USERS

INCLUYE NUEVAS TECNOLOGÍAS .NET

PROGRAMADOR .NET

DESARROLLO DE APLICACIONES EFICIENTES CON C# Y ASP

por Matías Iacono

MICROSOFT .NET FRAMEWORK 4.0 + SINTAXIS, PROCEDIMIENTOS Y FUNCIONES
C# PARA DESARROLLO MULTIPLATAFORMA + DESARROLLO CON XNA PARA WINDOWS PHONE 7
.NET AVANZADO: LINQ, PROGRAMACIÓN DE HILOS, GENÉRICOS

PROGRAMADOR .NET

DESARROLLO DE APLICACIONES EFICIENTES CON C# Y ASP

por Matías Iacono

RedUSERS

TÍTULO: Programador .NET

AUTOR: Matías Iacono

COLECCIÓN: Manuales USERS

FORMATO: 17 x 24 cm

PÁGINAS: 352

ISBN 978-987-1773-26-8

Iácono, Matías
Programador .NET. - 1a ed. - Buenos Aires : Fox Andina; Dalaga, 2011.
v. 212, 352 p. ; 24x17 cm. - (Manual users)

ISBN 978-987-1773-26-8

1. Informática. I. Título

CDD 005.3

ANTES DE COMPRAR

EN NUESTRO SITIO PUEDE OBTENER, DE FORMA GRATUITA, UN CAPÍTULO DE CADA UNO DE LOS LIBROS EN VERSIÓN PDF Y PREVIEW DIGITAL. ADEMÁS, PODRÁ ACCEDER AL SUMARIO COMPLETO, LIBRO DE UN VISTAZO, IMÁGENES AMPLIADAS DE TAPA Y CONTRATAPA Y MATERIAL ADICIONAL.

RedUSERS
COMUNIDAD DE TECNOLOGIA

redusers.com

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.

LLEGAMOS A TODO EL MUNDO VÍA OCA* Y DHL**

*SÓLO VÁLIDO EN LA REPÚBLICA ARGENTINA // **VÁLIDO EN TODO EL MUNDO EXCEPTO ARGENTINA

usershop.redusers.com // usershop@redusers.com

Matías Iacono

Ingeniero en sistemas y estudiante de Psicología, Microsoft Certified Technology Specialist, Microsoft Certified Professional Developer, Certified Scrum Master y Orador Regional para INETA Latam. Cuenta con más de quince años de experiencia en el desarrollo y gestión de software. Ha dictado más de cincuenta conferencias técnicas en distintos países latinoamericanos, y también lleva escritos y publicados variados artículos en numerosas publicaciones internacionales. Es coorganizador de los eventos Agile Open Córdoba dedicados al intercambio de conocimientos sobre formas de trabajo ágiles en el desarrollo de software.

Dedicatoria

A los amigos incondicionales que siempre me apoyan para que continúe con mis emprendimientos.

Agradecimientos

A todos los que han hecho que pueda escribir este libro.

PRÓLOGO

A través de los años, el desarrollo de software ha progresado a niveles de tal complejidad que se hace obligatorio abordar este tema desde un comienzo con conocimientos actualizados y bien fundamentados. Este es un proceso intrínsecamente creativo que no solo se lleva a cabo escribiendo código, sino que comprende otras tareas como diseñar, escribir, probar, depurar y mantener el código fuente de nuestras aplicaciones.
Desde su aparición con las primeras betas allá por el año 2000, Microsoft .Net ha ido evolucionando hasta convertirse hoy en una de las plataformas más importantes para el desarrollo de aplicaciones. Esta evolución tuvo como resultado un crecimiento en el Framework, desde la versión 1.0 hasta la 4.0, como así también el surgimiento de nuevas tecnologías como **XNA**, **WPF**, **WCF**, **WF** y **Silverlight**, solo por nombrar algunas.
El objetivo de este Framework es brindarle al desarrollador las herramientas necesarias para crear aplicaciones que correrán sobre distintas plataformas, como por ejemplo aplicaciones **Windows**, **web**, **dispositivos móviles**, **Xbox**, etcétera.
Este libro está pensado para aquellos que quieran iniciarse en el desarrollo de software, como así también para quienes provengan de otros lenguajes; para eso se abordarán las bases y conceptos teóricos, y se partirá desde una introducción a la programación hasta llegar a conceptos avanzados sobre .Net.
El autor posee una amplia experiencia docente y, por este motivo, en cada capítulo desarrolla los temas de forma clara, concisa y didáctica, con autoevaluaciones y prácticas al final de cada uno de ellos. De esta manera, permite que el lector pueda asimilar los conocimientos como así también investigar sobre los temas de su interés y profundizar en ellos, de forma autodidacta una vez finalizado este libro.
Desde mi experiencia, puedo asegurar que, para aprender sobre una determinada tecnología, no hay nada mejor que leer un libro sobre ella, y esto, sumado a horas de práctica, será la puerta de entrada que lo llevarán al mundo del desarrollo de software con **Microsoft .Net**.

Damián Galletini

Damián tiene más de 10 años de experiencia con tecnologías Microsoft y posee las certificaciones de MCAD y MCTS. Participa en forma activa en la comunidad de desarrolladores a través de distintos foros, es orador habitual en eventos técnicos en la Argentina y contribuye apasionadamente a la difusión de conocimientos en el entorno académico.

EL LIBRO DE UN VISTAZO

El objetivo de este manual es llevar al lector por el camino del desarrollo de software mediante el uso de Microsoft .Net como tecnología de soporte y C# como lenguaje de programación. En los primeros capítulos, aprenderemos los fundamentos detrás de Microsoft .Net así como conceptos generales de desarrollo de software, para luego adentrarnos en conceptos más profundos como el acceso a datos, la programación de escritorio, el desarrollo de páginas web y el manejo de tecnologías tales como WCF y WPF.

Capítulo 1

MICROSOFT .NET FRAMEWORK

En este capítulo, conoceremos qué es Microsoft .Net, su historia, las herramientas que podremos utilizar para desarrollar software para ella, así como los lenguajes de programación soportados. Además, crearemos nuestra primera aplicación de consola, que nos ayudará a entender el funcionamiento de esta tecnología.

Capítulo 2

INTRODUCCIÓN A LA PROGRAMACIÓN

Aquí aprenderemos todo lo necesario para entender cómo funcionan los lenguajes de programación, la lógica detrás de la creación de la construcción de las líneas de código de un programa, así como los distintos paradigmas computacionales. Este capítulo nos permitirá dar los primeros pasos en el mundo del desarrollo de software.

Capítulo 3

PROGRAMACIÓN ORIENTADA A OBJETOS

Uno de los paradigmas más importantes en el desarrollo de software actual es el de la programación orientada a objetos. En este capítulo, aprenderemos todo sobre este paradigma y cómo es aplicado en Microsoft .Net, en especial cuando desarrollamos programas con C# como lenguaje de programación.

Capítulo 4

ACCESO A DATOS

Las bases de datos son esenciales en el desarrollo de aplicaciones donde se requiera un medio en el cual resguardar la información generada por el usuario. En este capítulo, nos introduciremos en el uso de Microsoft SQL Server 2008 Express y veremos cómo podemos interactuar con este modelo de bases de datos desde el código C#.

Capítulo 5

PROGRAMACIÓN ORIENTADA A OBJETOS EN MICROSOFT .NET

Microsoft .Net y en especial C# explotan al máximo el paradigma de programación orientada a objetos. Aquí veremos las particularidades de C# al momento de implementar este paradigma. El uso de tipos genéricos, espacios de nombre e interfaces serán los temas principales de este capítulo.

Capítulo 6

MICROSOFT .NET AVANZADO

El manejo de excepciones, la programación de hilos de ejecución, LinQ y otros conceptos resultan de carácter avanzado en Microsoft .Net, pero al mismo tiempo necesarios para el desarrollo de buen software. En este capítulo, nos concentraremos en estos temas e iremos hasta lo más profundo de la tecnología.

Capítulo 7

PROGRAMACIÓN DE ESCRITORIO

En este capítulo, nos introduciremos en una de las áreas de aplicación del desarrollo de software y de Microsoft .Net. Las aplicaciones de escritorio para Windows constituyen una materia necesaria a la hora del desarrollo. Por lo tanto, aprenderemos cómo se crean estos programas, y cuáles son sus controles, además del concepto de las interfaces visuales.

Capítulo 8

PROGRAMACIÓN DE SITIOS WEB

Este capítulo está dedicado a otra gran área del desarrollo de aplicaciones. La Internet es, en la actualidad, una plataforma en ebullición, y Microsoft .Net a través de ASP.net tiene mucho para ofrecer. Aquí aprenderemos cómo funciona ASP.net y podremos realizar nuestros primeros sitios web.

Apéndice A

PROGRAMACIÓN ALTERNATIVA

Microsoft .Net no se limita al desarrollo de aplicaciones tradicionales. Aquí se presentan alternativas de desarrollo con esta plataforma, desde la programación de videojuegos para Windows y teléfonos inteligentes con soporte para el último sistema operativo Windows Phone 7.

Apéndice B

NUEVAS TECNOLOGÍAS EN MICROSOFT .NET

Los desarrollos más modernos requieren tecnologías modernas. La comunicación de aplicaciones y la presentación de los datos son una marca distintiva de las aplicaciones, por lo tanto, veremos cómo funcionan dos tecnologías de Microsoft .Net orientadas al diseño y a la comunicación

Servicios al lector

Para concluir el libro, en este apartado final, incluimos el índice temático, ordenado alfabéticamente, que nos permitirá acceder, en forma rápida y precisa, a la mayoría de los temas tratados a lo largo de este manual dedicado a la Programación.Net.

! INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual encontrará una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados.
Cada recuadro está identificado con uno de los siguientes iconos:

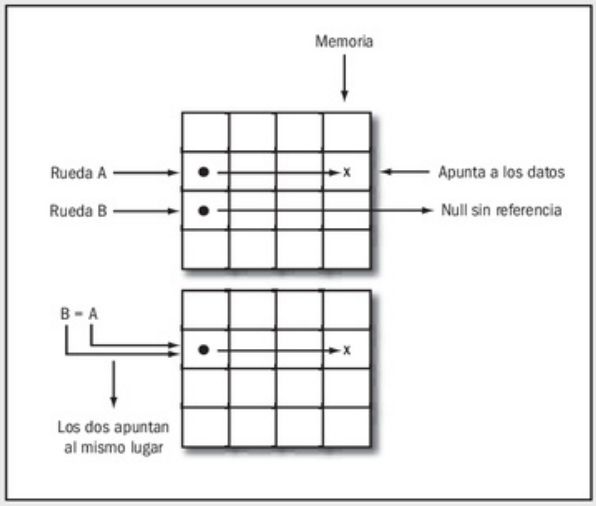

CURIOSIDADES E IDEAS

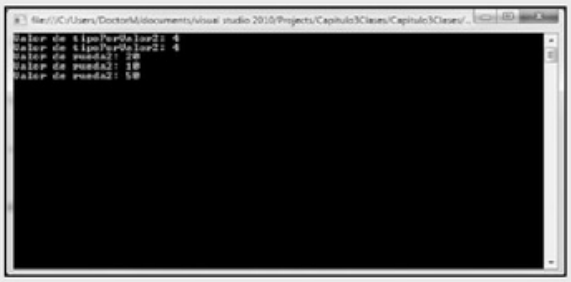

ATENCIÓN


DATOS ÚTILES Y NOVEDADES


SITIOS WEB

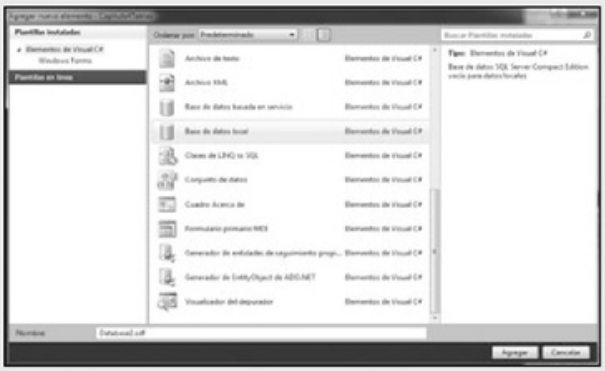
CONTENIDO





Capítulo 4

ACCESO A DATOS

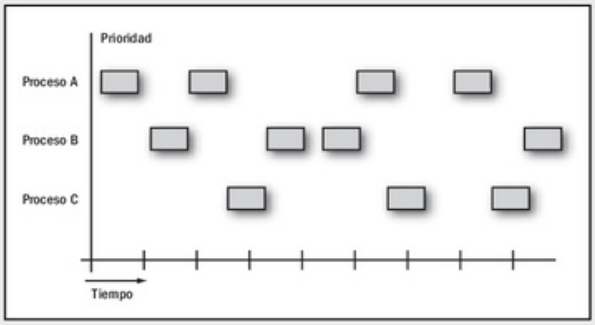


Capítulo 5

PROGRAMACIÓN EN MICROSOFT .NET

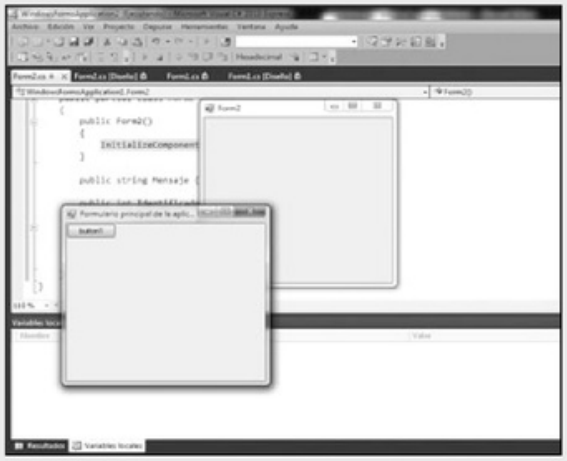
Capítulo 6

MICROSOFT .NET AVANZADO



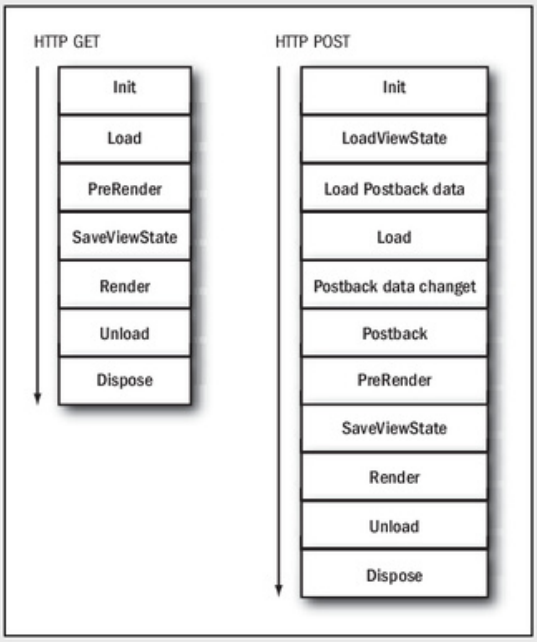
Capítulo 7

PROGRAMACIÓN PARA ESCRITORIO



Capítulo 8

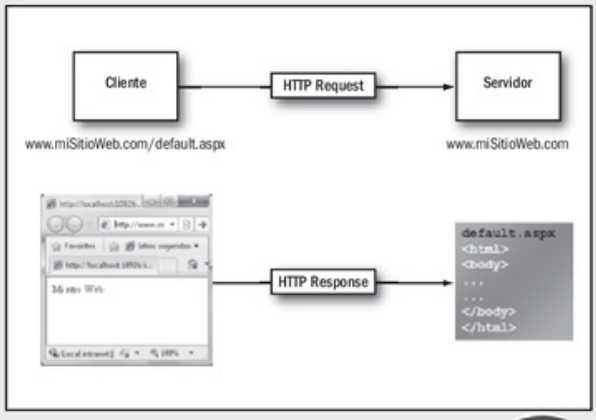
PROGRAMACIÓN DE SITIOS WEB



Apéndice A

¡Capítulo Online!

PROGRAMACIÓN ALTERNATIVA



Apéndice B

¡Capítulo Online!

NUEVAS TECNOLOGÍAS EN PROGRAMACIÓN .NET

Servicios al lector

INTRODUCCIÓN

Este libro tiene el objetivo de iniciar al lector en el desarrollo de software con **Microsoft .Net**, permitiéndole aprender desde los conceptos esenciales y más comunes de la programación aplicada a cualquier lenguaje de programación hasta las últimas técnicas aplicadas bajo esta tecnología.

Por otro lado, este libro no está orientado solo para quienes deseen aprender a programar bajo esta tecnología, ya que también aquellos desarrolladores con experiencia podrán encontrar en él una serie de referencias a las últimas técnicas de codificación encontradas en el **Microsoft .Net Framework 4.0**, que abrirán una puerta a la investigación y la aplicación de estas en sus desarrollos.

Por lo tanto, cuando lea este libro de principio a fin, pasando por cada capítulo, podrá aprender y entender cómo funcionan estas tecnologías, dónde aplicarlas, cómo conjugarlas entre sí y con otras tecnologías, qué herramientas de desarrollo son convenientes para cada área de aplicación con el fin de obtener los mejores resultados para sus desarrollos.

La lectura de este libro no solo lo llevará a conocer la tecnología, sino que le dará experiencia en uno de los lenguajes más populares que se encuentran en el mundo .Net, por lo tanto, junto con el aprendizaje de técnicas de programación orientada a objetos, encontrará desarrollo de aplicaciones de consola, aplicaciones de escritorio para Windows, aplicaciones orientadas a servicios con **Windows Communication Foundation**, desarrollo de aplicaciones web utilizando **ASP.net**, y, además, podrá dominar **C#** como lenguaje de programación.

Con todo esto, como lector, podrá mantenerse actualizado con la última propuesta de Microsoft para el desarrollo de software, y a la vez encontrar un punto de partida hacia un mundo mucho más profundo y rico en materia de desarrollo de software.

Capítulo 1

Microsoft .Net Framework

En este primer capítulo, aprenderemos los conceptos primordiales del Microsoft .Net Framework. Responderemos las primeras preguntas sobre esta tecnología que nos ayudarán a entender cómo funciona, dónde aplicarla, sus diferencias con otros modelos de desarrollo, así como los lenguajes de programación involucrados en su uso. Finalmente, realizaremos nuestro primer programa, que nos servirá de preámbulo del conocimiento que adquiriremos en cada uno de los capítulos de este libro.

¿QUÉ ES MICROSOFT .NET FRAMEWORK?

En este primer capítulo de la obra, haremos una introducción a los principales conceptos de **.Net**. Para entender el **Microsoft .Net Framework** es necesario verlo como un conjunto de componentes que interactúan entre sí y que, al mismo tiempo, brindan soporte al desarrollador de software y al software mismo creado bajo esta plataforma. Por lo tanto, veamos cada una de estas partes por separado.

El primer elemento que necesitamos considerar al momento del desarrollo de software bajo Microsoft .Net son los requisitos previos con los que tanto nuestro ordenador como el del usuario de nuestras aplicaciones deberá contar. Como requisitos no nos referimos al hardware, sea memoria RAM o espacio en disco, sino al mismo Microsoft .Net Framework que nos veremos en posición de instalar en cada ordenador donde necesitemos ejecutar nuestro software.

Si bien esto pudiera parecer un punto desfavorable al momento de elegir esta plataforma de desarrollo, en realidad, conlleva mayores beneficios que perjuicios. El Microsoft .Net Framework constituye el motor de ejecución de nuestras aplicaciones; esto quiere decir que el código que nosotros podamos escribir bajo esta tecnología será independiente del sistema operativo en el cual se ejecute, ya que es justamente el motor el que interpretará las líneas de código y las ejecutará según sea necesario.

Esto nos lleva a la posibilidad de que las aplicaciones puedan ejecutarse en distintos sistemas operativos sin tener que realizar modificaciones a nuestro código, además de poder funcionar en diferentes arquitecturas de hardware, nuevamente, sin tener que realizar modificaciones al código. Un ejemplo claro de esto es que, dentro del abanico de sistemas operativos provistos por Microsoft, nuestras aplicaciones funcionan sin mayores inconvenientes. Pero, cuando decimos sistemas operativos, no nos referimos a las diferentes versiones de Windows, sino que hacemos referencia a ellos e, incluso, a aquellos que salen fuera de las arquitecturas de hardware que podamos encontrar en nuestras computadores personales de escritorio o portátiles.

