] args)
    {
        string assemblyName = GetAssemblyName(args);

        Console.WriteLine("Cargando información de:" + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);
    }
}
```

```

Type[] types = a.GetTypes();
foreach (Type t in types)
{
    Console.WriteLine("\nInformación de tipo de:" +
                      t.FullName);
    Console.WriteLine("\tClase Origen =" +
                      t.BaseType.FullName);
}
}
}

```

NOTA Si intentamos ejecutar código que necesita verificación de seguridad —como por ejemplo código que use la API Reflection— en una intranet, será necesario modificar la política de seguridad existente. Una manera de hacerlo es con la herramienta (*caspol.exe*) de Gestión de contraseñas para política de seguridad (Code Access Security Policy tool). Aquí tenemos un ejemplo de uso de dicha herramienta:

```

caspol -addgroup 1.2 -url "file://somecomputer/someshare/*"
SkipVerification

```

Este ejemplo proporciona un permiso adicional —en este caso, *SkipVerification*— basado en la URL del código en ejecución. Observe que podemos asimismo modificar la política para todo el código para una área determinada o sólo para un ensamblaje particular, basado en una firma digital o incluso en una operación de dispersión (*hashing*) sobre los bits. Para ver los argumentos válidos para la utilidad *caspol.exe*, podemos teclear *caspol -?* en el símbolo del sistema o usar la documentación en pantalla de MSDN.

La primera parte del método *Main* no es realmente interesante —ese código determina si hemos pasado un nombre de ensamblaje a la aplicación. Si no lo hemos hecho, se usa el método estático *GetProcessName* de la clase *Process* para determinar el nombre de la aplicación actualmente en ejecución.

Después de eso, comenzamos a ver lo sencillas que son en realidad la mayoría de las tareas de ‘reflection’. El método más sencillo para instanciar un objeto *Assembly* es invocar al método *Assembly.LoadFrom*. Este método toma un único argumento: una cadena que representa el nombre del archivo físico que queremos cargar. A partir de ahí, una llamada al método *Assembly.GetTypes* nos devuelve un array de objetos *Type*. Una vez llegado a este punto, ¡tenemos un objeto que describe todos y cada uno de los tipos existentes en el ensamblaje! Finalmente, la aplicación imprime su clase base.

A continuación tenemos el resultado de la ejecución de esta aplicación cuando o bien especificamos el archivo *gettypes.exe* o no pasamos ningún argumento a la aplicación:

Cargando información de *GetTypes.exe*

```

Información de tipo de: DemoAttr
    Clase base = System.Attribute

Información de tipo de: DemoEnum
    Clase base = System.Enum

Información de tipo de: DemoBaseClass
    Clase base = System.Object

Información de tipo de: DemoDerivedClass
    Clase base = DemoBaseClass

Información de tipo de: DemoStruct
    Clase base = System.Object

Información de tipo de: AssemblyGetTypesApp
    Clase base = System.Object

```

Cómo conseguir una lista de los módulos de un ensamblaje

Aunque la mayor parte de las aplicaciones de este libro constan de un único módulo, podemos crear ensamblajes compuestos de múltiples módulos. Podemos recuperar módulos de un objeto *Assembly* de dos maneras distintas. La primera es solicitar un array de todos los módulos. Esto nos permite iterar a través de ellos y recuperar cualquier información que necesitemos. La segunda es recuperar un módulo específico. Veamos más en detalle cada una de estas aproximaciones.

Para iterar a través de los módulos de un ensamblaje, necesitamos crear un ensamblaje con más de un módulo. Vamos a hacerlo moviendo *GetAssemblyName* a su propia clase y poniendo dicha clase en un archivo denominado *AssemblyUtils.netmodule*; así:

```

using System.Diagnostics;

namespace MyUtilities
{
    public class AssemblyUtils
    {
        public static string GetAssemblyName(string[] args)
        {
            string assemblyName;

            if (0 == args.Length)
            {
                Process p = Process.GetCurrentProcess();
                assemblyName = p.ProcessName + ".exe";
            }
            else
            {
                assemblyName = args [0 ];
            }

            return assemblyName;
        }
    }
}

```

El módulo se crea entonces usando el siguiente comando:

```
csc /target:module AssemblyUtils.cs
```

El modificador */target:module* hace que el compilador genere un módulo para su posterior inclusión en el ensamblaje. El comando creará un archivo denominado *AssemblyUtils.netmodule*. En el Capítulo 18 veremos las diferentes opciones a la hora de crear ensamblajes y módulos en mayor detalle.

Llegados a este punto, crearemos un módulo secundario para tener dónde utilizar reflexión. A continuación tenemos la aplicación que usará la clase *AssemblyUtils*. Observe la instrucción *using* cuando hacemos referencia al espacio de nombres *MyUtilities*.

```
using System;
using System.Reflection;
using MyUtilities;

class GetModulesApp
{
    public static void Main(string[] args)
    {
        string assemblyName = AssemblyUtils.GetAssemblyName(args);

        Console.WriteLine("Cargando información de" + assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

        Module[] modules = a.GetModules();
        foreach(Module m in modules)
        {
            Console.WriteLine("Módulo:" + m.Name);
        }
    }
}
```

Para compilar esta aplicación y añadir el módulo *AssemblyUtils.netmodule* a su ensamblaje, necesitamos añadir estos modificadores de línea de comando:

```
csc /addmodule:AssemblyUtils.netmodule GetModules.cs
```

Llegados a este punto, tenemos un ensamblaje con dos módulos diferentes. Para verlo, ejecutemos la aplicación. Los resultados deberían ser los siguientes:

```
Cargando información de GetModulesApp.exe
Módulo: GetModulesApp.exe
Módulo: AssemblyUtils.netmodule
```

Como podemos ver en el código, simplemente hemos instanciado un objeto *Assembly* y hemos invocado el método *GetModules*. A partir de ahí, hemos iterado sobre el array que nos ha devuelto y hemos impreso el nombre de cada uno de ellos.

ENLACE EN TIEMPO DE EJECUCIÓN CON 'REFLECTION'

Hace unos cuantos años trabajé para la división Multimedia de IBM para su producto 'IBM/World Book Multimedia Encyclopedia'. Uno de los desafíos a los que tuvimos que enfrentarnos fue el crear una aplicación que permitiera al usuario configurar diferentes protocolos de comunicación para usar con los servidores World Book. Debía ser una solución dinámica, porque el usuario podría estar continuamente añadiendo y eliminando diferentes protocolos (por ejemplo, TCP/IP, IGN, CompuServ, y otros) desde su sistema. Sin embargo, la aplicación tendría que «saber» qué protocolos estaban presentes para que el usuario pudiera seleccionar un protocolo específico para configurar y usar. La solución a la que llegamos involucraba el crear DLL con una extensión especial e instalarlas en la carpeta de la aplicación. Entonces, cuando el usuario quería ver una lista de protocolos instalados, la aplicación invocaría la función *Win32 LoadLibrary* para cargar cada DLL y entonces invocar la función *GetProcAddress* para conseguir un puntero a función para la función que deseáramos. Esto es un ejemplo perfecto de enlace en tiempo de ejecución en programación estándar Win32, en la que el compilador no sabe nada sobre estas llamadas a la hora de construir el ejecutable. Como podemos ver en el siguiente ejemplo, esta misma tarea se podría ejecutar en .NET usando la clase *Assembly*, reflexión de tipos y una nueva clase llamada clase *Activator*.

Para que todo se ponga en marcha, creemos una clase abstracta denominada *CommProtocol*. Definiremos esta clase en su propia DLL de tal modo que pueda ser compartida por múltiples DLL que quieran derivar de ella. (Observe que los parámetros de línea de comando están incorporados en los comentarios del código).

```
//CommProtocol.cs
//Construir con los siguientes modificadores de línea de comando
//      csc /t:library commprotocol.cs
public abstract class CommProtocol
{
    public static string DLLMask = "CommProtocol*.dll";
    public abstract void DisplayName();
}
```

Ahora, crearemos dos DLL separadas, cada una representando un protocolo de comunicaciones y conteniendo una clase derivada de la clase abstracta *CommProtocol*. Observe que ambos necesitan referenciar a la DLL *CommProtocol* cuando se compile. Aquí tenemos la DLL IGN (IBM Global Network):

```
//CommProtocolIGN.cs
//Construir con los siguientes modificadores de línea de comando
//      csc /t:library CommProtocolIGN.cs /r:CommProtocol.dll
using System;

public class CommProtocolIGN : CommProtocol
{
    public override void DisplayName()
    {
```

```

    Console.WriteLine("Esta es la IBM Global Network");
}
}

```

Y esta es la DLL para TCP/IP:

```

//CommProtocolTcpIp.cs
//Construir con los siguientes modificadores de línea de comando
// csc /t:library CommProtocolTcpIp.cs /r:CommProtocol.dll
using System;

public class CommProtocolTcpIp : CommProtocol
{
    public override void DisplayName()
    {
        Console.WriteLine("Este es el protocolo TCP/IP");
    }
}

```

Veamos ahora lo sencillo que es cargar dinámicamente un ensamblaje, buscar un tipo, instanciar dicho tipo, e invocar uno de sus métodos. (Por cierto, hay un archivo independiente llamado `BuildLateBinding.cmd` en el CD que acompaña este libro que también construirá todos estos archivos).

```

using System;
using System.Reflection;
using System.IO;

class LateBindingApp
{
    public static void Main()
    {
        string[] fileNames = Directory.GetFiles
            (Environment.CurrentDirectory,
             CommProtocol.DLLMask);
        foreach(string fileName in fileNames)
        {
            Console.WriteLine("Cargando DLL'{0}'", fileName);

            Assembly a = Assembly.LoadFrom(fileName);

            Type[] types = a.GetTypes();
            foreach(Type t in types)
            {
                if (t.IsSubclassOf(typeof(CommProtocol)))
                {
                    object o = Activator.CreateInstance(t);

                    MethodInfo mi = t.GetMethod("DisplayName");

                    Console.WriteLine("\t");
                    mi.Invoke(o, null);
                }
            }
        }
    }
}

```

```

else
{
    Console.WriteLine("\tEsta DLL no tiene" +
        "CommProtocol-clase derivada definida");
}
}
}
}
}
}
}

```

Primero usamos la clase *System.IO.Directory* para encontrar las DLL en el directorio actual que correspondan al criterio de búsqueda *CommProtocol*.dll*. El método *Directory.GetFiles* devolverá un array de objetos de tipo *string* que representa los nombres de archivo que encajan con nuestro criterio de búsqueda. Podemos usar entonces un bucle *foreach* para iterar a lo largo del array, invocando el método *Assembly.LoadFrom* que vimos en un punto anterior de este capítulo. Una vez que se crea un ensamblaje para una DLL dada, iteramos por todos los tipos del ensamblaje, invocando el método *Type.SubClassOf* para determinar si el ensamblaje tiene un tipo que deriva de *CommProtocol*. Asumimos que si encontramos uno de ellos tenemos una DLL válida con la que trabajar. Una vez que encontramos el ensamblaje que tiene un tipo derivado de *CommProtocol*, instanciamos un objeto *Activator* y pasamos a su constructor el objeto *type*. Como podemos intuir, dado su nombre, la clase *Activator* se usa para crear dinámicamente, o activar, un tipo.

Entonces usaremos el método *Type.GetMethod* para crear un objeto *MethodInfo*, especificando el nombre de método *DisplayName*. Una vez que hemos hecho esto, podemos usar el método *Invoke* perteneciente al objeto *MethodInfo*, pasándole el tipo activado, y —¡aquí está!— llamamos al método *DisplayName* que reside en la DLL.

CÓMO CREAR Y EJECUTAR CÓDIGO EN TIEMPO DE EJECUCIÓN

Ahora que ya hemos visto cómo utilizar ‘reflection’ con tipos en tiempo de ejecución, a enlazar código en tiempo de ejecución y a ejecutar código dinámicamente, vamos a dar el siguiente paso lógico y crear código sobre la marcha. La creación de tipos en tiempo de ejecución implica el usar el espacio de nombres *System.Reflection.Emit*. Usando las clases de este espacio de nombres, podemos crear un ensamblaje en memoria, crear un módulo para un ensamblaje, definir nuevos tipos para un módulo (incluyendo sus miembros), e incluso producir los códigos de operación (*opcodes*) MSIL para la lógica de la aplicación.

Aunque el código de este ejemplo es muy sencillo, hemos separado el código del servidor —una DLL que contiene una clase que crea un método llamado *HelloWorld*— del código del cliente, una aplicación que instancia la clase generadora de código y llama a su método *HelloWorld*. (Observe que los modificadores para compilar están en los comentarios del código). Adjuntamos una explicación para el código DLL, que mostramos aquí:

```

using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ILGenServer
{
    public class CodeGenerator
    {
        public CodeGenerator()
        {
            //Obtener el actual currentDomain.
            currentDomain = AppDomain.CurrentDomain;

            //Crear ensamblaje en el actual currentDomain.
            assemblyName = new AssemblyName();
            assemblyName.Name = "TempAssembly";

            assemblyBuilder =
                currentDomain.DefineDynamicAssembly
                    (assemblyName, AssemblyBuilderAccess.Run);