La reconocida empresa informática Microsoft provee sistemas operativos para dispositivos multimedia portables como **Zune**, o consolas de videojuegos como **Xbox 360**, así como para teléfonos celulares como las versiones de **Windows Mobile**, y **Windows Phone 7**, recientemente lanzado.

MULTIPLATAFORMA

Si bien hemos comentado que es posible ejecutar código en diferentes plataformas sin modificaciones, es necesario considerar que cada una de estas posee componentes particulares que no tienen otras, por lo que, en estos casos deberemos realizar las modificaciones necesarias al código para que sea compatible con estas particularidades.

En todos los casos, Microsoft .Net Framework está presente, y, si hacemos un desarrollo cuidadoso, nuestro código podría funcionar en forma automática en cada uno de estos dispositivos sin tener que realizar cambios sustanciales a lo desarrollado.
Todo esto nos lleva a una pregunta fundamental: ya que hablamos de portabilidad es natural que nos cuestionemos si existe la posibilidad de que nuestros desarrollos funcionen fuera de ambientes provistos por Microsoft.
La respuesta es que lo podremos lograr siempre y cuando tengamos el motor de ejecución separados del código o programa creado. Porque si alguien desarrollase un motor de ejecución para otro sistema operativo u otra arquitectura, respetando los parámetros esperados por Microsoft .Net, indudablemente tendríamos nuestras aplicaciones funcionando en dicho sistema operativo.
Y es así como un proyecto, que nació independiente, ha llevado el desarrollo con Microsoft .Net a **Linux** y **MacOS X**, debemos recordar que este último es el sistema operativo de Apple. Este proyecto se conoce como **Mono** e, incluso, no se ha quedado solo en proveer soporte para las aplicaciones de escritorio que pudiéramos desarrollar, sino que, con el correr del tiempo, se ha creado también una versión para soportar el desarrollo con **Silverlight** (tecnología provista por Microsoft para realizar aplicaciones ricas para la Web) bajo Linux.
Esta plataforma posee el nombre de **MoonLight** y, de la misma forma que Mono, se encuentra en constante mejora al punto de que, con la llegada de los celulares con soporte táctil, Mono ha apuntado sus esfuerzos hacia este tipo de dispositivos, y llega a desarrollar, en la actualidad, aplicaciones para **iPhone** con Microsoft .Net.
Por último, es necesario aclarar un punto importante que muchas veces provoca confusiones, en especial cuando estamos iniciándonos en el desarrollo de software con Microsoft .Net. Este radica en que, si bien dijimos que necesitaremos instalar el motor de ejecución en cada ordenador, esto será necesario solo en aquellos que requieran ejecutar una aplicación del tipo escritorio, o sea, aplicaciones que funcionen directamente sobre el sistema operativo del usuario.
Resulta muy necesaria la aclaración del párrafo anterior, ya que, con Microsoft .Net, también podremos desarrollar sitios web en los que, para estos casos, no será necesario instalar ningún aditamento en el ordenador de cada usuario que visite nuestro sitio web en la red mundial de redes.

MONO Y MOONLIGHT

Podemos conocer mucho más sobre el proyecto Mono y MoonLight visitando sus sitios web oficiales. El sitio web oficial de Mono se encuentra en la siguiente dirección: **www.mono-project.com**. Dentro de este, encontraremos la sección de MoonLight. Si queremos acceder directamente, podremos hacerlo desde la siguiente dirección: **www.mono-project.com/Moonlight**

De cualquier manera, en el caso de que nos focalicemos en el desarrollo para Windows en sus distintas versiones, sepamos que estos traen por defecto instalado el motor de ejecución de Microsoft .Net. Por otra parte, si la PC donde estemos trabajando o necesitemos instalar una aplicación realizada con Microsoft .Net no contara con este motor, podremos descargar la última versión desde la siguiente dirección: **www.microsoft.com/downloads/en/details.aspx?FamilyID=0a391abd-25c1-4fc0-919f-b21f31ab88b7**

El origen de Microsoft .Net Framework

Microsoft .Net Framework hace su primera aparición a principios de 2002 con la versión 1.0. Desde aquel momento hasta la fecha, se han sucedido diferentes versiones y, en la actualidad, contamos con la versión 4.0, y es sobre esta versión sobre la cual nos concentraremos en este libro. Podemos ver la evolución de la plataforma en la **Figura 1**. ¿Pero cuáles han sido las motivaciones que llevaron a que Microsoft .Net viera la luz? ¿Qué se intentaba solucionar?

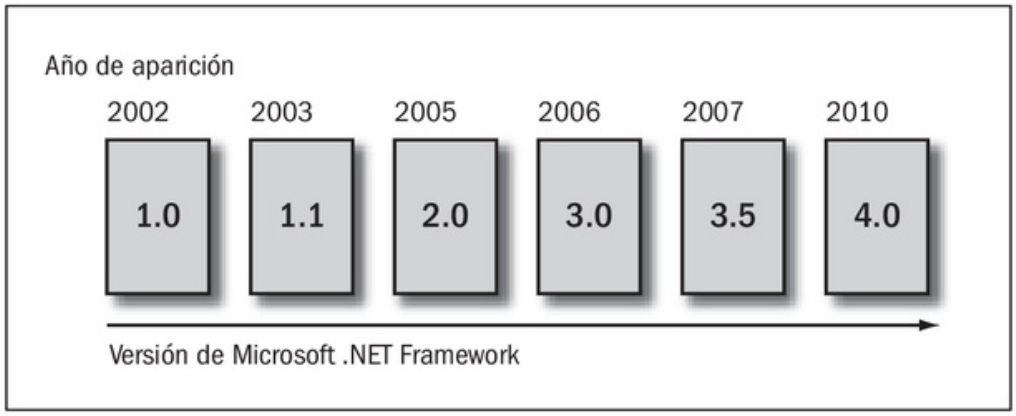


***Figura 1.** Microsoft .Net nace en 2002 con la **versión 1**. En el transcurso de ocho años, ha ido mutando y mejorándose, hasta alcanzar el potencial que tiene en la actualidad.*

Es necesario retroceder un poco en el tiempo para comprender algunos de los problemas que acarreaba el desarrollo de software bajo Windows. Este viaje en el tiempo nos remonta hasta el uso de **DLLs** (*Dynamic Link Library* o, en castellano, **librería de vínculos dinámicos**). Estas DLLs eran usadas, en la mayoría de los casos, como unidades de código independientes y reusables. Con estas, sí necesitábamos que diferentes porciones de nuestro código fueran utilizadas por otros programas además del nuestro. Podíamos colocar dichas líneas de código en este archivo, compilarlo y redistribuirlo para su uso y así obteníamos, por ejemplo, funciones para hacer uso de la impresora dentro de nuestro DLL y, cada vez que necesitáramos de esta funcionalidad, solo deberíamos consumir sus servicios sin tener que reescribirla de manera directa en nuestro programa.

Resultaba una idea sumamente útil, que generó una industria que muchas empresas supieron aprovechar al crear funcionalidad empaquetada dentro de estos DLLs con la finalidad de venderlas. ¿Entonces, cuál es el problema? Todo parecía perfecto. Pero la realidad no era tan armónica, ya que cada DLL necesitaba de un identificador único, el cual era registrado en el **Registro de Windows** para que este y los demás programas pudieran saber dónde se encontraba el archivo, así como el detalle de su funcionalidad y otras características relacionadas.

Por lo tanto, si nosotros distribuíamos uno de estos archivos y cientos de otros programas los usaban, esto es, tenían algún tipo de vínculo con él, cualquier cambio que le hiciéramos afectaría a todos los programas. Un cambio en su funcionalidad o un nuevo parámetro agregado a una función haría que, simplemente, todas estas aplicaciones dejaran de funcionar, y sus desarrolladores se vieran en la obligación de realizar los cambios pertinentes para que volvieran a ser estables.

Estas rupturas resultaban más comunes de las que pudiéramos pensar. Un nuevo programa que instaláramos en nuestra computadora y que hiciera uso de una versión nueva del DLL, que otras aplicaciones ya instaladas en nuestro ordenador estuvieran usando, haría que, al superponer el viejo DLL con el nuevo, todas las demás aplicaciones dejaran de funcionar.

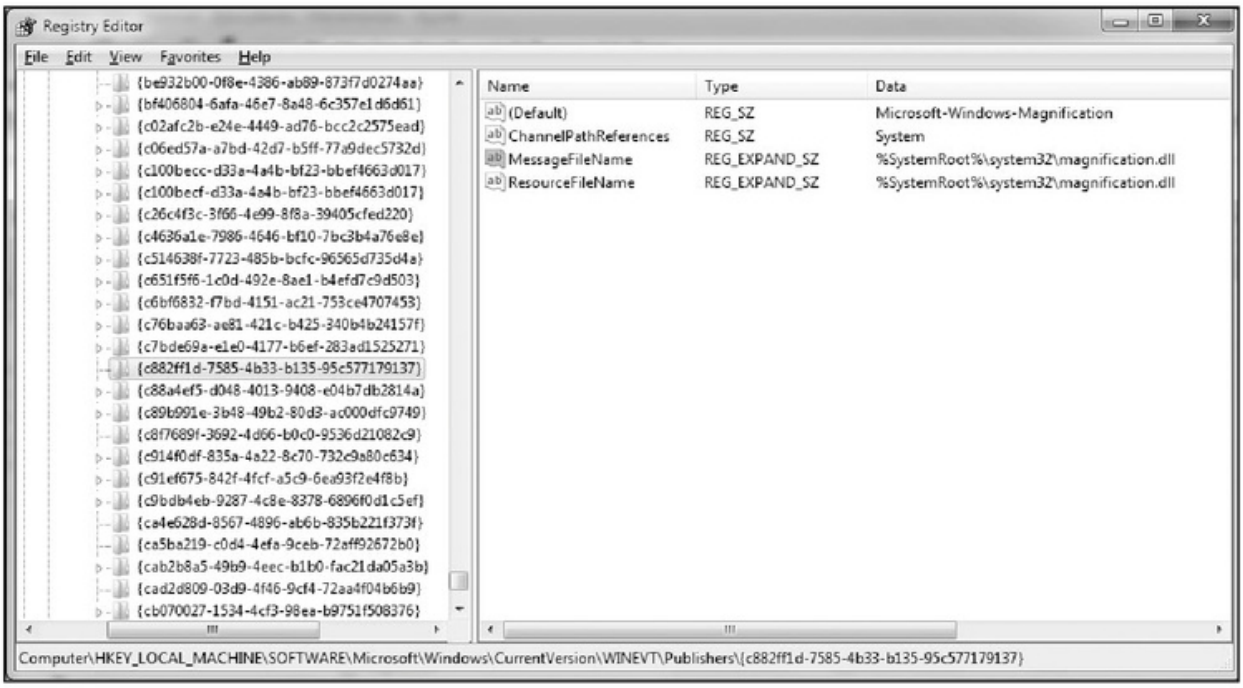


***Figura 2.** El Registro de Windows muestra dónde y cómo está registrada la DLL llamada Magnification. La modificación de estos valores puede acarrear problemas de ejecución.*

Esta dificultad tiene un nombre y, si hemos podido cuantificar el problema, entenderemos que este le hace justicia. El **DLL Hell** (infierno de las DLLs en castellano) fue, ciertamente, un gran dolor de cabeza y, por eso, es uno de los grandes problemas que

atacó Microsoft .Net. Este arranca de raíz el conflicto de las DLLs permitiéndonos, de la misma forma, aislar funcionalidad en archivos separados para luego consumirlos, pero sin la necesidad de registrarlos como únicos a nivel del sistema operativo, sino que podremos usarlos solo con mantener una copia de ellos junto al programa que necesitemos ejecutar.

Así, si una nueva versión del mismo archivo fuera creada, solo necesitaríamos reemplazarla en aquellos programas que la necesiten, y los demás programas no se verían afectados, y podrían funcionar de manera normal.

Esta mejora también trajo aparejado otro cambio significativo en la forma en cómo funcionaban las aplicaciones bajo Windows, debido a que desde ese momento en adelante ya no era necesario instalar la aplicación para que funcionara. Al no tener que registrar ninguna referencia en el registro de Windows, con el solo hecho de copiar la aplicación en una carpeta, esta se encontraría lista para funcionar y, al mismo tiempo, para que ya no existiera físicamente en nuestro ordenador: con solo borrarla, dicha carpeta quedaría fuera del sistema.

Este último punto puede ser trivial, pero dista de serlo ya que muchas aplicaciones, por mala programación o porque simplemente no podían hacerlo, al desinstalarse no limpiaban en forma correcta el **Registro de Windows** o las carpetas donde residían, dejando residuos que ocuparán espacio físico en nuestros discos duros y, a su vez, hacían más lentas nuestras máquinas. Esto último se debía a que, al no eliminarse correctamente todas las entradas del **Registro de Windows**, el mismo sistema operativo debía consumir memoria y tiempo para conocer estas entradas y, por ende, hacerlo más lento.

Otro gran cambio introducido con la llegada de Microsoft .Net ha sido la posibilidad de unificar el desarrollo para distintas plataformas objetivo, y si bien ya hemos hablado de la eventualidad de ejecutar código en diferentes sistemas operativos y hardware, en este caso nos referimos a la posibilidad de unificar la funcionalidad para diferentes áreas de desarrollo.

Antes de Microsoft .Net, cuando necesitábamos crear una aplicación para trabajar como aplicación de escritorio, teníamos que utilizar algún lenguaje de programación que tuviera como objetivo la creación de este tipo de aplicaciones; si necesitábamos crear un sitio web, debíamos buscar algún lenguaje que sirviera para esto, lo

MICROSOFT .NET FRAMEWORK

Cuando trabajemos con Microsoft .Net y necesitemos instalar el motor de ejecución en la PC donde funcionará nuestra aplicación, no será necesario instalar la versión para desarrolladores. Esta contiene un tamaño considerable por las herramientas que provee. Por el contrario, podremos instalar la versión redistribuible, más pequeña y con lo fundamental para la ejecución de aplicaciones.

mismo con los diferentes objetivos que persiguiéramos, por lo que nos veíamos obligados a aprender diferentes lenguajes de programación, a usar distintos ambientes de desarrollo, saber cómo lidiar con cada componente y, a su vez, cada uno de ellos poseía un comportamiento diferente dependiendo de dónde lo usáramos.
Por el contrario, Microsoft .Net quiso terminar con este problema proveyéndonos una plataforma común, sin importar el lenguaje de programación que usásemos o el objetivo que persiguiéramos. Y si bien en las primeras etapas de la plataforma aún existía una difusa frontera sobre esta característica, en la actualidad, la maduración del modelo nos ha dado la capacidad plena de poder desarrollar para múltiples plataformas con el mismo lenguaje y aplicando los mismos conceptos, algoritmos e ideas independientemente del resultado perseguido.

¿Cómo funciona Microsoft .Net?

Podemos decir que, en este momento, tenemos una idea general sobre la conveniencia de usar Microsoft .Net como plataforma de desarrollo por sobre otras plataformas para Windows. Hemos enumerado algunos de los problemas que intenta solucionar y cómo ha evolucionado desde su aparición. Por lo tanto, es momento de profundizar más sobre los conceptos de su funcionamiento y cómo podremos aprovecharnos de ellos. La **Figura 3** muestra los cimientos de Microsoft .Net.

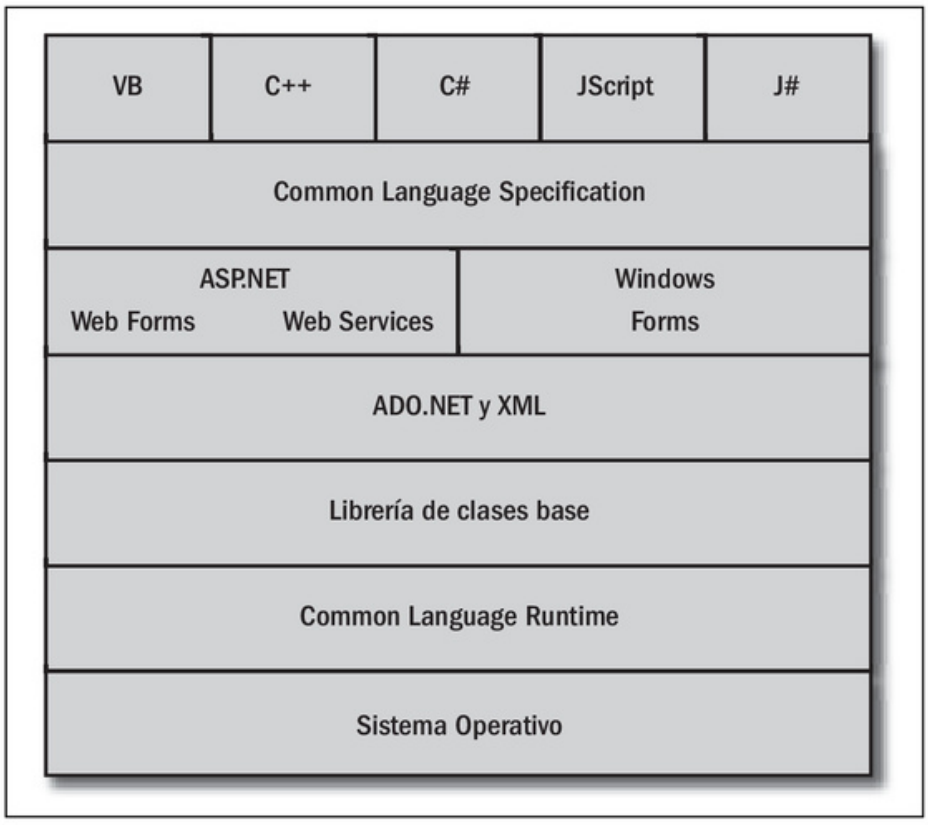


Figura 3. *Arquitectura de Microsoft .Net, desde los lenguajes soportados (superior), hasta el contacto con el sistema operativo (inferior).*

Observemos la **Figura 3**. Desde la parte superior, podemos ver los distintos lenguajes de programación soportados por Microsoft .Net; si somos completamente noveles en el desarrollo de software, es necesario que aclaremos lo que entendemos por lenguaje de programación. Entonces, un lenguaje de programación es similar al lenguaje que los seres humanos utilizamos para comunicarnos entre nosotros; dependiendo del grupo con el cual queramos entablar una comunicación, será necesario utilizar uno u otro lenguaje, y, por el mismo motivo, deberemos saber entender dicho lenguaje para comprender qué se nos quiere decir.

En el caso del desarrollo de software, diferentes lenguajes poseen aplicaciones o usos distintos y definidos: algunos enfocados a resolver problemas matemáticos, otros para el desarrollo de páginas web, algunos otros para crear software que trabajará en algún sistema operativo específico, y así la lista continúa.

Muchas veces necesitaremos elegir un lenguaje de programación cuidadosamente, ya que aquello que necesitemos construir puede que no esté soportado por el lenguaje elegido; también, hay que considerar la forma en la cual podríamos construirlo, ya que alcanzar nuestro objetivo podría requerir de mucho más esfuerzo que si hubiéramos optado por otro lenguaje de programación.

Por lo tanto, es indispensable que sepamos elegir el lenguaje correcto para el trabajo que necesitemos realizar. En nuestro caso, y para el resto del libro, elegiremos a C# como el lenguaje por utilizar, porque es, dentro del ambiente de Microsoft .Net, uno de los más versátiles y el que soporta la mayor cantidad de desarrollos objetivos posibles dentro de Microsoft .Net.

Volviendo al punto inicial, son muchos los lenguajes que soportan el desarrollo para Microsoft .Net, y notemos que no es a la inversa, donde Microsoft .Net soportaría diferentes lenguajes. Esto se debe a que Microsoft .Net brinda una capa extra que especifica el comportamiento, las cláusulas y contratos por seguir para que, independientemente del lenguaje usado, estos mantengan coherencia entre sí y hacia el motor de ejecución de Microsoft .Net.