            //crear un módulo en el ensamblaje
            moduleBuilder = assemblyBuilder.DefineDynamicModule
                ("TempModule");

            //crear un tipo en el módulo
            typeBuilder = moduleBuilder.DefineType
                ("TempClass",
                    TypeAttributes.Public);

            //añade un miembro (un método) al tipo
            methodBuilder = typeBuilder.DefineMethod
                ("HelloWorld",
                    MethodAttributes.Public,
                    null, null);

            //Genera MSIL.
            msil = methodBuilder.GetILGenerator();
            msil.EmitWriteLine("Hello World");
            msil.Emit(OpCodes.Ret);

            //Último paso en la "construcción": crear el tipo.
            t = typeBuilder.CreateType();
        }

        AppDomain currentDomain;
        AssemblyName assemblyName;
        AssemblyBuilder assemblyBuilder;
        ModuleBuilder moduleBuilder;
        TypeBuilder typeBuilder;
        MethodBuilder methodBuilder;
        ILGenerator msil;
        object o;

        Type t;
        public Type T
        {

```

```

        get
        {
            return this.t;
        }
    }
}

```

Primero instanciamos un objeto *AppDomain* a partir del dominio actual. (Veremos en el Capítulo 17, «Cómo interoperar con código no gestionado», que los dominios de aplicación (*app domains*) son funcionalmente similares a procesos Win32). Después instanciamos un objeto *AssemblyName*. La clase *AssemblyName* se cubrirá en más detalle en el Capítulo 18; para resumir, diremos que es una clase utilizada por el gestor de caché del ensamblaje para recuperar información sobre un ensamblaje concreto. Una vez que tenemos el dominio de aplicación actual y un nombre de ensamblaje inicializado, invocamos el método *AppDomain.DefineDynamicAssembly* para crear un nuevo ensamblaje. Observe que los dos argumentos que le estamos pasando son un nombre de ensamblaje, así como el modo en que accederemos a dicho ensamblaje. *AssemblyBuilderAccess.Run* define que el ensamblaje se puede ejecutar desde memoria pero no se puede almacenar. El método *AppDomain.DefineDynamicAssembly* devuelve un objeto *AssemblyBuilder*, cuyo tipo se puede convertir a un objeto *Assembly*. Llegados a este punto, tenemos un ensamblaje totalmente operativo en memoria. Ahora necesitamos crear su módulo temporal y el tipo de dicho módulo.

Comenzamos invocando al método *Assembly.DefineDynamicModule* para recuperar un objeto *ModuleBuilder*. Una vez que tenemos dicho objeto, llamamos a su método *DefineType* para crear un objeto *TypeBuilder*, pasándole el nombre del tipo («*TempClass*») y los atributos usados para definirlo (*TypeAttributes.Public*). Ahora que tenemos un objeto *TypeBuilder* a mano, podemos crear cualquier tipo de miembro que deseemos. En este caso, creamos un método usando el método *TypeBuilder.DefineMethod*.

Finalmente, tenemos un flamante tipo nuevo llamado *TempClass* con un método incorporado llamado *HelloWorld*. Ahora todo lo que tenemos que hacer es decidir qué código colocar en este método. Para hacerlo, el código instancia un objeto *ILGenerator* mediante el uso del método *MethodBuilder.GetILGenerator* e invoca los diferentes métodos pertenecientes a *ILGenerator* para escribir código MSIL que se introducirá en el método.

Observe que podemos utilizar código estándar como *Console.WriteLine* utilizando diversos métodos pertenecientes a *ILGenerator*, o podemos producir códigos de operación MSIL utilizando el método *ILGenerator.Emit*. El método *ILGenerator.Emit* lleva como único argumento un miembro de la clase *OpCodes* que se relaciona directamente con un código de operación MSIL.

Por último, invocamos el método *TypeBuilder.CreateType*. Éste debería ser siempre el último paso realizado después de que hayamos definido los miembros para un tipo nuevo. Entonces recuperaremos el objeto *Type* para el nuevo tipo usando el método *Type.GetType*. Este objeto se almacena en una variable de miembro para una recuperación posterior por la aplicación cliente.

Ahora, todo lo que el cliente tiene que hacer es recuperar el miembro *Type* perteneciente a *CodeGenerator*, crear una instancia de *Activator*, instanciar un objeto *Method*-

Info desde el tipo, y entonces invocar el método. Aquí tenemos el código para hacerlo, con una pequeña comprobación de errores añadida para asegurarnos de que las cosas funcionan como debieran.

```
using System;
using System.Reflection;
using ILGenServer;

public class ILGenClientApp
{
    public static void Main()
    {
        Console.WriteLine("Llamando a la función de la DLL para generar" +
                           "un tipo y un método nuevo en memoria...");
        CodeGenerator gen = new CodeGenerator();

        Console.WriteLine("Recuperando el tipo generado dinámicamente...");
        Type t = gen.T;
        if (null != t)
        {
            Console.WriteLine("Instanciando el nuevo tipo...");
            object o = Activator.CreateInstance(t);
            Console.WriteLine("Recuperando el método" +
                              "HelloWorld del tipo...");
            MethodInfo helloWorld = t.GetMethod("HelloWorld");
            if (null != helloWorld)
            {
                Console.WriteLine("Invocando nuestro método" ++
                                  "HelloWorld creado dinámicamente...");
                helloWorld.Invoke(o, null);
            }
            else
            {
                Console.WriteLine("No se pudo localizar" +
                                  "el método HelloWorld");
            }
        }
        else
        {
            Console.WriteLine("No se pudo acceder al tipo" +
                              "desde el servidor");
        }
    }
}
```

Ahora, si compilamos y ejecutamos esta aplicación, veremos el siguiente resultado:

```
Llamando a la función de la DLL para generar un tipo y un método nuevo en
memoria...
Recuperando el tipo generado dinámicamente...
Instanciando el nuevo tipo...
Recuperando el método HelloWorld del tipo...
Invocando nuestro método dinámicamente creado HelloWorld...
Hello World
```

RESUMEN

'Reflection' es la capacidad de descubrir información sobre un tipo concreto en tiempo de ejecución. La API Reflection nos permite hacer cosas como iterar a través de los módulos de un ensamblaje, iterar sobre los tipos de un ensamblaje y recuperar las diferentes características en tiempo de diseño de un tipo. Las tareas más avanzadas de 'reflection' incluyen utilizarla para invocar métodos y usar tipos dinámicamente (vía enlace en tiempo de ejecución), e incluso crear y ejecutar código MSIL en tiempo de ejecución.

Cómo interoperar con código no gestionado

Cualquier lenguaje o entorno de desarrollo nuevo tendría una vida corta si ignorara la existencia de sistemas y código heredado (*legacy systems*) y tan sólo proporcionara un medio de desarrollar sistemas nuevos. Independientemente de lo fantástica que pueda ser una nueva tecnología, sus creadores deben tener en cuenta que durante un tiempo lo nuevo debe coexistir con lo viejo. Para ese fin, los equipos de diseño de .NET y C# han hecho sencillo para los programadores el interoperar con código existente por medio del uso de código no gestionado. Por *código no gestionado* (*unmanaged code*) nos referimos al código que no es gestionado, o controlado, por el entorno de ejecución .NET. En este capítulo, cubriremos los tres ejemplos principales de código no gestionado en .NET, como sigue:

- **Servicios de invocación de plataforma** (Platform Invocation Services). Estos servicios permiten al código .NET acceder a funciones, estructuras e incluso devolución de llamadas (*callbacks*) en bibliotecas de enlace dinámico (DLL) existentes y no gestionadas.
- **Código inseguro**. Escribir código inseguro permite al programador de C# usar construcciones (como por ejemplo punteros) en aplicaciones C#, a expensas de que dicho código sea gestionado por el entorno de ejecución .NET.
- **Interoperabilidad COM**. Este término se refiere a la capacidad tanto del código .NET de usar componentes COM como de las aplicaciones COM de utilizar componentes .NET.

SERVICIOS DE INVOCACIÓN DE PLATAFORMA

Los Servicios de invocación de plataforma de .NET —en ocasiones mencionados como *PInvoke*— permiten al código gestionado trabajar con funciones y estructuras que han

sido exportadas desde DLL. En esta sección veremos cómo invocar funciones de DLL y cómo se usan los atributos para transferir los datos entre una aplicación .NET y una DLL.

Dado que no proporcionamos al compilador C# el código fuente para la función DLL, debemos especificar al compilador la signatura del método nativo, así como información sobre cualquier tipo de valor de retorno y cómo transferir los parámetros que se envían a la DLL.

NOTA Como ya sabemos, podemos crear DLL con C# y otros compiladores .NET. En esta sección hemos evitado usar el término «DLL Win32 no gestionada». Asumimos que en cualquier momento que nos referimos a una DLL nos referimos a la variedad no gestionada.

Cómo declarar una función exportada desde la DLL

El primer problema que afrontaremos es cómo declarar una sencilla función DLL en C#. Usaremos lo que se está convirtiendo, cada vez más, en el ejemplo clásico de *PInvoke* para .NET, el ejemplo «MessageBox» de Win32, para empezar. Entonces procederemos a movernos hacia áreas más avanzadas de la organización de parámetros.

Como aprendimos en el Capítulo 8, «Atributos», los atributos se emplean para proporcionar información en tiempo de diseño para un tipo C#. Por medio de 'reflection', esta información se puede consultar en tiempo de ejecución. C# hace uso de un atributo para permitirnos describirle al compilador la función de la DLL que invocará la aplicación. Este atributo se denomina atributo *DllImport*, y su sintaxis es:

```
[DllImport(Nombre_DLL)]
Modificador_de_Acceso static extern valRetorno Función_de_la_dll (param1, param2...);
```

Como podemos ver, importar una función DLL es tan sencillo como adjuntar el atributo *DllImport* (pasando el nombre de la DLL a su constructor) a la función de la DLL que queramos invocar. Hay que poner especial cuidado en el hecho de que se deben utilizar los modificadores *static* y *extern* también en la función que estamos definiendo. Aquí tenemos el ejemplo *MessageBox* mostrando lo sencillo que es usar *PInvoke*:

```
using System;
using System.Runtime.InteropServices;

class PInvoke1App
{
    [DllImport("user32.dll")]
    static extern int MessageBoxA(int hWnd,
                                string msg,
                                string caption,
                                int type);
```

```

public static void Main()
{
    MessageBoxA(0,
        "¡Hola, Mundo!",
        "¡Esto ha sido llamado desde una aplicación C#!",
        0);
}

```

Ejecutar esta aplicación muestra la esperada ventana de mensaje con el texto «¡Hola, Mundo!».

Fijémonos en la cláusula *using*, que se refiere al espacio de nombres *System.Runtime.InteropServices*. Este es el espacio de nombres que define el atributo *DllImport*. A continuación, observe que hemos definido un método *MessageBoxA* invocado por el método *Main*. Pero ¿qué ocurre si queremos invocar nuestro método interno C# de forma distinta al nombre existente de la función DLL? Podemos conseguirlo usando uno de los atributos de *DllImport* mencionados.

Lo que ocurre en el código que mostramos a continuación es que le estamos diciendo al compilador que queremos que nuestro método se llame ahora *MessageBoxA*. Comoquiera que no especificamos el nombre de la función DLL en el atributo de *DllImport*, el compilador asume que ambos nombres son el mismo.

```

[DllImport("user32.dll")]
static extern int MessageBoxA(int hWnd,
                              string msg,
                              string caption,
                              int type);

```

Para ver cómo cambiar este comportamiento predeterminado, veamos otro ejemplo; esta vez usando un nombre interno *MsgBox*, mientras que estamos en realidad llamando a la función de la DLL *MessageBoxA*.

```

using System;
using System.Runtime.InteropServices;

class Pinvoke2App
{
    [DllImport("user32.dll", EntryPoint="MessageBoxA")]
    static extern int MsgBox(int hWnd,
                              string msg,
                              string caption,
                              int type);

    public static void Main()
    {
        MsgBox(0,
            "¡Hola, Mundo!",
            "¡Esto ha sido llamado desde una aplicación C#!",
            0);
    }
}

```

Como podemos ver, sólo hemos necesitado especificar el parámetro con nombre *EntryPoint* del atributo *DllImport* para ser capaces de nombrar de la manera que deseemos un equivalente interno a una función de una DLL externa.

Lo último que veremos antes de movernos a la organización de parámetros es el parámetro *CharSet*. Este parámetro nos permite especificar el juego de caracteres usado por el fichero DLL. Normalmente, cuando escribimos aplicaciones C++, no especificamos *MessageBoxA* o *MessageBoxW*; mediante una directiva *pragma* le hemos indicado al compilador con anterioridad si estamos usando los juegos de caracteres ANSI o Unicode. Esta es la razón por la que en C++ invocamos a *MessageBox*, y el compilador determina qué versión del juego de caracteres invocar. En el siguiente ejemplo, la función *MessageBoxA* seguirá siendo la invocada como resultado del valor que le pasamos al atributo *DllImport* por medio de su parámetro *CharSet* mencionado.