Esta capa, el **CLS** (*Common Language Specification* o, en castellano, especificaciones de idioma común) entonces permitiría que los distintos lenguajes hablaran, de alguna forma, el mismo idioma en relación al uso de variables, eventos, firmas y demás. Gracias a esta sección, es posible que código creado en un lenguaje como Visual Basic.Net

III EL REGISTRO DE WINDOWS

Si queremos ver el **Registro de Windows**, deberemos presionar sobre el icono **Inicio** de Windows; dependiendo de la versión que usemos, presionaremos en **Ejecutar** y escribiremos **RegEdit**. En caso de Windows Vista o 7, solo escribiremos **RegEdit**. Deberemos tener cuidado si realizamos alguna modificación, esto podría causar mal funcionamiento del sistema operativo.

pueda ser usado por código creado en C#, y viceversa, o, yendo más lejos, que un lenguaje como Cobol para .Net, J#, Borland C# y otros, puedan comunicarse entre sí.
El siguiente grupo de componentes que encontramos de acuerdo con la **Figura 3** es el que se refiere a las tecnologías **ASP.net** y **Windows Forms**. En un principio, la intención de Microsoft .Net era la de traer soporte para dos de las grandes áreas del desarrollo, por una parte **ASP.net** con soporte para el desarrollo web y una evolución de **ASP** *(Active Server Page* o, en castellano, páginas activas de servidor).
Esta tecnología nos proporciona la posibilidad de desarrollar sitios web con Microsoft .Net de tal forma que aquellos desarrolladores que siempre hubiesen trabajado en desarrollos para escritorio les resultase un poco más simple realizar la transición al ambiente web. Tengamos en cuenta que Internet y la posibilidad de desarrollar para esta ha ido creciendo cada vez más, y se tornó casi en la alternativa por defecto a la hora de emprender un desarrollo. Por lo tanto, contar con una solución que pudiera inducir con mayor facilidad a desarrolladores de escritorio en el mundo web resultaba una apuesta importante.
El siguiente componente es **Windows Form** o formularios para Windows, el cual hace referencia exclusiva al desarrollo bajo Windows.
ADO.net y XML brinda el soporte para la manipulación de datos, tanto hacia motores de bases de datos como de **XML** (*Extensible Markup Language* o, en castellano, lenguaje extensible de etiquetas), el mismo que provee gran flexibilidad para el transporte y comunicación de datos. En la actualidad, XML es utilizado por la mayoría de los programas, ya sea en ambientes web como en aplicaciones de escritorio. Su versatilidad ha provocado que, en muchos casos, se hiciera un uso desmedido de este por parte de los desarrolladores, que ha causado más problemas que soluciones.
La capa que continúa es la llamada **Base Class Library** o librería base de clases en castellano. Esta capa contiene gran cantidad de código común al que podemos acceder desde nuestro código. La librería de clases nos facilita funcionalidad para diferentes tareas que, de otra forma, nos veríamos forzados a realizar por cuenta propia. Si necesitamos leer un archivo desde el disco duro de la computadora, no es necesario que escribamos el código que se encargue de navegar por los diferentes sectores del disco en busca de este archivo, esta lógica ya se encuentra encapsulada dentro de la librería base de clases para su uso.

III MÁS SOBRE EL CLS

Si estamos interesados en saber más sobre el CLS, sus capacidades, cualidades, cómo interactúan los distintos lenguajes con el Microsoft .Net Framework, podemos visitar: **http://msdn.microsoft.com/es-es/library/12a7a7h3%28v=VS.100%29.aspx**. En esta dirección podremos encontrar una descripción detallada de qué es y cómo funciona el CLS dentro de Microsoft .Net.

Otras funcionalidades, como los tipos de datos, algunos controles y componentes para el desarrollo Windows o web (funcionalidad para crear ventanas o comprimir archivos o enviar información por la red) están dentro de la librería base de clases, listas para que las utilicemos. Además de proveer este gran conjunto de código, la librería base de clases tiene una segunda función, y es que, como hablábamos al principio, múltiples lenguajes de programación pueden coexistir y funcionar bajo Microsoft .Net, por lo tanto, estos lenguajes pueden hacer uso del mismo conjunto de código alojado en la librería base de clases y no solo aprovechar el código existente, sino que el código creado en un lenguaje podrá ser entendido por cualquier otro. Para entender esto, imaginemos el siguiente ejemplo donde el **lenguaje A** necesita para su funcionamiento acceder a una base de datos. La librería base de clases trae consigo funcionalidad para manipular distintos motores de bases, por lo tanto, con pocas líneas de código por parte de este lenguaje podrá conectarse, consultar y mostrar datos desde la base de datos elegida. Así, otro lenguaje, el **lenguaje B**, gracias a la definición de la librería base de clases, podría hacer uso de los mismos métodos y funciones para conseguir un objetivo similar sin tener que implementar sus propios algoritmos. Ambos lenguajes hacen uso de código común simplificando el desarrollo para el programador.

El último cimiento de Microsoft .Net es el **CLR** (*Common Language Runtime* o, en castellano, lenguaje común de ejecución) y es el que ejecuta nuestras aplicaciones. Este es el más cercano al sistema operativo y es el encargado de procesar el código creado para transformarlo en código que pueda ser ejecutado e interpretado por el sistema operativo donde se esté intentando correr nuestro programa.

En este punto cabe aclarar que el código creado bajo Microsoft .Net se considera **código manejado**, esto quiere decir que nosotros, con nuestro código, no estaremos trabajando al mismo nivel que, por ejemplo, el sistema operativo. A diferencia de otros lenguajes de programación como **C** o **C++**, una de las principales ventajas de trabajar bajo Microsoft .Net es que este motor de ejecución es el que se encargará de, por ejemplo, gestionar la memoria, su uso, así como su liberación.

Lo explicado en el párrafo anterior puede parecer trivial, a simple vista, pero en realidad, el manejo de memoria puede ser un tremendo dolor de cabeza cuando debemos hacerlo por nuestra propia cuenta.

III MÁS SOBRE LA LIBRERÍA BASE DE CLASES

Si estamos interesados en conocer más acerca de la librería base de clases, en especial su amplio contenido y funcionalidad, podemos visitar la siguiente dirección del sitio oficial de Microsoft: **http://msdn.microsoft.com/es-es/library/hfa3fa08.aspx**. En ella, encontraremos una descripción detallada de las características de este conjunto de elementos.

La incorrecta gestión podría acarrear problemas de pérdida de datos hasta dejar fuera de funcionamiento al sistema operativo y, con él, toda la computadora. Por supuesto, es necesario destacar que, como comentábamos al principio, dependiendo de nuestros objetivos, muchas veces es necesario poder gestionar la memoria con lenguajes como C++. Por lo tanto, todo código que escribamos con cualquier lenguaje soportado por Microsoft .Net no será un código final, el cual pueda ser ejecutado directamente por el sistema operativo, este código es conocido como código **IL** (*Intermediate Language* o, en castellano código intermedio). Este código es igual independientemente del lenguaje de programación que hayamos usado para crear nuestra aplicación. Lo que sucede es que cada lenguaje de programación viene acompañado de un motor de compilación o **compilador** que lee y reconoce lo que hemos escrito bajo dicho lenguaje y lo transforma en código IL común para que el CLR pueda interpretarlo y ejecutarlo. Podemos ver en la **Figura 4** un ejemplo de código IL.

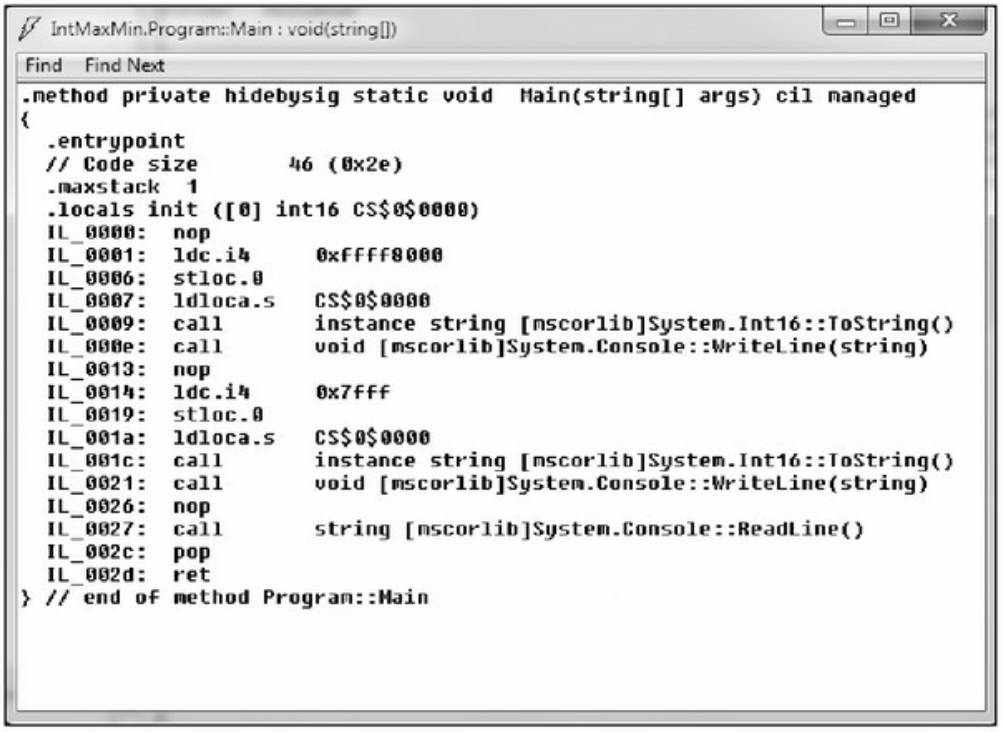


***Figura 4.** Una vez compilada nuestra aplicación, obtenemos como resultado el código IL de la ventana derecha.*

MICROSOFT .NET Y LENGUAJES

Microsoft .Net Framework no solo soporta los lenguajes de programación propuestos por Microsoft. También podemos encontrar otros como Boo, Icc, Clarion#, Cobol, Corba, Eiffel, Fortran, Lisp, PHP, Perl y más. Esto puede resultar especialmente interesante ya que no necesitamos aprender un nuevo lenguaje para desarrollar en Microsoft .Net Framework.

En la **Tabla 1** podemos ver una lista de algunos de los mecanismos implementados por el CLR y que ayudan a la ejecución de nuestras aplicaciones.

TIPO	DESCRIPCIÓN
Compilación Just in Time	La compilación JIT (o justo en el momento) hace referencia a la capacidad de compilar partes del código IL de acuerdo con la necesidad de uso de este.
Manejo de la memoria	La búsqueda de espacio libre en la memoria, así como su limpieza es tarea del CLR. Dentro de los procesos que se ejecutan para mantener la estabilidad del uso de memoria, podemos encontrar el GC (Garbage collector o, en castellano, recolector de basura).
Manejo de excepciones	El CLR se encarga de gestionar los posibles errores que las aplicaciones en ejecución pudieran arrojar. Esto previene que cualquier error no controlado se propague a todo el sistema operativo.
Acceso a la metadata	Parte de la información que se incluye en nuestro código compilado es información de sí mismo. Métodos y funciones, propiedades, así como información sobre seguridad y otros acompañan a nuestros programas. Es posible, por lo tanto, consultar esta información para manipularla.
Gestión de seguridad de código	La ejecución de nuestras aplicaciones no queda librada al azar. Es posible definir un sinnúmero de reglas de ejecución que van a actuar sobre las aplicaciones creadas con Microsoft .Net, impidiendo la ejecución de aquellas que no cuenten con autorización, o restringiendo el acceso a ciertos recursos según sea necesario.

***Tabla 1.** Algunos de los mecanismos implementados por el CLR.*

Herramientas de desarrollo

En el mercado, existen muchas herramientas para desarrollar software para la plataforma Microsoft .Net. De cualquier manera, la herramienta por excelencia es la que provee Microsoft bajo el nombre de Visual Studio. En la actualidad, la última versión de **Microsoft Visual Studio** es la 2010, lanzada al mercado ese mismo año.
Microsoft Visual Studio 2010 cuenta con diferentes versiones, cada una de estas orientadas a distintas áreas de la ingeniería y el desarrollo de software, por lo que encontraremos versiones para la realización de pruebas, para la creación de código, o incluso para el desarrollo de la arquitectura de una aplicación.
Es importante elegir una versión adecuada al trabajo que realizaremos aunque, si estamos iniciándonos en el desarrollo de software, Visual Studio 2010 Professional puede ser una buena elección y si ya contamos con algo de experiencia, la versión Ultimate, la versión más grande y completa de Visual Studio 2010, puede resultarnos la mejor alternativa. Es importante aclarar que, para el desarrollo de aplicaciones con Microsoft .Net, no es necesario contar con herramientas de este tipo; es preciso también hacer una distinción entre **IDE** y lenguaje de programación, ya que suelen confundirse estos dos términos. **IDE** (*Integrated Development Environment* o, en castellano, ambiente integrado de desarrollo) es un conjunto de herramientas o programa que nos ayuda en la construcción del código que estemos realizando. Así, un IDE contará con atajos,

ayuda en la escritura y otras características que podrán acelerar nuestro desarrollo, tanto como advertirnos de posibles errores sintácticos al momento de escribir las distintas líneas de código. Por consiguiente, no es necesario que contemos con un IDE para poder crear código, lo único que se requiere es conocer el lenguaje y contar con el compilador de dicho lenguaje para poder obtener el producto terminado. En el caso de Microsoft .Net, tanto el compilador como algunas herramientas son completamente gratuitas, pudiendo desarrollar para esta plataforma sin inversión de dinero alguna.
Pero, lamentablemente, el IDE provisto por Microsoft no lo es y, si queremos acceder a la potencia que este nos brinda, será necesario comprarlo. Por supuesto, la misma empresa provee una versión más liviana, pero igual de poderosa, sin costo alguno. Estas versiones se las encuentra con el nombre de versiones Express y pueden ser descargadas directamente desde el sitio web de Microsoft. Para descargar e instalar la versión Express de Visual Studio 2010, deberemos ingresar a la siguiente dirección **www.microsoft.com/express/Downloads/#2010-Visual-CS** y buscar la versión con la cual nos sintamos más cómodos a la hora de desarrollar código.
Si decidimos utilizar Visual Studio 2010 Express, será necesario para este libro contar con dos versiones del programa. La primera versión que deberemos descargar e instalar es **Visual C# 2010 Express**, que usaremos para realizar los distintos ejemplos de código, tanto en los primeros capítulos como cuando ahondemos en el desarrollo de aplicaciones para escritorio. La segunda versión es la denominada **Visual web Developer 2010 Express**. Podemos ver la lista completa de versiones de Visual Studio 2010 Express en la **Figura 5**.

Figura 5. *Desde el sitio web de Microsoft, podremos descargar distintas versiones de Visual Studio 2010 Express. Nosotros necesitaremos descargar e instalar las versiones web y C#.*

Existen diferentes lenguajes de programación; incluso, en la lista de la **Figura 5**, se puede observar una versión para desarrolladores que utilicen Visual Basic.Net como lenguaje de programación. Nosotros nos enfocaremos en C# como lenguaje de aprendizaje por varios motivos: uno de los más importantes es que C# es considerado un lenguaje estándar **ECMA** e **ISO/IEC**; otro motivo interesante es su raíz en lenguajes como C y C++, lo que brinda mayor flexibilidad a la hora de escribir código y resulta en una sintaxis más clara y elegante.

Nuestra primera aplicación

Solo hemos hablado sobre Microsoft .Net y las herramientas de desarrollo, y no así de conceptos de programación. De cualquier manera, es necesario que practiquemos un poco con estos dos elementos para que no nos resulten ajenos.

Aplicación de consola

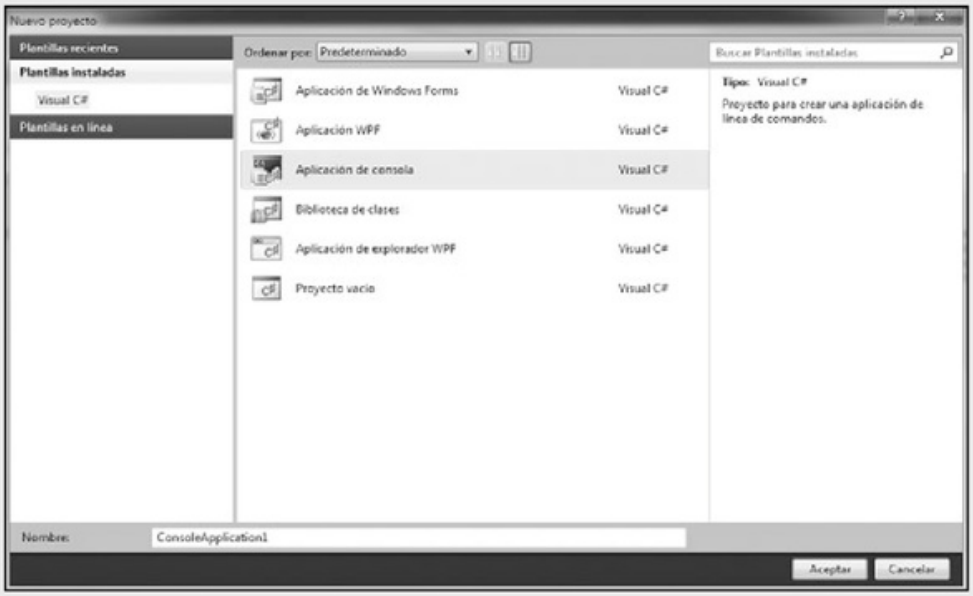
PASO A PASO

1 Abra Visual C# 2010 Express y presione en **Nuevo Proyecto**. Esto mostrará la lista de proyectos disponibles.

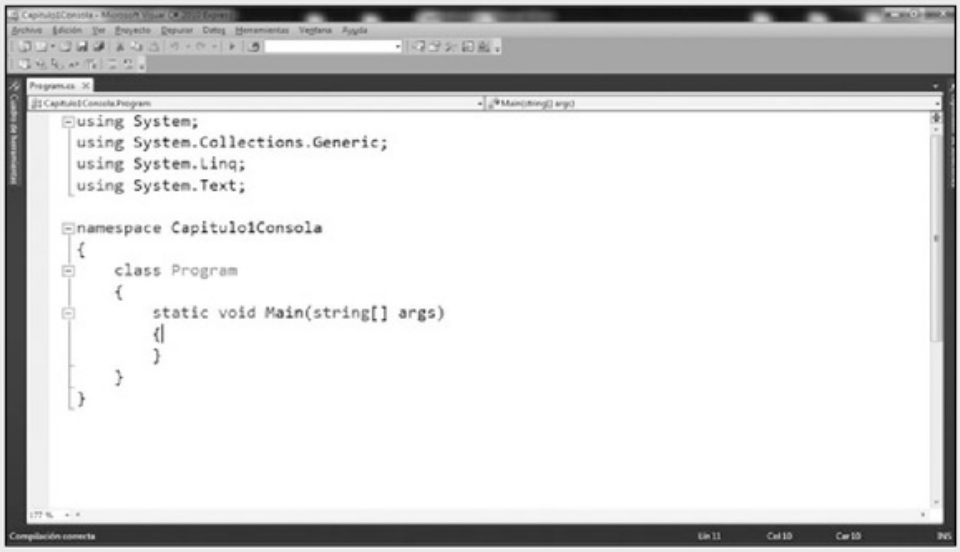
III C# COMO ESTÁNDAR

C# es considerado un estándar por diferentes organizaciones encargadas de mantener y difundir los estándares. Podemos acceder a la información publicada por **ECMA** para C# en **www.ecma-international.org/publications/standards/Ecma-334.htm**. También podemos ver el material de **ISO** sobre C# en **http://standards.iso.org/ittf/licence.html**.

2 El siguiente paso es seleccionar el tipo de aplicación que va a realizar. Para el ejemplo, cree una **Aplicación de consola**. Una vez seleccionada esta opción, coloque un nombre para esta. Presione el botón **Aceptar**.



3 Note que unas cuantas líneas de código son creadas, ahorrándole trabajo. Estas representan la sección principal de la aplicación de consola.