```
using System;
using System.Runtime.InteropServices;

class PInvoke3App
{
    //CharSet.Ansi producirá una llamada a MessageBoxA.
    //CharSet.Unicode producirá una llamada a MessageBoxW.
    [DllImport("user32.dll", CharSet=CharSet.Ansi)]
    static extern int MessageBox(int hWnd,
                                string msg,
                                string caption,
                                int type);

    public static void Main()
    {
        MessageBox(0,
                  "¡Hola, Mundo!",
                  "¡Esto está llamado desde una aplicación C#!",
                  0);
    }
}
```

La ventaja de usar el parámetro *CharSet* es que podemos crear una variable en nuestra aplicación y hacer que controle qué versión de las funciones invocamos (ANSI o Unicode); no tenemos que cambiar todo el código si decidimos cambiar de una versión a otra.

Cómo utilizar funciones de devolución de llamada en C#

No solamente podemos invocar una función existente en una DLL desde una aplicación C#, sino que una función de la DLL puede asimismo llamar a los métodos C# de nuestra aplicación en escenarios donde permitimos la devolución de llamada (*callback*). Estos escenarios comprenden el uso de cualquiera de las funciones Win32 *EnumXXX*, donde llamamos a una función para que nos enumere algo, pasándole un puntero a la función que será invocada por Windows para cada objeto encontrado. Esto se hace a través

de una combinación de *PInvoke* (para llamar a la función de la DLL) y delegados (para definir la devolución de llamada). Si necesitamos refrescar nuestro conocimiento sobre delegados, podemos echar un vistazo al Capítulo 14, «Delegados y manejadores de eventos».

El siguiente código enumera e imprime las cabeceras de todas las ventanas en el sistema:

```
using System;
using System.Runtime.InteropServices;
using System.Text;

class CallbackApp
{
    [DllImport("user32.dll")]
    static extern int GetWindowText(int hWnd, StringBuilder text, int
        count);

    delegate bool CallbackDef(int hWnd, int lParam);

    [DllImport("user32.dll")]
    static extern int EnumWindows (CallbackDef callback, int lParam);

    static bool PrintWindow(int hWnd, int lParam)
    {
        StringBuilder text = new StringBuilder(255);
        GetWindowText(hWnd, text, 255);

        Console.WriteLine("Cabecera de ventana: {0}", text);
        return true;
    }

    static void Main()
    {
        CallbackDef callback = new CallbackDef(PrintWindow);
        EnumWindows(callback, 0);
    }
}
```

Primero definimos las funciones *Win32 EnumWindows* y *GetWindowText* utilizando el atributo *DllImport*. Una vez hecho esto, definimos un delegado denominado *CallbackDef* y un método denominado *PrintWindows*. A continuación, lo único que tenemos que hacer en *Main* es instanciar el delegado *CallbackDef* (pasándole el método *PrintWindows*) e invocar el método *EnumWindows*. Para cada ventana encontrada en el sistema, *Windows* llamará al método *PrintWindows*.

El método *PrintWindows* es interesante porque usa la clase *StringBuilder* para crear una cadena de longitud fija que se pasa a la función *GetWindowText*. Esta es la razón por la que la función *GetWindowText* se define de la siguiente manera:

```
static extern int GetWindowText(int hWnd, StringBuilder text, int count);
```

De cualquier modo, la razón para todo esto es que la función de la DLL no puede alterar una cadena, así que no podemos usar el tipo *string*. E incluso si intentamos pasarlo

por referencia, el código que invoca no tiene manera de inicializar una cadena del tamaño correcto. Esta es la razón de la aparición de la clase *StringBuilder*. Un objeto *StringBuilder* puede ser despojado de su referencia y modificado por la función llamada, en tanto en cuanto que la longitud del texto no exceda la máxima longitud pasada al constructor de *StringBuilder*.

Envío de datos y *PInvoke*

A pesar de que normalmente no veamos el envío de datos (*marshalling*) o no definamos en ninguna parte cómo funciona en detalle, cada vez que invocamos a una función en una DLL, .NET tiene que enviar tanto los parámetros para dicha función como el valor de retorno para la aplicación .NET que lo llama. No tuvimos que hacer nada en los ejemplos anteriores de este capítulo para que el envío tuviera lugar, porque .NET ha definido un tipo nativo por defecto para cada tipo .NET. Por ejemplo, el segundo y tercer parámetro de las funciones *MessageBoxA* y *MessageBoxW* se definieron como tipo *string*. Sin embargo, el compilador de C# sabe que el equivalente de una cadena C# para Win32 es LPSTR. Pero ¿qué ocurre si queremos redefinir el envío de datos por defecto de .NET? Para hacerlo, utilizamos el atributo *MarshalAs*, que está asimismo definido en el espacio de nombres *System.Runtime.InteropServices*.

En el siguiente ejemplo, de nuevo usamos *MessageBox* por razones de simplicidad. Aquí hemos escogido usar la versión Unicode de la función Win32 *MessageBox*. Como ya sabemos por la sección anterior, sólo necesitamos especificar la enumeración *CharSet.Unicode* para el parámetro con nombre *CharSet* del atributo *DllImport*. Sin embargo, en este caso queremos que el compilador envíe los datos como caracteres largos (*wide character*, LPWSTR), así que usamos el atributo *MarshalAs* y especificamos con una enumeración *UnmanagedType* el tipo al que deseamos que se convierta nuestro tipo original. Aquí está el listado:

```
using System;
using System.Runtime.InteropServices;

class PInvoke4App
{
    [DllImport("user32.dll", CharSet=CharSet.Unicode)]
    static extern int MessageBox(int hWnd,
                                [MarshalAs(UnmanagedType.LPWStr)]
                                string msg,
                                [MarshalAs(UnmanagedType.LPWStr)]
                                string caption,
                                int type);

    public static void Main()
    {
        MessageBox(0,
                  "¡Hola, Mundo!",
                  "¡Esto está siendo llamado desde una aplicación C#!",
                  0);
    }
}
```

Observemos que el atributo *MarshalAs* puede ir asociado a parámetros de método (como en este ejemplo), valores de retorno de métodos, y por último, campos de estructuras y clases. Observemos también que para cambiar el envío de datos por defecto para el valor de retorno de un método, necesitamos asociar el atributo *MarshalAs* al propio método.

ESCRIBIENDO CÓDIGO INSEGURO

Una preocupación que afecta a muchos programadores para pasar de C++ a C# tiene que ver con si mantendrán «control absoluto» sobre la manipulación de memoria en los casos que lo necesiten, un hecho que tiene que ver con código inseguro. A pesar de su ominoso nombre, el *código inseguro* no es código que sea inherentemente inseguro y no digno de confianza —es código para el que el entorno de ejecución .NET no controlará la asignación y liberación de memoria. La capacidad de escribir código inseguro nos proporciona determinadas ventajas cuando estamos usando punteros para comunicarnos con código heredado (como API de C) o cuando nuestra aplicación exige manipulación directa de la memoria (habitualmente por razones de rendimiento).

Escribimos código inseguro utilizando dos palabras reservadas: *unsafe* y *fixed*. La palabra reservada *unsafe* especifica que el bloque marcado se ejecutará en un contexto no gestionado. Esta palabra reservada se puede aplicar a todos los métodos, incluyendo constructores y propiedades, e incluso a bloques de código que se encuentra dentro de métodos. La palabra reservada *fixed* es responsable del marcado (*pinning*) de los objetos gestionados. El marcado es el acto de especificarle al recolector de basura (*garbage collector*=GC) que el objeto en cuestión no se puede mover. En el trabajo normal durante la ejecución de una aplicación, los objetos son asignados y desasignados y aparecen «espacios libres» en la memoria. En lugar de permitir que la memoria se fragmente, el entorno de ejecución .NET mueve los objetos de sitio para hacer un uso de la memoria lo más eficiente posible. Obviamente, esto no es bueno cuando tenemos un puntero a una dirección específica de memoria y luego —sin saberlo nosotros— el entorno de ejecución .NET mueve el objeto de esa dirección, dejándonos con un puntero no válido. Comoquiera que el GC mueve objetos en memoria debido a que quiere incrementar la eficiencia de la aplicación, debemos usar esta palabra reservada de manera juiciosa.

Cómo utilizar punteros en C#

Observemos algunas reglas sobre el uso de punteros y código inseguro en C#, y entonces nos lanzaremos a ver algunos ejemplos. Los punteros se pueden obtener sólo para tipos valor, arrays y cadenas. Démonos cuenta también de que en el caso de los arrays, el primer elemento debe ser un tipo valor, porque C# realmente devuelve un puntero al primer elemento del array y no el array en sí mismo. Por lo tanto, desde la perspectiva del compilador, todavía sigue devolviendo un puntero a un tipo valor y no un tipo referencia.

La Tabla 17.1 muestra cómo se soporta la semántica estándar de los punteros C/C++ en C#

Tabla 17.1. Operadores de gestión de punteros C/C++

Operador	Descripción
&	El operador <i>dirección-de</i> devuelve un puntero que representa la dirección de memoria de la variable.
*	El operador de <i>desreferencia</i> se usa para indicar el valor al que apunta el puntero.
->	Los operadores de <i>desreferencia</i> y <i>acceso a miembro</i> se usan para acceder a miembros concretos de una estructura referenciada y para devolver sus valores asociados.

El siguiente ejemplo será familiar para cualquier desarrollador C o C++. Aquí invocamos un método que genera dos punteros a variables de tipo *int* y modifica sus valores antes de volver al invocador. No es particularmente espectacular, pero ilustra cómo usar punteros en C#.

//Compile esta aplicación con la opción /unsafe.

```
using System;
```

```
class Unsafe1App
```

```
{
    public static unsafe void GetValues(int* x, int* y)
    {
        *x = 6;
        *y = 42;
    }

    public static unsafe void Main()
    {
        int a = 1;
        int b = 2;
        Console.WriteLine("Antes de llamar a GetValues() : a = {0}, b = {1}",
                           a, b);
        GetValues(&a, &b);
        Console.WriteLine("Después de llamar a GetValues() : a = {0},
                           b = {1}", a, b);
    }
}
```

Este ejemplo necesita ser compilado con la opción /unsafe del compilador. La salida de esta aplicación debería ser la siguiente:

```
Antes de llamar a GetValues() : a = 1, b = 2
Después de llamar a GetValues() : a = 6, b = 42
```

La instrucción *fixed*

La instrucción *fixed* tiene la siguiente sintaxis:

fixed (tipo ptr = expresión) instrucción*

Como mencionamos, la instrucción le indica al GC que no se preocupe de gestionar la variable especificada. Observemos que *tipo* es un tipo no gestionado o *void*; *expresión* es cualquier expresión que dé como resultado un puntero a *tipo*, e *instrucción* se refiere al bloque de código para el que se aplica el marcado de la variable. Podemos ver un ejemplo simple a continuación:

```
using System;

class Foo
{
    public int x;
}

class Fixed1App
{
    unsafe static void SetFooValue(int* x)
    {
        Console.WriteLine("Usando la referencia del puntero para modificar
                           foo.x");
        *x = 42;
    }

    unsafe static void Main()
    {
        //Crea una instancia de la estructura.
        Console.WriteLine("Creando la clase Foo");
        Foo foo = new Foo();

        Console.WriteLine("foo.x inicializado con el valor {0}", foo.x);

        //La instrucción fixed marca el objeto foo hasta que
        //la instrucción entre llaves termine.
        Console.WriteLine("Asignando un puntero a foo.x");
        //Asigna la dirección del objeto foo a Foo*.
        fixed(int* f = &foo.x)
        {
            Console.WriteLine("Llamando a SetFooValue pasando " +
                              "un puntero a foo.x");
            SetFooValue(f);
        }

        //Mostramos que realmente cambiamos el valor del miembro vía
        //su puntero.
        Console.WriteLine("Después de volver de " +
                           "SetFooValue, foo.x = {0}", foo.x);
    }
}
```

Este código instancia una clase llamada *Foo*, y dentro de una instrucción *fixed*, marca dicho objeto mientras asigna la dirección de su primer miembro a una variable de tipo *int** (el tipo requerido por el método *SetFooValue*). Observemos que la instrucción *fixed* sólo se usa para delimitar el código que se verá afectado si el GC intentara mover el objeto *Foo*. Este es un aspecto sutil, pero importante, para bloques de código más grandes y con un tiempo de ejecución mayor donde queremos minimizar la cantidad de tiempo en la que tenemos un objeto marcado. Si compilamos y ejecutamos el código anterior, se genera la siguiente salida:

```
Creando la clase Foo
foo.x inicializado con el valor 0
Asignando un puntero a foo.x
Llamando a SetFooValue pasando un puntero a foo.x
Usando la referencia del puntero para modificar foo.x
Después de volver de SetFooValue, foo.x = 42
```

NOTA Un punto importante sobre variables marcadas es el hecho de que el compilador C# no restringe el acceso a una variable marcada al ámbito inseguro. Por ejemplo, podemos usar una variable marcada como un operando por la derecha (valor-*der*) para un operando por la izquierda (valor-*izq*) que está definido en un ámbito que engloba al bloque inseguro. En relación con la posibilidad de que el compilador emita un aviso o un error en este caso, es la responsabilidad del desarrollador el tener cuidado cuando utiliza las variables marcadas como operandos por la derecha.

INTEROPERABILIDAD COM

Si se ha llegado a preguntar cómo funcionan todos esos componentes COM que hemos estado escribiendo a lo largo de los años con el entorno de ejecución .NET, esta es su sección. Veremos cómo los componentes clásicos —vaya, duele ver a COM siendo llamado COM *clásico*— están posicionados en el mundo .NET.

Un mundo indómito

Como ha podido ver a lo largo de este libro, no hay duda de que el entorno .NET y el lenguaje C# se combinan para formar una poderosa manera de construir sistemas basados en componentes. Sin embargo, ¿qué ocurre con las toneladas de componentes COM existentes y reutilizables que hemos construido a lo largo de los últimos años, por no mencionar todas las tazas de café y las noches sin dormir? ¿Significa .NET el fin de dichos componentes? ¿Pueden trabajar codo con codo con el entorno gestionado de .NET? Para todos los que nuestro trabajo consiste en programar COM y para aquellos que viven con el mantra «COM es bueno», tenemos excelentes noticias. COM está aquí para

quedarse, y las aplicaciones gestionadas por el entorno .NET pueden aprovecharse de los componentes COM existentes. Como estamos a punto de ver, los componentes clásicos COM interoperan con el entorno de ejecución .NET a través de una capa de interoperabilidad (COM Interop) que maneja toda la fontanería necesaria para que los mensajes que se pasan entre el entorno de ejecución gestionado y los componentes COM que operan en un reino no gestionado.

Poniéndonos en marcha

Comoquiera que saltar sobre la capa de interoperabilidad COM puede ser un poco excesivo al principio, vamos a renunciar a las definiciones técnicas durante unos minutos y vamos a pasar a un ejemplo realista de donde podría querer utilizar un componente COM desde una aplicación .NET. Según vayamos avanzando, iremos explicando qué es lo que pasa y cómo podemos utilizar lo que aprendamos aquí y usarlo en nuestras propias aplicaciones.

En este ejemplo, vamos a asumir que tenemos un componente COM *AirlineInfo* escrito con Microsoft Visual C++ y ATL. No recorreremos todos los pasos necesarios para construir este componente, porque queremos concentrarnos en los aspectos .NET y C#. Sin embargo, explicaremos el código principal y mencionaremos que el proyecto completo en Visual C++ está en el CD que acompaña este libro.

Nuestro componente COM está diseñado para generar los detalles de llegadas para una línea aérea específica. En aras de la máxima simplicidad, digamos que el componente devuelve detalles para la línea aérea «Air Scooby IC 5678» y devolverá un error para cualquier otra línea aérea. Hemos insertado a propósito este mecanismo de error para que podamos ver en detalle cómo el error producido por el componente COM se puede propagar de vuelta y puede ser recibido por la aplicación cliente .NET que hace la llamada.

Este es el IDL para el componente COM:

```
interface IAirlineInfo:IDispatch
{
    [id(1), helpstring("método GetAirlineTiming")]
    HRESULT GetAirlineTiming([in] BSTR bstrAirline, [out, retval]
        BSTR* pBstrDetails);

    [propget, id(2), helpstring("propiedad LocalTimeAtOrlando")]
    HRESULT LocalTimeAtOrlando([out, retval] BSTR *pVal);
};
```

Nada espectacular aquí, ni siquiera para el más joven de los desarrolladores COM. Tenemos una interfaz llamada *IAirlineInfo* con dos métodos: *GetAirlineTiming* y *LocalTimeAtOrlando*. Ahora veamos la implementación real del método *GetAirlineTiming*:

```
STDMETHODIMP CAirlineInfo::GetAirlineTiming(BSTR
    bstrAirline, BSTR *pBstrDetails)
{
```

```

    _bstr_t bstrQueryAirline(bstrAirline);
    if(NULL == pBstrDetails) return E_POINTER;

    if(_bstr_t("Air Scooby IC 5678") == bstrQueryAirline)
    {
        //Devolver el horario para esta línea aérea.
        *pBstrDetails = _bstr_t(_T("16:45:00 -Llegará
        al Terminal 3")).copy();
    }
    else
    {
        //Devolver un mensaje de error.
        return Error(LPCTSTR(_T("No disponible")),
            __uuidof(AirlineInfo),
            AIRLINE_NOT_FOUND);
    }
    return S_OK;
}

```

El método *GetAirlineTiming* lleva dos argumentos: el primero (*bstrAirline*) es una *BSTR* que representa la línea aérea, y el segundo (*pBstrDetails*) es un parámetro *de salida* que devuelve la información de llegada (hora local y puerta). Dentro del método, comprobamos que el valor del parámetro *bstrAirline* que llega es igual a «Air Scooby IC 5678». Si es así, devolvemos la información de llegada que hemos indicado en el código. Si el valor no es el que estamos esperando, invocamos un método de error para respaldar el hecho de que sólo prestamos servicio a una línea aérea.

Con esta visión básica del componente ya cerrada, echemos un vistazo a la generación de metadatos desde la biblioteca de tipos del componente (*typelib*) para que el cliente .NET pueda usar estos metadatos con el fin de interactuar con nuestro componente e invocar sus métodos.

Cómo generar metadatos desde una biblioteca de tipos COM

Una aplicación .NET que necesite hablar con nuestro componente COM no puede hacer uso directamente de la funcionalidad que muestra dicho componente. ¿Por qué no? Como vimos en el Capítulo 16, «Cómo obtener información sobre metadatos con Reflection», el entorno de ejecución .NET está diseñado para trabajar con componentes que tienen metadatos, mientras que COM está diseñado para trabajar por medio del registro de Windows y una serie de métodos de consulta que implementa el propio componente. Por lo tanto, la primera cosa que tenemos que hacer para permitir que este componente COM se utilice en el mundo .NET es generar metadatos para él. En el caso de un componente COM, esta capa de metadatos la utiliza el entorno de ejecución para determinar información de tipo. Esta información de tipo se utiliza en ese momento por el entorno de ejecución para crear lo que se llama un adaptador invocable en tiempo de ejecución (*runtime callable wrapper*=RCW) (véase Figura 17.1). El RCW maneja la activación real del objeto COM y maneja los requisitos de envío de información cuando la aplicación

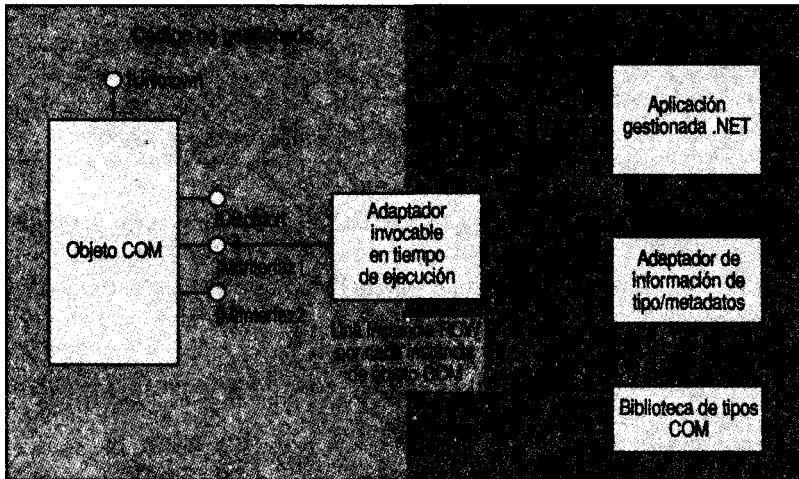


Figura 17.1. Los componentes básicos de la interoperabilidad COM-.NET.

.NET interactúa con él. El RCW también hace otra docena de cosas, tales como gestionar la identidad del objeto, sus ciclos de vida y la caché de la interfaz.

La gestión del tiempo de vida de un objeto es un aspecto crítico, porque el GC de .NET es capaz de mover los objetos de un sitio a otro y automáticamente se deshace de ellos cuando dejan de utilizarse. El RCW tiene como objetivo el darle a la aplicación .NET la noción de que está interactuando con un componente gestionado .NET, y le da al componente COM, en el espacio no gestionado, la impresión de que está siendo llamado por un cliente COM tradicional.

La creación y el comportamiento del RCW varía dependiendo de si estamos enlazando con el objeto COM en tiempo de compilación o en tiempo de ejecución. De manera transparente al usuario, el RCW está haciendo todo el trabajo sucio y haciendo los cálculos y las manipulaciones necesarias para traducir todos las invocaciones a método en llamadas que el componente COM, que vive en el mundo no gestionado, pueda entender. Básicamente, actúa como un embajador de buena voluntad entre el mundo gestionado y el mundo no gestionado, representado por *IUnknown*.

¡Basta de charla! Vamos a generar el recubrimiento de metadatos para nuestro componente COM AirlineInfo. Para hacerlo, necesitamos usar una herramienta llamada Importador de bibliotecas de tipo (Type Library Importer, *tlbimp.exe*). Esta utilidad viene con el SDK de .NET y se usa para leer una biblioteca de tipos (*typelib*) COM y para generar el recubrimiento de metadatos correspondiente para que el entorno de ejecución .NET pueda usarlo adecuadamente. Para hacer esto, necesitaremos instalar las aplicaciones de ejemplo que vienen en el CD que acompaña este libro y localizar el componente AirlineInfo.

Una vez que lo hayamos hecho, tecleamos lo siguiente en el símbolo del sistema:

```
TLBIMP AirlineInformation.tlb /out:AirlineMetadata.dll
```

Este comando indica al *TLBIMP* que lea la biblioteca de tipos COM de *AirlineInfo* y que genere un recubrimiento de metadatos llamado *AirlineMetadata.dll*. Si todo funciona como debiera, veríamos el siguiente mensaje:

```
TypeLib imported successfully to AirlineMetadata.dll
(Biblioteca de tipos importada de manera satisfactoria a AirlineMetadata.dll)
```

Así que, ¿qué tipo de información de tipo contienen estos metadatos generados, y cómo se presenta? Como chicos acostumbrados a usar el entorno COM, siempre hemos guardado como un tesoro nuestra querida utilidad *OleView.exe*, por —entre otras cosas— su habilidad para permitirnos echar un vistazo a los contenidos de una *biblioteca de tipos*. Afortunadamente, el SDK .NET viene con algo similar: el desensamblador de IL, llamado *ILDASM* —presentado en el Capítulo 2, «Introducción a Microsoft .NET»—, que nos permite ver los metadatos y el código MSIL que se ha generado para los ensamblajes que estamos gestionando. Como aprendimos en el Capítulo 16, todos los ensamblajes gestionados contienen metadatos autodescriptivos, e *ILDASM* es una herramienta muy útil cuando necesitamos profundizar en esos metadatos. Sigamos adelante y abramos *AirlineMetadata.dll* usando *ILDASM*. Deberíamos ver unos resultados similares a los mostrados en la Figura 17.2.

A partir de los metadatos generados, podemos ver que el método *GetAirlineTiming* está listado como un miembro público de la clase *AirlineInfo*. Hay también un constructor generado para la clase *AirlineInfo*. Observemos que los parámetros del método se han sustituido automáticamente por sus equivalentes .NET. En este ejemplo, el *BSTR* se ha

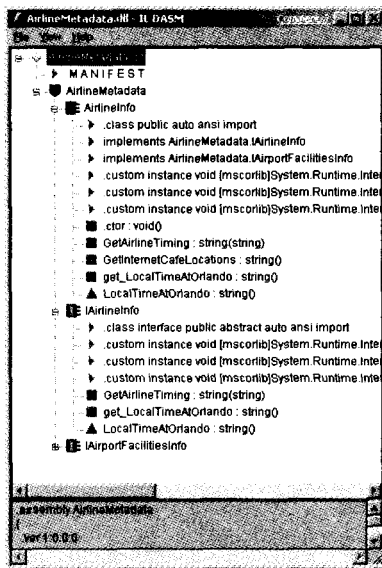


Figura 17.2. *ILDASM* es una fantástica herramienta para ver metadatos y MSIL para ensamblajes gestionados.

sustituido por el parámetro *System.String*. Fijémonos también que el parámetro que estaba marcado *[out, retval]* en el método *GetAirlineTiming* se convirtió al valor real que devuelve el método (devuelto como *System.String*). Adicionalmente, cualquiera de los valores erróneos *HRESULT* que devuelve el objeto COM —en caso de un error o un fallo en la lógica de negocio— se tratan como excepciones.

Enlace en tiempo de compilación a componentes COM

Ahora que hemos generado los metadatos requeridos por el cliente .NET, intentemos invocar el método *GetAirlineTiming* de nuestro objeto COM desde el cliente .NET. Aquí tenemos una aplicación cliente que crea el objeto COM usando los metadatos que generamos anteriormente y que invoca el método *GetAirlineTiming*. Observemos que en este ejemplo estamos usando enlace en tiempo de compilación. En breve, veremos dos ejemplos más que ilustrarán el descubrimiento dinámico de tipos y enlace en tiempo de ejecución.

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient1App
{
    public static void Main()
    {
        //////////////////////////////////////
        /// EJEMPLO DE ENLACE EN TIEMPO DE COMPILACIÓN
        //////////////////////////////////////
        String strAirline = "Air Scooby IC 5678";
        String strFoodJunkieAirline = "Air Jughead TX 1234";
        try
        {
            AirlineInfo objAirlineInfo;

            //Creamos un nuevo objeto AirlineInfo.
            objAirlineInfo = new AirlineInfo();

            //Mostramos el resultado de la llamada al
            //método GetAirlineTiming.
            Console.WriteLine("Detalles de la Aerolínea {0} --> {1}",
                strAirline, objAirlineInfo.GetAirlineTiming(strAirline));

            //ERROR: ;Lo siguiente dará como resultado el lanzamiento
            //de una excepción!
            //Console.WriteLine("Detalles de la Aerolínea {0} --> {1}",
            //strFoodJunkieAirline, objAirlineInfo.GetAirlineTiming
            //(strFoodJunkieAirline));
        }
        catch (COMException e)
        {
        }
    }
}
```

```

        Console.WriteLine("Vaya -Nos hemos encontrado un error" +
            "para la Aerolínea {0}. El mensaje de error" +
            "es: {1}. El código de error es {2}",
            strFoodJunkieAirline,
            e.Message, e.ErrorCode);
    }
}
}