Para completar nuestro primer programa, deberemos escribir algunas líneas de código. Estas líneas no hacen uso directo de C#, sino de los componentes provistos por la librería base de clases. Por lo tanto, sin importar el lenguaje de programación que usemos, estas líneas serán similares. Por este motivo, debemos copiar y pegar el siguiente bloque de código en el resultado mostrado por Visual C# 2010 Express:

```
static void Main(string[] args)
{
Console.WriteLine("Hola Mundo!");
Console.ReadLine();
}
```

Si bien hemos dicho que estamos usando elementos de la librería base de clases, y que los distintos lenguajes escribirían estas líneas de forma similar, estas tienen algunas particularidades relacionadas con el lenguaje de programación seleccionado, en este caso C#, que veremos a continuación.

En C# el uso de mayúsculas y minúsculas para los nombres es importante; si por equivocación escribimos un nombre en minúsculas cuando es esperado en mayúsculas, obtendremos un error al no ser encontrado o considerarse que este no existe. Por otro lado, toda línea de código debe ser finalizada con el símbolo **;** (punto y coma), en especial aquellas que son líneas únicas.

También es importante notar el uso de **{** (llave abierta) y **}** (llave cerrada) para demarcar el inicio y el final de un método o función. En los siguiente capítulos, nos enfocaremos más en la sintaxis de C#. El código debería verse similar al de la **Figura 6**.

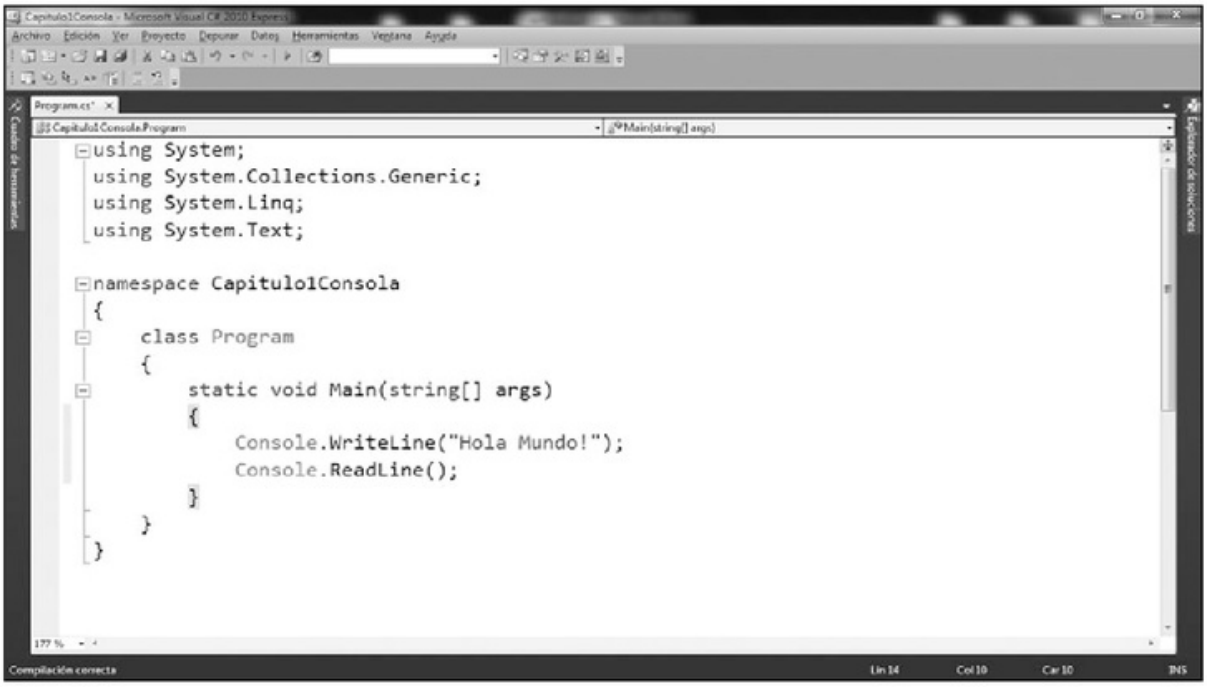


***Figura 6.** Estructura final de código de acuerdo con la implementación de nuestra primera aplicación de consola en C#.*

Por último, si presionamos la tecla **F5**, dispararemos dos acciones. Por un lado, la acción de compilar nuestra aplicación y al mismo tiempo de ejecutar el código resultante. Si necesitamos solo compilar nuestra aplicación, podemos usar la

combinación de teclas **CONTROL+SHIFT+B**, lo que hará que solo se compile, pero que no se ejecute el programa. Si hemos seguido correctamente los pasos, deberíamos ver un resultado similar al de la **Figura 7**.

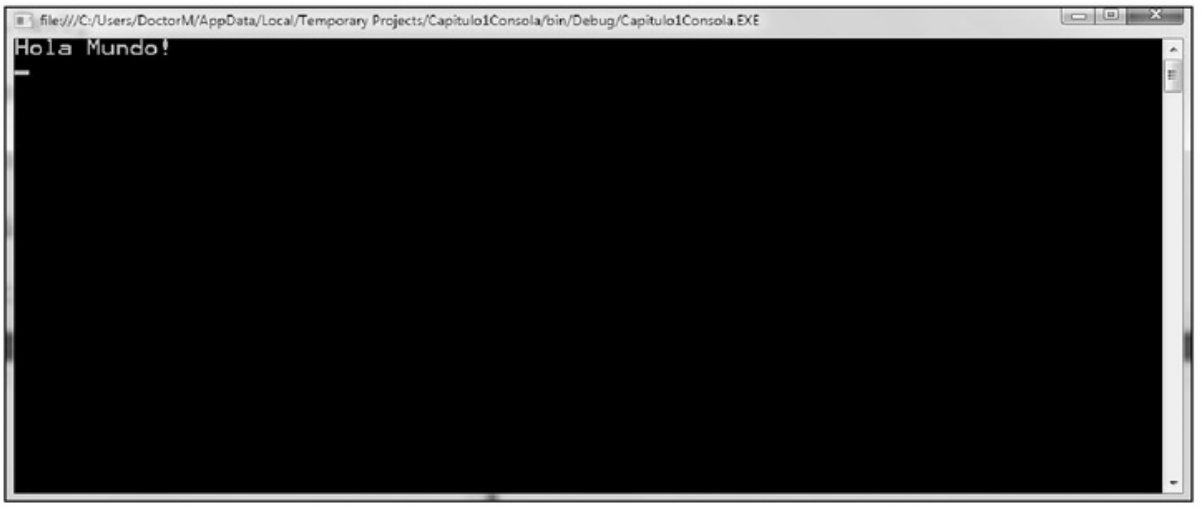


Figura 7. *Nuestra aplicación de consola ejecutándose y mostrando el mensaje* ***"Hola Mundo!"****.*

Si presionamos cualquier tecla, daremos por terminada la aplicación y su ejecución se detendrá. Hemos terminado con la introducción a Microsoft .Net Framework.

... RESUMEN

Solo hemos dado una mirada inicial al concepto detrás de Microsoft .Net; hemos conocido algunas razones que nos deberían motivar a elegir esta plataforma de desarrollo y optado por C# como lenguaje de programación basándonos en sus características, y además porque es un estándar internacional. Por último, hemos podido crear nuestra primera aplicación y así ganar experiencia sobre la herramienta de desarrollo que utilizaremos durante este libro. En los siguiente capítulos, aprenderemos paso a paso y en profundidad todo lo necesario para desarrollar nuestros propios programas, por lo que no debemos asustarnos si en este punto tenemos dudas sobre el código o el uso de las herramientas asociadas, ya que iremos creando experiencia a medida que avancemos por el libro.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 Nombre las partes que componen el Microsoft .Net Framework.

2 ¿A qué se hace referencia cuando se habla de código manejado?

3 ¿Indique la diferencia entre IDE y lenguaje de programación?

4 ¿Existe alguna versión de herramientas de desarrollo para Microsoft .Net que pueda usar de forma gratuita? ¿Cuál?

5 Nombre dos ventajas de C# sobre otros lenguajes de programación soportados por Microsoft .Net.

6 ¿A qué hacen referencia las iniciales JIT?

7 ¿Qué es el DLL Hell?

8 ¿Es posible desarrollar software para Linux con Microsoft .Net?

9 ¿Es posible desarrollar aplicaciones para la consola de juegos Xbox 360 con Microsoft .Net?

10 ¿Cuántas versiones de Microsoft .Net existen? ¿Cuál es la última?

EJERCICIOS PRÁCTICOS

1 Intente modificar el mensaje mostrado en la aplicación de consola creada en este capítulo para que se vea un mensaje diferente.

2 Ingrese al siguiente sitio web para afianzar lo visto en este capítulo sobre las partes del Microsoft .Net Framework: http://msdn.microsoft.com/es-es/library/a4t23ktk.aspx.

3 Utilizando Visual C# 2010 Express, intente crear una aplicación Windows y ejecútela.

4 Pruebe las distintas combinaciones de teclas para ejecutar y compilar el software creado.

5 Analice la ventana derecha que se muestra en Visual C# 2010 Express, donde se puede ver la solución y los archivos creados para su proyecto.

Capítulo 2

Introducción a la programación

En este capítulo, aprenderemos todo lo necesario para poder leer, escribir y plasmar nuestras ideas en código que pueda interpretar un ordenador. Además nos introduciremos en uno de los primeros paradigmas computacionales para la creación y estructuración del código.
Será fundamental para todos los lectores que no hayan creado código anteriormente, así como para los que recién se inician en el desarrollo con C#.

PARADIGMAS DE PROGRAMACIÓN

Entendemos por **paradigma** el conjunto de reglas, modelo o patrón planteado. Por lo tanto, es el conjunto de reglas que representan la forma de escribir, formular y estructurar el código de una manera específica y determinada.
En el transcurso de la evolución de los lenguajes de programación y del desarrollo de software mismo, muchos han sido los paradigmas propuestos. Algunos, como norma específica adoptada por distintos lenguajes de programación, y otros, creados y provistos por lenguajes específicos.
Podemos ver una lista de los distintos paradigmas en la **Tabla 1**. Para aprender a programar se ha usado de forma tradicional el paradigma de **programación estructurada**, y luego, al conseguir cierto dominio de la lógica necesaria para escribir programas, se agrega otro paradigma que suele ir de la mano, el paradigma de **programación modular**, para finalmente concluir con uno de los más renombrados, el paradigma de **programación orientado a objetos**.
Nosotros pasaremos por estos tres paradigmas en el mismo orden, con el objetivo de alcanzar un buen nivel en la comprensión de los conceptos de desarrollo de software y en la capacidad de escribir buen código. Sin embargo, es necesario aclarar, para que estemos atentos, que, si bien el camino propuesto y usado normalmente al recorrer estos paradigmas y al aprender a desarrollar software aparenta ser el adecuado, muchas veces, por dificultades en la comprensión o en la enseñanza, en especial del último paradigma nombrado, se termina desarrollando con los dos primeros al creer, de manera errónea, que se está desarrollando bajo el tercero. Por lo tanto, intentaremos destacar estos errores comunes para no caer en ellos cuando desarrollemos código.