```

Lo que está ocurriendo aquí es que el entorno de ejecución está fabricando un *RCW* que hace corresponder los metadatos de métodos y campos de las clases con los métodos y propiedades existentes en la interfaz que implementa el objeto COM. Se crea una instancia *RCW* para cada instancia del objeto COM. El entorno de ejecución .NET sólo se preocupa de gestionar el tiempo de vida del *RCW* y hacer la recogida de basura de dichos *RCW*. Es el *RCW* el que se preocupa de mantener el conteo de referencias del objeto COM al que corresponde, y de ese modo protege al entorno .NET de gestionar el conteo de referencias en el objeto COM real. Como podemos ver en la Figura 17.2, los metadatos de *AirlineInfo* se definen bajo un espacio de nombres denominado *AIRLINEINFORMATIONLib*. El cliente .NET ve todos los métodos de la interfaz como si fueran miembros de la clase *AirlineInfo*. Todo lo que necesitamos es crear una instancia de la clase *AirlineInfo* usando el operador *new* e invocar los métodos públicos del objeto creado. Cuando el método es invocado, el *RCW* se ocupa de traducir la llamada a la correspondiente llamada al método COM. EL *RCW* también maneja todos los aspectos relativos al envío de datos y gestión del tiempo de vida del objeto. ¡Para el cliente .NET, nada parece diferente a crear un típico objeto gestionado e invocar a uno de sus miembros de clase públicos!

Observemos que en el momento en que el método COM produzca un error, este error COM será capturado por el *RCW*. Este error se convertirá en una clase *COMException* equivalente (que podemos encontrar en el espacio de nombres *System.Runtime.InteropServices*). Por supuesto, el objeto COM todavía necesita implementar la interfaz *ISupportErrorInfo* para que funcione la propagación de este error y para que de esta manera el *RCW* sepa que nuestro objeto proporciona información detallada sobre el error. El error puede ser capturado por el cliente .NET con el mecanismo de gestión de excepciones usual *try-catch*, y el cliente tiene acceso al número de error, descripción, origen de la excepción, y otros detalles que hubieran estado disponibles para cualquier otro cliente con capacidades COM. Ahora vamos a llevar este ejemplo un poco más allá y ver otras maneras de enlazar con componentes COM.

Cómo utilizar descubrimiento dinámico de tipos para seleccionar interfaces COM

Entonces, ¿cómo funciona el escenario tradicional *QueryInterface* (QI) desde la perspectiva del cliente .NET cuando quiere acceder a otra interfaz implementada por el objeto COM? Para usar *QI* para obtener otra interfaz, todo lo que necesitamos hacer es convertir el tipo del objeto actual a la otra interfaz que necesitamos, y aquí está, ¡nuestro *QI* está

hecho! Ahora estamos listos para invocar todos los métodos y propiedades de la interfaz deseada. Así de simple.

De nuevo, es el RCW el que hace todo el trabajo duro internamente. De algún modo, es análogo al modo en que el entorno de ejecución de Visual Basic evita a los programadores de aplicaciones cliente el tener que escribir cualquier código explícito relativo a *QueryInterface* —simplemente, hace el QI para nosotros cuando fijamos un tipo de objeto a un objeto de otro tipo asociado.

Veamos esto en acción para darnos cuenta de lo sencillo que es. En nuestro ejemplo, supongamos que queremos llamar a los métodos existentes en la interfaz *IAirportFacilities*, que es otra interfaz implementada por nuestro objeto COM. Para hacerlo, convertiríamos el tipo del objeto *AirlineInfo* a la interfaz *IAirportFacilities*. Ahora ya podemos llamar a todos los métodos que forman parte de la interfaz *IAirportFacilities*. Pero antes de realizar la conversión de tipo, posiblemente quisiéramos comprobar si la instancia del objeto que estamos usando en ese instante soporta o implementa el tipo de interfaz por la que estamos preguntando. Podemos hacerlo usando el método *IsInstanceOf* de la clase *System.Type*. Si el resultado es *true* (verdadero), sabemos que el QI ha tenido éxito y que podemos realizar sin problemas la conversión de tipo. En caso de que usáramos la conversión de tipo con el objeto sobre una interfaz arbitraria que el objeto no soportara, se lanza una excepción de tipo *System.InvalidCastException*. De este modo, el RCW se asegura de que solamente aplicamos la conversión de tipo a interfaces que implementa el objeto COM. Así es como se presenta en código:

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient2App
{
    public static void Main()
    {
        //////////////////////////////////////
        /// QUERY INTERFACE/ RT Chequeo de tipos
        //////////////////////////////////////
        try
        {
            AirlineInfo objAirlineInfo;
            IAirportFacilitiesInfo objFacilitiesInfo;

            //Crear un nuevo objeto AirlineInfo.
            objAirlineInfo = new AirlineInfo();

            //Invocar el método GetAirlineTiming.
            String strDetails = objAirlineInfo.GetAirlineTiming
                                   (strAirline);

            //QI en busca de la interfaz IAirportFacilitiesInfo.
            objFacilitiesInfo =
                (IAirportFacilitiesInfo)objAirlineInfo;

            //Invoca un método en la interfaz IAirportFacilitiesInfo
```

```

        Console.WriteLine("{0}",
            objFacilitiesInfo.GetInternetCafeLocations());
    }
    catch(InvalidCastException eCast)
    {
        Console.WriteLine("Tenemos una excepción de conversión" +
            "de tipo Inválida -El mensaje es"
            "{0}", eCast.Message);
    }
}
}

```

Enlace en tiempo de ejecución a componentes COM

Los dos ejemplos que hemos visto hasta ahora —*AirlineClient1App* y *AirlineClient2App*— usan los metadatos del RCW para enlazar en tiempo de compilación el cliente .NET con el objeto COM. Aunque el enlace en tiempo de compilación proporciona un auténtico aluvión de beneficios —como un chequeo fuerte de tipos en tiempo de compilación, capacidades de autocompletarse a partir de la información de tipos en herramientas de desarrollo (como Visual Studio .NET) y, por supuesto, mejor rendimiento—, puede haber ocasiones en que no tenemos los metadatos en tiempo de compilación para el objeto COM que estamos enlazando, y por tanto necesitamos hacer un *enlace en tiempo de ejecución* al componente. Por ejemplo, si el componente que estamos intentando utilizar contiene únicamente una interfaz *dispinterface*, estamos bastante limitados y en cierto modo obligados a usar enlace en tiempo de ejecución para utilizar el componente.

Podemos conseguir enlace en tiempo de ejecución a un objeto COM por medio del mecanismo ‘reflection’ que aprendimos en el Capítulo 16. Para enlazar de esta manera con un componente COM, necesitaremos saber el *ProgID* del componente. Esto se debe a que el método estático *CreateInstance* de la clase *System.Activator* requiere un objeto *Type*. Sin embargo, usando el *ProgID* del componente, podemos invocar al método *GetTypeFromProgID* de la clase *System.Type*. Esto devolverá un objeto *Type* válido para .NET que podemos usar entonces en la llamada al método *System.Activator.CreateInstance*. Una vez hecho, podemos invocar cualquiera de los métodos o propiedades soportados por la interfaz predeterminada del componente usando el método instanciado *System.Type.InvokeMember* del objeto *Type* que obtuvimos de *GetTypeFromProgID*.

Lo único que tenemos que saber es el nombre del método o la propiedad y la información de parámetros que acepta la llamada al método. Cuando llamamos a un método de un componente enlazado en tiempo de ejecución, la manera en la que pasamos parámetros es agrupándolos en un array genérico *System.Object* y pasándoselo al método. Tenemos también que fijar los indicadores de enlace (*binding flags*) adecuados, dependiendo de si estamos invocando un método o leyendo/asignando el valor de una propiedad.

Como podemos ver en el siguiente código, es un poco más de trabajo que con el enlace en tiempo de compilación. Sin embargo, en casos en los que el enlace en tiempo de ejecución es la única opción, nos sentiremos muy contentos de tenerlo a nuestra disposición.

```

using System;
using System.Runtime.InteropServices;
using System.Reflection;
using AIRLINEINFORMATIONLib;

public class AirlineClient3App
{
    public static void Main()
    {
        //////////////////////////////////////
        /// ENLACE EN TIEMPO DE EJECUCIÓN
        //////////////////////////////////////
        try
        {
            object objAirlineLateBound;
            Type objTypeAirline;

            object[] arrayInputParams = {"Air Scooby IC 5678"};

            objTypeAirline = Type.GetTypeFromProgID
                ("AirlineInformation.AirlineInfo");

            objAirlineLateBound = Activator.CreateInstance
                (objTypeAirline);

            String str = (String)objTypeAirline.InvokeMember
                ("GetAirlineTiming",
                BindingFlags.Default |
                BindingFlags.InvokeMethod,
                null, objAirlineLateBound,
                arrayInputParams);

            Console.WriteLine("{0}", str);

            String strTime = (String)objTypeAirline.InvokeMember
                ("LocalTimeAtOrlando",
                BindingFlags.Default |
                BindingFlags.GetProperty,
                null, objAirlineLateBound,
                new object []{});

            Console.WriteLine (";Hola! La hora local en" +
                "Orlando, Florida es: {0}", strTime);
        }
        catch(COMException e)
        {
            Console.WriteLine("Vaya -Hemos encontrado un error" +
                " para la línea aérea {0}."
                " El mensaje de error" +
                " es: {1}. El código de error es {2}",
                strFoodJunkieAirline,
                e.Message,e.ErrorCode);
        }
    }
}

```

Modelos de proceso COM

La mayor parte de la gente que comienza a programar en COM tiene poco o ningún conocimiento de los modelos de subprocesos y apartamentos COM. No es hasta que se convierten en profesionales mucho más experimentados cuando se dan cuenta de que el modelo de subproceso libre que han estado usando tiene un coste de rendimiento importante cuando un proceso cliente que usa un apartamento de un solo subproceso (*single-threaded apartment=STA*) se utiliza para crear un objeto de apartamento de multiproceso (*multithreaded apartment=MTA*). Aparte de esto, los programadores nuevos en COM, frecuentemente no se dan cuenta de la seguridad de subprocesos y del peligro inminente que les acecha cuando subprocesos concurrentes acceden a sus componentes COM.

Antes de que un subproceso pueda invocar un objeto COM, tiene que declarar su pertenencia a un tipo de apartamento concreto, declarando si entrará en un *STA* o un *MTA*. Los subprocesos de cliente *STA* llaman a *CoInitialize(NULL)* o *CoInitializeEx(0, COINIT_APARTMENTTHREADED)* para entrar en un *STA*, y los subprocesos *MTA* llaman a *CoInitializeEx(0, COINIT_MULTITHREADED)* para entrar en un *MTA*. De igual manera, en el mundo gestionado por .NET, tenemos la opción de permitir al subproceso invocador en el espacio gestionado el declarar sus afinidades con respecto a apartamentos. Por defecto, el subproceso en una aplicación gestionada elige vivir en un *MTA*. Es como si el subproceso que invoca se inicializara a sí mismo con *CoInitializeEx(0, COINIT_MULTITHREADED)*. Pero pensemos en el desperdicio de recursos y las penalizaciones de rendimiento en las que incurriríamos si estuviéramos llamando a un componente clásico COM *STA* que está diseñado para ser gestionado en un subproceso dentro de un apartamento. Esos apartamentos incompatibles incurrirían en el derroche de un par adicional de conectores de código auxiliar (*proxy/stub*), y ciertamente esto redundaría en una penalización en el rendimiento.

Para evitarlo, podemos ignorar la elección de apartamento por defecto para un hilo de ejecución gestionado en una aplicación .NET por medio del uso de la propiedad *ApartmentState* de la clase *System.Threading.Thread*. La propiedad *ApartmentState* toma uno de los siguientes valores enumerados:

- **MTA**. Apartamento de múltiples subprocesos.
- **STA**. Apartamento de un solo subproceso.
- **Unknown**. Equivalente al comportamiento MTA por defecto.

También necesitaremos especificar la propiedad *ApartmentState* para el subproceso invocador antes de hacer ninguna llamada al objeto COM. Debemos darnos cuenta de que no es posible cambiar el *ApartmentState* (el apartamento al que asignamos el objeto) una vez que se ha creado el objeto COM. Por lo tanto, tiene sentido el fijar el *ApartmentState* del subproceso tan pronto como sea posible en nuestro código. Este código muestra cómo hacerlo:

```
//Fija el ApartmentState del subproceso cliente para entrar en un STA.
Thread.CurrentThread.ApartmentState = ApartmentState.STA;
```

```
//Creamos nuestro objeto COM a través del Interop.
MySTA objSTA = new MySTA();
objSTA.MyMethod()
```

RESUMEN

La última información con la que nos vamos a quedar es con cómo encajan todos esos distintos mecanismos para trabajar con código heredado —*PInvoke*, código inseguro e interoperabilidad COM— dentro del esquema global de .NET. En este capítulo hemos aprendido lo siguiente:

- Con respecto al uso de llamadas a función estilo C, aprendimos cómo usar *PInvoke* junto con varios atributos para hacer que las tareas de adaptar diferentes tipos de datos, incluyendo datos personalizados, sean más sencillas.
- Con respecto al código inseguro, aprendimos cómo renunciar a los beneficios del código gestionado dentro de una aplicación C# en situaciones donde necesitemos tener más control sobre lo que está ocurriendo. Estos escenarios pueden incluir ocasiones en las que necesitemos manipular manualmente memoria por razones de eficiencia o cuando estamos moviendo bloques de código en una aplicación C# y simplemente no estamos listos todavía para convertir ese código en código gestionado.
- Con respecto a COM, vimos cómo exponer componentes COM clásicos a las aplicaciones .NET y cómo la interoperabilidad COM nos permite, de manera sencilla, reutilizar componentes COM desde código gestionado. Una vez visto esto, vimos varias maneras de invocar nuestros componentes COM usando tanto enlace en tiempo de compilación como enlace en tiempo de ejecución junto con distintas maneras de hacer comprobación de tipos en tiempo de ejecución. Finalmente, vimos cómo declarar los subprocesos gestionados su afiliación a apartamentos concretos cuando invocan componentes COM.

Una vez llegados a este punto, como desarrolladores que utilizan estos mecanismos para manejar código no gestionado, puede que nos estemos preguntando si tiene sentido continuar usando estas metodologías o si deberíamos liarnos la manta a la cabeza y realizar la transición directamente al mundo .NET mediante la reescritura de todos los componentes y código que contiene la lógica de negocio usando un lenguaje .NET como C#. Digamos que la respuesta depende de la situación en la que nos encontremos.

Si tenemos montañas de código obsoleto —tanto si son funciones tipo C en DLL, código que manipula directamente memoria, componentes COM, o una combinación de los tres—, el hecho es que probablemente no seamos capaces de convertir todo nuestro código de la noche a la mañana. En este caso, tiene sentido el hacer uso de todos estos mecanismos .NET para trabajar con código obsoleto. Sin embargo, si estamos escribiendo nuevo código para soportar lógica de negocio desde cero, recomendamos fervientemente el escribir nuestro código como componentes gestionados mediante la utilización de un lenguaje como C#. De este modo, podemos salvar la penalización de rendimiento en la que inevitablemente incurriremos mientras estamos en el período de transición entre lo gestionado y lo no gestionado.

Capítulo 18

Cómo trabajar con ensamblajes

Este capítulo describe las principales ventajas de utilizar ensamblajes, incluyendo la posibilidad de «paquetizar»* (*packaging*) y gestionar versiones de nuestros componentes .NET. Veremos asimismo cómo crear ensamblajes de un único o varios archivos usando la herramienta de Generación de ensamblajes (*Assembly Generation Tool=al.exe*), cómo crear ensamblajes compartidos usando la herramienta de Nombres únicos (*Strong Name Too=sn.exe*), cómo explorar la caché global de ensamblajes usando la extensión del intérprete de comandos Visor de caché de ensamblaje (*Assembly Cache Viewer=shfusion.dll*) y cómo manipular la caché del ensamblaje con la herramienta de gestión de la Caché global de ensamblaje (*Global Assembly Cache=gacutil.exe*). Finalmente, pasaremos por varios ejemplos y veremos de qué trata la gestión de versiones y cómo los ensamblajes y las políticas de gestión de versiones .NET nos ayudan a evitar el «infierno de las DLL».

UN VISTAZO A LOS ENSAMBLAJES

El Capítulo 16, «Cómo obtener información sobre metadatos con Reflection», describía los ensamblajes como archivos físicos que constan de uno o más archivos ejecutables portables (PE) generados por un compilador .NET. En el contexto de ese capítulo, esa definición era aceptable. Sin embargo, los ensamblajes son algo más complicados. Aquí tenemos una definición más completa: un ensamblaje es el resultado de empaquetar un manifiesto, uno o más módulos, y opcionalmente, uno o más recursos. La utilización de ensamblajes permite agrupar unidades funcionales en un único archivo para propósitos de despliegue, gestión de versiones y mantenimiento.

Todos los archivos PE que se usan en el entorno de ejecución .NET constan de un ensamblaje o un grupo de ensamblajes. Cuando compilamos una aplicación usando el compilador C#, realmente estamos creando un ensamblaje. Podemos no darnos cuenta de

* Se utilizará el término «paquetizar» para no confundir con empaquetado y desempaquetado (*boxing/unboxing*) (N. de los t.).

ello inicialmente, a menos que específicamente estemos intentando poner múltiples módulos en un único ensamblaje o aprovechándonos de las ventajas que presentan algunos aspectos específicos de los ensamblajes, como es la gestión de versiones. Sin embargo, es importante darse cuenta de que en cualquier momento que creamos un EXE o una DLL (usando el modificador */t:library*), estamos creando un ensamblaje con un manifiesto que describe el ensamblaje al entorno de ejecución .NET. De manera adicional, podemos crear un módulo (usando el modificador */t:module*) que es realmente una DLL (con una extensión *.netmodule*) sin un manifiesto. En otras palabras, aunque lógicamente es todavía una DLL, no pertenece a un ensamblaje, y se debe añadir a éste bien mediante el uso del modificador */addmodule* al compilar una aplicación o usando la herramienta de Generación de ensamblajes. Veremos cómo hacerlo posteriormente en la sección «Cómo construir ensamblajes».

Datos del manifiesto

El manifiesto de un ensamblaje se puede almacenar de varias maneras. Si fuéramos a compilar una aplicación o una DLL aislada, el manifiesto se incorporaría al PE resultante. Esto se conoce como un *ensamblaje de archivo único*. Un *ensamblaje de varios archivos* se puede generar también, con el manifiesto existiendo bien como una entidad aislada dentro del ensamblaje o como un anexo a uno de los módulos dentro del ensamblaje.

La definición de un ensamblaje también depende de una manera muy importante de cómo lo usemos. Desde una perspectiva de cliente, un ensamblaje es una colección de módulos con nombre y versiones de módulos, tipos exportados y, opcionalmente, recursos. Desde el punto de vista del creador del ensamblaje, un ensamblaje es un medio de empaquetar módulos relacionados entre sí, tipos y recursos, exportando sólo lo que debería utilizar un cliente. Una vez dicho esto, es el manifiesto el que proporciona el nivel de indirección entre los detalles de implementación del ensamblaje y lo que se supone que debe usar el cliente. Aquí tenemos una descomposición de la información almacenada en el manifiesto de un ensamblaje:

- **Nombre del ensamblaje.** El nombre textual del ensamblaje.
- **Información de gestión de versiones.** Esta cadena contiene cuatro partes diferenciadas que constituyen un número de versión. Incluyen un número principal y secundario de versión, así como un número de revisión y de construcción.
- **Un nombre compartido (opcional) y un certificado criptográfico del ensamblaje (*signed assembly hash*).** Esta información se aplica al despliegue de ensamblajes y se ve en más detalle en «Despliegue de ensamblajes», en un apartado posterior de este capítulo.
- **Archivos.** Esta lista incluye todos los archivos que existen en el ensamblaje.
- **Ensamblajes referenciados.** Esta es una lista de todos los ensamblajes externos que están referenciados directamente desde el ensamblaje del manifiesto.
- **Tipos.** Esta es la lista de todos los tipos existentes en el ensamblaje con una correspondencia con el módulo que contiene el tipo. Estos datos son los que permiten al ejemplo de 'reflection' que vimos en el Capítulo 16 (que itera a través de todos los tipos del ensamblaje) ejecutarse tan rápidamente.

- **Permisos de seguridad.** Es una lista de permisos de seguridad que son denegados explícitamente por el ensamblaje.
- **Atributos personalizados.** El Capítulo 8, «Atributos», describía cómo crear nuestros propios atributos personalizados. De manera similar a los tipos, los atributos personalizados se almacenan en el manifiesto del ensamblaje para acceder a ellos cuando usamos 'reflection'.
- **Información del producto.** Esta información incluye Compañía, Marca registrada, Producto y Copyright.

BENEFICIOS DE LOS ENSAMBLAJES

Los ensamblajes permiten al desarrollador múltiples ventajas, como paquetización, despliegue y gestión de versiones.

Paquetización de ensamblajes

Una ventaja de la posibilidad de paquetizar múltiples módulos en un único archivo físico es la mejora del rendimiento. Cuando creamos una aplicación y la desplegamos usando un ensamblaje multiarchivo, el entorno de ejecución .NET tan sólo necesita cargar los módulos requeridos. Esto tiene el efecto inmediato de reducir la carga de trabajo de la aplicación.

Despliegue de ensamblajes

La unidad de desarrollo más pequeña en .NET es el ensamblaje. Como vimos anteriormente, podemos crear un módulo .NET con el modificador */t:module*, pero debemos incluir dicho módulo en un ensamblaje si queremos proceder a su despliegue. Además, aunque es tentador decir que los ensamblajes son un medio de despliegue de aplicaciones, técnicamente, no es verdad. Es más preciso el ver los ensamblajes de .NET como una manera de hacer *despliegue de clases* (muy parecido a lo que hace una DLL para Win32), en el que una única aplicación puede estar compuesta de muchos ensamblajes.

Comoquiera que los ensamblajes son autodescriptivos, la manera más sencilla de desplegarlos es copiando el ensamblaje en la carpeta destino deseada. En el momento en que intentemos ejecutar una aplicación contenida en el ensamblaje, el manifiesto dará las instrucciones necesarias al entorno de ejecución .NET en lo referente a los módulos que contiene el ensamblaje. De manera adicional, el ensamblaje también contiene referencias a cualquier ensamblaje externo que sea necesario para la aplicación.

La manera más habitual de proceder al despliegue es por medio de ensamblajes privados; esto es, ensamblajes que son copiados a una carpeta y que no son compartidos. ¿Cómo podemos especificar un ensamblaje privado? Esta es la situación por defecto, y ocurre de manera automática, a menos que hagamos de manera explícita un *ensamblaje compartido*. Compartir ensamblajes lleva un poco más de trabajo y se verá más tarde en la sección «Cómo crear ensamblajes compartidos».

Versiones de ensamblajes

Otra fantástica ventaja de usar ensamblajes es que incorporan gestión de versiones; más en concreto, el final del «infierno de las DLL». El «infierno de las DLL» se aplica a la situación en la que una aplicación sobrescribe una DLL necesaria para otra aplicación, normalmente con una versión más antigua de la misma DLL, inutilizando la primera aplicación. Aunque el formato de archivo de recursos de Win32 permite una especie de gestión de versiones de recursos, el sistema operativo no fuerza ningún tipo de reglas de gestión de versiones de modo que garantice que las aplicaciones sigan funcionando. Esta es una responsabilidad exclusiva de los programadores de las aplicaciones.

Como solución a este problema, el manifiesto incluye información de gestión de versiones para el ensamblaje, así como una lista de todos los ensamblajes referenciados y la información de versión de todos ellos. Debido a la existencia de esta arquitectura, el entorno de ejecución .NET puede asegurarse de que las políticas de gestión de versiones se llevan a rajatabla y que las aplicaciones continuarán funcionando incluso cuando se instalen versiones en el sistema más nuevas e incompatibles de DLL compartidas. Debido a que la gestión de versiones es uno de los mayores beneficios de los ensamblajes, lo analizaremos en detalle, incluyendo varios ejemplos, en «Cómo gestionar versiones de ensamblajes».

CÓMO CONSTRUIR ENSAMBLAJES

Si creamos una DLL con el modificador */t:library*, no seremos capaces de añadirlo a otro ensamblaje. La razón de esto es que el compilador ha generado de manera automática un manifiesto para la DLL, y por lo tanto la propia DLL es un ensamblaje. Para verlo en vivo, observemos el siguiente ejemplo. Tenemos una DLL (Module1Server.cs) que tiene un tipo de ejemplo llamado *Module1Server*.

```
//Module1Server.cs
//Construir con los siguientes modificadores de línea de comandos
//    csc /t:library Module1Server.cs
public class Module1Server
{
}
```

Esta DLL es referenciada por el código cliente (Module1Client.cs):

```
//Module1ClientApp.cs
//Construir con los siguientes modificadores de línea de comandos
//    csc Module1ClientApp.cs /r:Module1Server.dll
using System;
using System.Diagnostics;
using System.Reflection;

class Module1ClientApp
{
    public static void Main()
```

```

{
    Assembly DLLAssembly = Assembly.GetAssembly(typeof(Module1Server));
    Console.WriteLine("Información de Ensamblaje en Module1Server.dll");
    Console.WriteLine("\t" + DLLAssembly);

    Process p = Process.GetCurrentProcess();
    string AssemblyName = p.ProcessName + ".exe";
    Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
    Console.WriteLine("Información de Ensamblaje en Module1Client.exe");
    Console.WriteLine("\t" + ThisAssembly);
}
}

```

Hemos construido estos dos módulos haciendo uso de los siguientes modificadores:

```

csc /t:library Module1Server.cs
csc Module1ClientApp.cs /r:Module1Server.dll

```

Una vez llegados a este punto y ejecutando el código, obtenemos los siguientes resultados y probamos que tanto el EXE como la DLL existen en ensamblajes diferentes:

```

Información de Ensamblaje en Module1Server.dll
Module1Server, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Información de Ensamblaje en Module1Client.exe
Module1Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

```

De hecho, si tuviéramos que cambiar el modificador de acceso de la clase *Module1Server* de *public* a *internal*, el código de cliente no se compilaría, porque, por definición, el modificador de acceso *internal* especifica que el tipo que está siendo modificado sólo es accesible a otro código en el mismo ensamblaje.

Cómo crear ensamblajes con módulos múltiples

Podemos colocar ambos módulos de nuestro ejemplo en el mismo ensamblaje de dos maneras distintas. La primera es cambiar los modificadores que hemos usado con el compilador. Aquí tenemos un ejemplo:

```

//Module2Server.cs
//Construir con los siguientes modificadores de línea de comando
//    csc /t:module Module2Server.cs
internal class Module2Server
{
}

```

Démonos cuenta de que ahora podemos usar el modificador de acceso interno de manera que la clase sea sólo accesible al código existente dentro del ensamblaje.

```

//Module2ClientApp.cs
//Construir con los siguientes modificadores de línea de comando
//    csc /addmodule:Module2Server.netmodule Module2ClientApp.cs

```

```

using System;
using System.Diagnostics;
using System.Reflection;

class Module2ClientApp
{
    public static void Main()
    {
        Assembly DLLAssembly =
            Assembly.GetAssembly(typeof(Module2Server));
        Console.WriteLine("Información de Ensamblaje en Module2Server.dll");
        Console.WriteLine("\t" + DLLAssembly);

        Process p = Process.GetCurrentProcess();
        string AssemblyName = p.ProcessName + ".exe";
        Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
        Console.WriteLine("Información de Ensamblaje en Module2Client.dll");
        Console.WriteLine("\t" + ThisAssembly);
    }
}

```

Démonos cuenta de cómo están contruidos Module2Server.cs y Module2Client.exe:

```

csc /t:module Module2Server.cs
csc /addmodule:Module2Server.netmodule Module2Client.cs

```

Primero deberemos eliminar el modificador */r*, porque dicho modificador se usa únicamente para referenciar ensamblajes, y ahora ambos módulos residirán en el mismo ensamblaje. Entonces añadiremos el modificador */addmodule*, que se usa para decirle al compilador qué módulos se van a añadir al ensamblaje que se está creando.

Construir y ejecutar la aplicación produce ahora los resultados siguientes:

```

Información de Ensamblaje en Module2Server.dll
  Module2Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
Información de Ensamblaje en Module2Client.dll
  Module2Client, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null

```

Otro modo de crear un ensamblaje es con la herramienta para la Generación de ensamblajes (*Assembly Generation tool*). Esta herramienta usará como parámetros de entrada uno o más archivos, que o son módulos .NET (conteniendo MSIL), o archivos de recursos o archivos de imágenes. El resultado es un archivo con un manifiesto de un ensamblaje. Por ejemplo, usaríamos la herramienta de Generación de ensamblajes si tuviéramos varias DLL y quisiéramos distribuir las y gestionar su versión como una única unidad. Asumiendo que nuestras DLL se llamaran A.DLL, B.DLL y C.DLL, usaríamos la aplicación *al.exe* para crear el ensamblaje compuesto de la manera siguiente:

```

al /out:COMPOSITE.DLL A.DLL B.DLL C.DLL

```

CÓMO CREAR ENSAMBLAJES COMPARTIDOS

Compartir ensamblajes es algo que habitualmente se hace cuando un ensamblaje lo utilizarán múltiples aplicaciones y la gestión de versiones es importante. (Entraremos en más detalle en la gestión de versiones en la próxima sección). Para compartir ensamblajes, debemos crear un *nombre compartido* —también conocido como un *nombre fuerte* o *único* (*strong name*)— para el ensamblaje usando la herramienta para Nombres únicos (*Strong Name tool*), que viene con el SDK .NET. Los cuatro principales beneficios derivados del uso de nombres únicos son los siguientes:

- Es el mecanismo que utiliza .NET para generar un nombre global unívoco.
- Comoquiera que la pareja de claves generadas (lo explicamos en breve) incluye una firma digital, podemos detectar si ha sido manipulado desde su creación original.
- Los nombres únicos garantizan que un grupo de terceros no puede generar una versión superior de un ensamblaje que hemos construido. De nuevo, esto es por la firma digital —el grupo de terceros no tiene nuestra clave privada.
- Cuando .NET carga un ensamblaje, el entorno de ejecución puede verificar que el ensamblaje viene del autor que la persona que lo invoca está esperando.

El primer paso para crear un nombre único es usar la herramienta para Nombres únicos para generar un archivo de claves para el ensamblaje. Lo hacemos por medio del modificador *-k* con el nombre del archivo de salida que contendrá la clave. Aquí vamos a inventarnos algo para probar —*InsideCSharp.key*— y crear un archivo de esta manera:

```
sn -k InsideCSharp.key
```

Una vez ejecutado, obtendríamos un mensaje de confirmación como este:

```
Key pair written to InsideCSharp.key
(Pareja de claves escrita en el archivo InsideCSharp.key)
```

Ahora añadimos el atributo *assembly:AssemblyKeyFile* al archivo fuente. Aquí hemos creado otro conjunto de archivos de ejemplo para ilustrar cómo se hace:

```
//Module3Server.cs
//Construir con los siguientes modificadores de línea de comando
//    csc /t:module Module3Server.cs
internal class Module3Server
{
}

//Module3ClientApp.cs
//Construir con los siguientes modificadores de línea de comando
//    csc /addmodule:Module3Server.netmodule Module3ClientApp.cs
using System;
using System.Diagnostics;
using System.Reflection;
```

```
[assembly:AssemblyKeyFile("InsideCSharp.key")]

class Module3ClientApp
{
    public static void Main()
    {
        Assembly DLLAssembly =
            Assembly.GetAssembly(typeof(Module3Server));
        Console.WriteLine("Información de Ensamblaje en Module3Server.dll");
        Console.WriteLine("\t" + DLLAssembly);

        Process p = Process.GetCurrentProcess();
        string AssemblyName = p.ProcessName + ".exe";
        Assembly ThisAssembly = Assembly.LoadFrom(AssemblyName);
        Console.WriteLine("Información de Ensamblaje en Module3Client.dll");
        Console.WriteLine("\t" + ThisAssembly);
    }
}
```

Como podemos ver, el constructor del atributo *assembly:AssemblyKeyFile* lleva el nombre del archivo de claves que se había generado con la utilidad para Nombres únicos y es la manera por la cual especificamos que una pareja de claves se va a utilizar para darle un nombre único a nuestro ensamblaje. Un punto adicional importante que hay que entender es que este atributo es un atributo al nivel de ensamblaje. Por lo tanto, técnicamente, puede ser colocado en cualquier archivo en el ensamblaje y no está asociado a una clase específica. Sin embargo, por regla general, colocamos este atributo justo debajo de las instrucciones *using* y antes de cualquier definición de clases. Ahora, cuando ejecutemos la aplicación, tomamos nota del valor *PublicKeyToken* del ensamblaje. Este valor era *nulo* (*null*) en los dos ejemplos anteriores, porque dichos ensamblajes se consideraban privados. Sin embargo, ahora el ensamblaje se ha definido como un ensamblaje compartido, así que el ensamblaje tiene una clave pública asociada.

```
Información de Ensamblaje en Module3Server.dll
Module3Client, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=6ed7cef0c0065911
Información de Ensamblaje en Module3Client.dll
Module3Client, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=6ed7cef0c0065911
```

De acuerdo con el objeto *Assembly* que hemos instanciado para este ensamblaje ejemplo, es compartido. Sin embargo, ¿cómo sabemos qué ensamblajes de nuestro sistema .NET son compartidos? La respuesta está en la caché global de ensamblajes (*global assembly cache*). En la próxima sección analizaremos esta parte de .NET y explicaremos el rol que juega en los ensamblajes compartidos.

CÓMO TRABAJAR CON LA CACHÉ GLOBAL DE ENSAMBLAJES

Todas las instalaciones .NET tienen una caché de código denominada la *caché global de ensamblajes*. Esta área sirve a tres propósitos principales:

- Se usa para almacenar código bajado de Internet o desde otros servidores (tanto servidores http como servidores de archivos). Observemos que el código bajado para una aplicación concreta está almacenado en la porción privada de la caché; esto evita que sea accedido por otros.
- Es un almacén de datos para componentes compartidos por múltiples aplicaciones .NET. Los ensamblajes que están instalados en la caché por medio de la herramienta para la caché global de ensamblajes se almacenan en la porción global de la caché, y por ello son accesibles por todas las aplicaciones existentes en la máquina.
- Una cuestión que oímos frecuentemente es: «¿Dónde se almacena el código 'jitted' (compilado a la plataforma nativa en tiempo de ejecución) de tal modo que nuestro código C# sea sólo 'jitted' la primera vez que se ejecuta? Ahora sabemos la respuesta: las versiones en código nativo de los ensamblajes que han sido 'jitted' con anterioridad se almacenan en la caché.

Cómo examinar la caché

Echemos un vistazo a la caché para ver los ensamblajes que están ahora mismo instalados y compartidos. Usando el Explorador de Microsoft, abrimos la carpeta `c:\winnt\assembly`. Para ayudarnos con la visualización de información asociada a los ensamblajes, .NET tiene una extensión a su intérprete de comandos (*shell*) llamada Visor de la caché de ensamblajes —*Assembly Cache Viewer* (*shfusion.dll*). Esta herramienta nos permite ver información sobre los ensamblajes, como el número de versión, clave pública, e incluso si el ensamblaje ha sido 'jitted' con anterioridad.

Otro método de examinar la caché es usando la herramienta de Caché global de ensamblajes. Esta herramienta nos permite realizar varias tareas básicas mediante la especificación de alguno de estos modificadores de línea de comando (que han de utilizarse de forma exclusiva).

- **-i.** Esta opción instala un ensamblaje en la caché global de ensamblajes. Un ejemplo sería el siguiente:

```
gacutil -i HelloWorld.DLL
```

En breve, veremos cómo añadir el ensamblaje *Module3Client* a la caché usando este modificador.

- **-u.** Esta opción desinstala un ensamblaje, incluyendo cualquier tipo de información de versión, de la caché global de ensamblajes. Si no especificamos la información de versión, se eliminarán *todos* los ensamblajes con el nombre especificado. Por lo tanto, el primer ejemplo que vemos aquí desinstala todos los ensamblajes *HelloWorld*, independientemente de su número de versión, y el segundo ejemplo desinstala las versiones especificadas:

```
gacutil -u HelloWorld
gacutil -u HelloWorld, ver=1,0,0,0
```

- **-l.** Esta opción lista los contenidos de la caché global de ensamblajes, incluyendo el nombre del ensamblaje, su número de versión, su localización y su nombre compartido.

NOTA En alguna de las primeras betas de .NET, un problema que nos podíamos encontrar al explorar la carpeta `c:\winnt\assembly` era que la extensión al intérprete de comandos no era capaz de ejecutar nada. La causa era que la extensión `shfusion.dll` no se registraba de manera adecuada. Si esto ocurre en nuestro sistema, lo que hay que hacer es abrir un intérprete de comandos y escribir lo siguiente desde la carpeta `c:\winnt\Microsoft.net\framework\vXXX`, donde XXX representa el número de versión del entorno .NET que estemos ejecutando. Obviamente, comoquiera que estamos trabajando con una beta, el nombre definitivo de la carpeta será diferente cuando .NET se lance oficialmente. Intentemos entonces localizar el archivo `shfusion.dll` y usemos la carpeta donde esté situado. A continuación mostramos la carpeta correspondiente a la versión .NET con la que estamos trabajando actualmente:

```
c:\winnt\microsoft.net\framework\v1.0.2615>regsvr32 shfusion.dll
```

Ahora que hemos creado un archivo con una clave pública y lo hemos asignado a un ensamblaje, vamos a añadir dicho ensamblaje a la caché. Para hacer esto, tecleamos lo siguiente en el símbolo del sistema:

```
gacutil -i Module3ClientApp.exe
```

Si todo sale bien, deberíamos recibir la siguiente confirmación:

```
Assembly successfully added to the cache
(Ensamblaje añadido de manera satisfactoria a la caché)
```

Llegados a este punto, podemos usar la instrucción `gacutil -l` para ver listados los ensamblajes en la caché y encontrar el `Module3Client`, o podemos usar el Visor de la caché de ensamblajes. Vamos a usar esta última opción. Si abrimos la caché con el Explorador de Windows (`C:\Winnt\Assembly` o `C:\Windows\Assembly`), deberíamos ver el ensamblaje `Module3Client` listado junto con los otros ensamblajes. Pulsemos el botón secundario del ratón y seleccionemos Propiedades, y entonces veremos cosas como el valor de la clave pública y la localización física del ensamblaje en nuestro disco duro. Un punto de referencia es que la clave pública que usted genere será diferente a la que yo haya generado, pero la idea principal es que será la misma que la mostrada si ejecutamos la aplicación `Module3ClientApp`.

CÓMO GESTIONAR VERSIONES DE ENSAMBLAJES

El manifiesto de un ensamblaje contiene un número de versión, así como una lista de todos los ensamblajes referenciados con su información de versión asociada. Como pronto veremos, los números de versión se dividen en cuatro segmentos y tienen el siguiente formato:

```
<major><minor><build><revision>
(número de versión principal)<número de versión secundaria><versión
de compilación><revisión>
```

La manera en la que funciona es que .NET, en tiempo de ejecución, usa esta información de versión para decidir qué versión de un ensamblaje utilizar en particular para una aplicación dada. Como veremos pronto, el comportamiento por defecto —llamado también política de gestión de versiones— es que una vez que una aplicación se instala, .NET usará automáticamente la versión más reciente de los ensamblajes referenciados por la aplicación si las versiones coinciden en los niveles principal (*major*) y secundario (*minor*). Podemos cambiar este comportamiento predeterminado mediante la utilización de archivos de configuración.

La gestión de versiones se aplica sólo a ensamblajes compartidos —los ensamblajes privados no lo necesitan—, y es probablemente el factor más importante de largo a la hora de decidir crear y compartir ensamblajes. Por lo tanto, veamos algo de código de ejemplo para ilustrar cómo funciona todo esto y cómo trabajar con la gestión de versiones de ensamblajes.

El ejemplo que vamos a usar es una modalidad simplificada del ejemplo de gestión de versiones que viene por defecto en el SDK .NET, y no incluye el material de Windows Forms, porque queremos centrarnos en la gestión de versiones y qué hace el entorno de ejecución para garantizarla.

Tenemos dos ejecutables que representan dos paquetes llamados Personal y Business (Negocio). Ambas aplicaciones usan un ensamblaje compartido llamado *Account* (Cuenta). La única funcionalidad que tiene la clase *Account* es la capacidad de anunciar su versión de tal modo que podamos estar seguros de que nuestras aplicaciones están usando la versión que pretendemos de la clase *Account*. Con ese fin, el ejemplo incluirá múltiples versiones de la clase *Account*, de modo que podamos ver por nosotros mismos cómo funciona la gestión de versiones con la política de gestión de versiones por defecto y cómo usamos XML para crear una asociación entre una aplicación y una versión específica de un ensamblaje.

Para arrancar, vamos a crear una carpeta llamada Accounting (Contabilidad). En esta carpeta, creamos un par de claves que utilizarán todas las versiones de la clase *Account*. Para hacerlo, teclee lo siguiente en la línea de comandos, una vez que esté en la carpeta Accounting:

```
sn /k account.key
```

Una vez que se ha creado la pareja de claves, creamos una carpeta dentro de la carpeta Accounting llamada Personal. En esa carpeta Personal, creamos un archivo llamado Personal.cs, que sería algo así:

```
//Accounting\Personal\Personal.cs
using System;

class PersonalAccounting
{
    public static void Main()
    {
        Console.WriteLine
            ("PersonalAccounting llamando a Account.PrintVersion");
        Account.PrintVersion();
    }
}
```

Dentro de la misma carpeta Personal, creamos una nueva carpeta llamada Account1000. Esta carpeta acogerá la primera versión de la clase de ejemplo *Account*. Una vez que hemos hecho eso, creamos el siguiente archivo (Account.cs) en la carpeta Account1000:

```
//Accounting\Personal\Account1000\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("../..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.0")]
public class Account
{
    public static void PrintVersion()
    {
        Console.WriteLine
            ("Esta es la versión 1.0.0.0 de la clase Account");
    }
}
```

Como podemos ver, estamos usando los atributos *AssemblyKeyFile* y *AssemblyVersion* para guiar al compilador C# hasta el par de claves creadas antes y para especificar explícitamente la versión de la clase *Account*. Ahora construimos la DLL Account de la siguiente manera:

```
csc /t:library account.cs
```

Una vez que se ha creado la clase *Account*, se ha de añadir a la caché global de ensamblajes:

```
gacutil -i Account.dll
```

Si así lo deseamos, podemos verificar que el ensamblaje *Account* está efectivamente en la caché del ensamblaje. Ahora nos dirigimos a la carpeta Personal y construimos la aplicación de esta manera:

```
csc Personal.cs /r:Account1000\Account.dll
```

Finalmente, si ejecutamos la aplicación, se producirá como resultado esta salida:

```
PersonalAccounting llamando a Account.PrintVersion
Esta es la versión 1.0.0.0 de la clase Account
```

Hasta ahora no hemos hecho nada nuevo. Sin embargo, veamos qué ocurre cuando se instala otra aplicación que usa una nueva versión más actualizada de la clase *Account*.

Creamos una nueva carpeta llamada *Business* dentro de la carpeta *Accounting*. En la carpeta *Business* creamos una carpeta denominada *Account1001* para representar una nueva versión de la clase *Account*. Dicha clase se almacenará en un archivo denominado *Account.cs*, y será similar a la versión anterior.

```
//Accounting\Business\Account1001\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.1")]
public class Account
{
    public static void PrintVersion()
    {
        Console.WriteLine
            ("Esta es la versión 1.0.0.1 de la clase Account");
    }
}
```

Como antes, construimos esta versión de la clase *Account* usando las siguientes instrucciones:

```
csc /t:library Account.cs
gacutil -i Account.dll
```

Llegados a este punto, deberíamos ver dos versiones de la clase *Account* en la caché global de ensamblajes. Ahora crearemos el archivo *Business.cs* (en la carpeta *Accounting\Business*) de la manera siguiente:

```
//Accounting\Business\Business.cs
using System;

class PersonalAccounting
{
    public static void Main()
    {
        Console.WriteLine
            ("BusinessAccounting llamando a Account.PrintVersion");
        Account.PrintVersion();
    }
}
```

Construyamos ahora la aplicación Business utilizando el siguiente comando:

```
csc business.cs /r:Account1001\Account.dll
```

Al ejecutar la aplicación se obtendrá lo que veremos a continuación, demostrando el hecho de que la aplicación Business utiliza la versión 1.0.0.1 del ensamblaje *Account*:

```
BusinessAccounting llamando a Account.PrintVersion  
Esta es la versión 1.0.0.1 de la clase Account
```

Sin embargo, ¡ejecutemos de nuevo la aplicación Personal y veamos qué pasa!:

```
PersonalAccounting calling Account.PrintVersion  
Esta es la versión 1.0.0.1 de la clase Account
```

¡Ambas aplicaciones, Personal y Business, están usando la última versión del ensamblaje *Account*! ¿Por qué? Tiene que ver con lo que se llama Operativa de corrección rápida (*Quick Fix Engineering*=QFE) y la política de gestión de versiones por defecto de .NET.

QFE y la política de gestión de versiones por defecto

Las operativas de corrección rápida, también conocidas como correcciones en caliente (*hot fixes*), son correcciones o actualizaciones que se producen para corregir un problema grave. Comoquiera que una corrección en caliente no suele modificar la interfaz del código, las posibilidades de que el código del cliente se vea afectado de manera adversa son mínimas. Por lo tanto, la política de gestión de versiones por defecto es asociar automáticamente todo el código de cliente a la nueva versión «corregida», a menos que exista un archivo de configuración para la aplicación que asocie explícitamente la aplicación con una versión específica del ensamblaje. Una nueva versión de un ensamblaje se encuentra bajo operativa de corrección rápida si la única parte del número de la versión que se modifica es la parte relativa a *revisión*.

Cómo crear un archivo de configuración en modo seguro

Esta política predeterminada de gestión de versiones puede ser adecuada para la mayor parte de las ocasiones, pero ¿qué ocurre si necesitamos que la aplicación Personal se ejecute tan sólo con la versión con la que fue creada? Aquí es donde entran los archivos de configuración XML. Dichos archivos tienen el mismo nombre que la aplicación y residen en la misma carpeta. Cuando la aplicación se ejecuta, se lee el archivo de configuración, y entonces .NET usa las etiquetas XML allí contenidas para determinar qué versión de un ensamblaje determinado usar.

Para especificar que una aplicación debería usar siempre la versión del ensamblaje con la que se creó, debemos especificar que queremos que el modo de enlace de la

aplicación sea «safe» (seguro). A esta operación se la denomina a veces de forma coloquial como *poner la aplicación en modo seguro*. Para comprobarlo, creamos un archivo llamado `PersonalAccounting.cfg` en la carpeta `Accounting/Personal` y lo modificamos de manera que aparezca como se muestra a continuación. Observemos la etiqueta `<AppBindingMode>`.

```
<?xml versión="1.0"?>
<Configuration>
<BindingMode>
<AppBindingMode Mode="safe"/>
</BindingMode>
</Configuration>
```

Ahora, si volvemos a ejecutar la aplicación `Personal`, obtendremos la siguiente salida:

```
PersonalAccounting llamando a Account.PrintVersion
Esta es la versión 1.0.0.0 de la clase Account
```

Cómo utilizar versiones específicas de los ensamblajes

Ahora observemos otro escenario común asociado a la gestión de versiones. En la última parte de este ejemplo, introduciremos un error en la ecuación. Creamos una carpeta denominada `Account1002` en la carpeta `Account\Business`. En dicha carpeta, creamos la siguiente clase `Account`. Observemos que esta vez la versión `AccountPrint` está programada a propósito para lanzar una excepción para simular una actualización de la clase `Account` que no funcione.

```
//Accounting\Business\Account1002\Account.cs
using System;
using System.Reflection;

[assembly:AssemblyKeyFile("..\\..\\Account.key")]
[assembly:AssemblyVersion("1.0.0.2")]
public class Account
{
    public static void PrintVersion()
    {
        //Esto está lanzado a propósito para simular un fallo en el código.
        throw new Exception();

        Console.WriteLine
            ("Esta es la versión 1.0.0.2 de la clase Account");
    }
}
```

Construyamos ahora esta versión del ensamblaje `Account` de la manera siguiente:

```
csc /t:library Account.cs
gacutil -i Account.dll
```

Tanto si ejecutamos la aplicación `Personal` como si ejecutamos `Business`, el resultado final será que la excepción no capturada hará que la aplicación cancele su ejecución.

Estamos imitando un fenómeno muy común en el despliegue de software —la instalación de algo que no funciona y que termina perjudicando a varias aplicaciones distintas. En el caso de la aplicación Personal, no queremos volver a modo seguro, porque queremos ejecutar la última versión disponible de *Account* (versión 1.0.0.1), y el modo seguro nos forzaría a trabajar con la primera versión (1.0.0.0).

De nuevo, la respuesta está en el archivo de configuración de la aplicación. Ahora modificaremos el archivo `Accounting\PersonalAccounting.cfg`, de tal modo que sus etiquetas XML se asemejen a lo siguiente:

```
<?xml versión="1.0"?>
<Configuration>
<BindingPolicy>
<BindingRedir Name="Account"
                Originator="32ab35a4550339b1"
                Version="*"
                VersionNew="1.0.0.1"
                UseLatestBuildRevision="no"/>
</BindingPolicy>
</Configuration>
```

Observemos que la clave especificada para el atributo *Originator* es la usada en mi propio sistema. Nosotros necesitaremos sustituirlo por el que hayamos generado cuando creamos el archivo `Account.key`. Podemos localizar el valor que necesitamos abriendo la caché de ensamblajes y localizando la clase *Account*.

Finalmente, ejecutemos la aplicación Personal de nuevo y veremos que se ejecuta sin el más mínimo problema y que la clase *Account* que estamos usando es por supuesto la versión (1.0.0.1) que solicitamos.

RESUMEN

En este capítulo hemos tratado las principales ventajas del uso de ensamblajes, incluyendo la paquetización, despliegue y seguridad. También vimos ensamblajes de un único archivo y multiarchivo, la herramienta de Generación de ensamblajes (`al.exe`), cómo crear y compartir ensamblajes usando la herramienta de Nombres únicos (`sn.exe`), y la herramienta de Caché global de ensamblajes (`gacutil.exe`). También demostramos cómo adecuar a nuestras necesidades las políticas predeterminadas de gestión de versiones usadas por .NET con la creación de archivos de configuración basados en XML.

“Internet ha desarrollado nuevas oportunidades para que creadores de música, literatura, cine y software puedan hacer su trabajo más accesible para usuarios en todo el mundo.

Para lograr este potencial, una protección rigurosa de la Propiedad Intelectual será aún más importante que antes.”

Bill Gates

Defraudar contra la propiedad intelectual es defraudar a quienes crearon la música que oímos, los libros que leemos, las películas que vemos y el software que ha hecho posible el espectacular avance tecnológico y de comunicaciones que estamos viviendo.

Microsoft

***ASEGÚRATE
que es Legal***