PARADIGMA	DESCRIPCIÓN
Programación estructurada	En este modelo de programación, cada línea de código es ejecutada secuencialmente. Cada línea da paso a la siguiente, y la siguiente no puede ejecutarse hasta que la anterior no haya finalizado.
Programación modular	En este modelo de programación, los distintos bloques de código pueden ser separados en grupos, y así aislar posibles rutinas reutilizables desde distintos puntos de nuestro código.
Programación orientada a objetos	La **POO** (Programación orientada a objetos) intenta imitar, a través del código, los comportamientos de objetos reales a nivel computacional; para esto, combina diferentes paradigmas y enfoques de programación (Figura 1).
Programación funcional	El código en este tipo de paradigma está escrito en torno a funciones y algoritmos matemáticos.
Programación declarativa	Los lenguajes basados en este paradigma procuran describir el problema por solucionar a través de sus líneas de código. La suma de dos valores podría ser representado como "**Sumar A más B y alojar el resultado en C**".
Programación orientada a aspectos	Intenta generar una organización más detallada de los diferentes aspectos del software para así poder encarar cada una de estas partes de forma más efectiva. Además, pretende disminuir la disgregación y duplicidad del código.

***Tabla 1.** Lista de paradigmas de programación más conocidos y utilizados.*

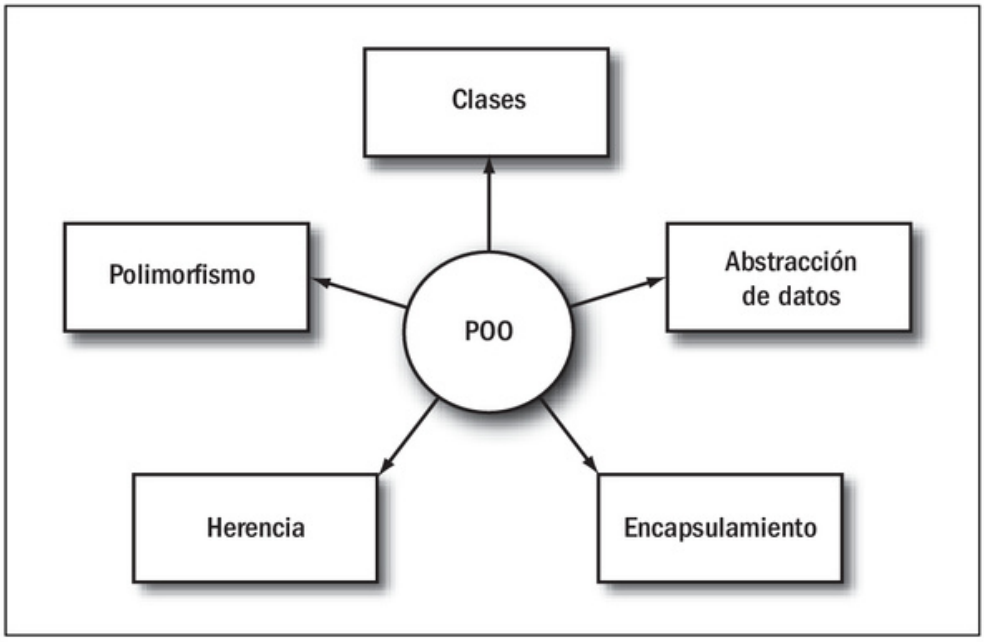


Figura 1. *Elementos de los cuales se nutre y fundamenta la programación orientada a objetos.*

LÓGICA BOOLEANA

Cuando hablamos de **lógica booleana** o **álgebra booleana** nos referimos al planteo realizado por George Boole. Esta propuesta es ampliamente usada tanto en el desarrollo de software como en el de hardware. Pero quedémonos tranquilos, no es nuestro objetivo entrar en complicaciones matemáticas, donde podamos encontrar involucrados conceptos como el de tensiones eléctricas, cableado, chips, o similares, sino que utilizaremos este concepto desde la perspectiva del desarrollo de software, por lo tanto, nos alejaremos del álgebra y nos acercaremos a la lógica, ya que nuestro objetivo es entender cómo funcionan las preposiciones booleanas.
Como veremos durante este capítulo, en el software esta lógica booleana resulta indispensable en más de una ocasión; prácticamente todas las estructuras de control de flujo del código dependen de esta lógica, por lo que dominarla resulta fundamental.

{} GEORGE BOOLE

Es importante que conozcamos algo de historia y en especial sobre los personajes que, con sus estudios y aplicaciones, han dado vida a la computación moderna. George Boole es uno de los padres de la computación moderna. Podemos dar una mirada rápida a su biografía ingresando a la siguiente dirección: **http://es.wikipedia.org/wiki/George_Boole**.

Para entender la lógica booleana, debemos imaginar dos proposiciones. Cada una de estas proposiciones puede tener dos posibles valores: **verdadero** o **falso** para cada caso, y solo uno en cada oportunidad. Teniendo estos valores, podremos realizar diferentes operaciones con las cuales obtendremos un tercer valor que, al mismo tiempo, será verdadero o falso. Este comportamiento lógico podrá aplicarse en cualquier caso donde se requiera un patrón de resolución lógica.
Ya que esta lógica proviene en primera medida de interrupciones eléctricas de hardware, podríamos extender la idea de verdadero o falso a **encendido** para el primero caso, y **apagado** para el segundo, lo que finalmente nos llevará a la base de la computación, los **bits** mismos, que se representan con **1** (encendido o verdadero) y **0** (apagado o falso). En la **Tabla 2**, vemos una lista de las operaciones que podremos realizar aplicando lógica booleana. Cada una de estas operaciones tendrá su correspondiente concordancia en el lenguaje de programación que utilicemos.

OPERACIÓN	DESCRIPCIÓN	EJEMPLO
OR	**OR**, o en castellano **O**, selecciona todos los elementos de ambas partes y retorna todos los valores de cada una de ellas.	Verdadero **OR** (O) falso será igual a verdadero y falso.
AND	**AND**, o en castellano **Y**, analiza las intersecciones de las dos partes dadas y retorna el resultado sobre la base de las coincidencias encontradas entre ambas partes.	Verdadero **AND** (Y) verdadero será igual a verdadero.
NOT	Podríamos entender a **NOT**, o en castellano **NO**, como la negación o exclusión de la selección entre ambas proposiciones o en una única proposición.	Verdadero **NOT** (NO) falso será igual a verdadero. **NOT** verdadero, será igual a falso.

***Tabla 2.** Listado de las principales operaciones booleanas desde la teoría de conjuntos.*

Habiendo entendido lo propuesto en la **Tabla 2**, podemos extender esta idea a las diferentes posibles combinaciones que encontraremos dentro de nuestros programas. En la **Tabla 3**, hallaremos estas combinaciones, pero en notación de bits, esto es, **1** (unos) y **0** (ceros), o encendido y apagado (**Figura 2**).

OPERACIÓN	RESULTADO
0 AND 0	0 (Falso)
1 AND 0	0 (Falso)
1 AND 1	1 (Verdadero)
0 OR 0	0 (Falso)
1 OR 0	1 (Verdadero)
1 OR 1	1 (Verdadero)
NOT 0	1 (Verdadero)
NOT 1	0 (Falso)

***Tabla 3.** Distintas combinaciones de lógica booleana en notación de bits.*

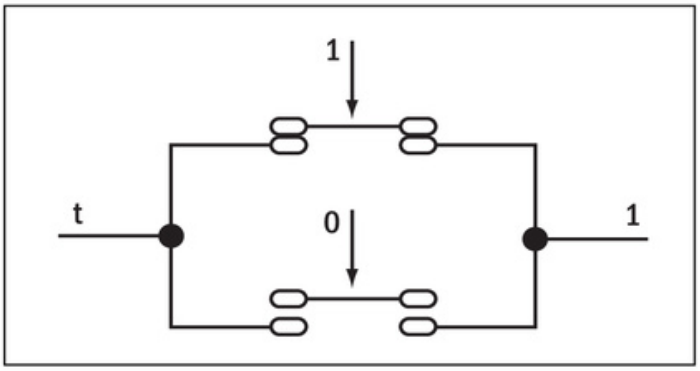


***Figura 2.** Una compuerta lógica OR. Dado un valor de entrada se evalúa sobre la base de dos caminos posibles. Uno verdadero (encendido) y otro falso (apagado).*

Con lo expuesto hasta ahora, ya podemos dar el siguiente paso e implementar esta lógica en código, aunque antes tendremos que hacer una breve parada para asociar estas operaciones booleanas a la sintaxis y palabras reservadas propias de C#.
Como hemos aclarado, cada lenguaje posee su propia sintaxis, así como sus palabras reservadas, las mismas que deberemos respetar para poder escribir código en cada uno de estos. En el caso de C#, las operaciones utilizadas en la lógica booleana tienen su correlación. En la **Tabla 4**, vemos estas correlaciones y un ejemplo de cada una.

OPERADOR BOOLEANO	OPERADOR EN C#	EJEMPLO
AND	&&	Verdadero **&&** verdadero será igual a verdadero.
OR	||	Falso **||** verdadero será igual a verdadero.
NOT	!	**!** Verdadero será igual a falso.

***Tabla 4.** Correlación de la notación booleana con la sintaxis de C#.*

Poner todas las piezas juntas

Teniendo en cuenta lo visto hasta el momento, es válido que llevemos esta teoría a líneas de código (**Paso a paso**). Recordemos que, para realizar los ejemplos, necesitaremos tener ya instalada alguna de las versiones de Visual Studio o en su defecto, si no contamos con la versión paga, Visual C#2010 Express.

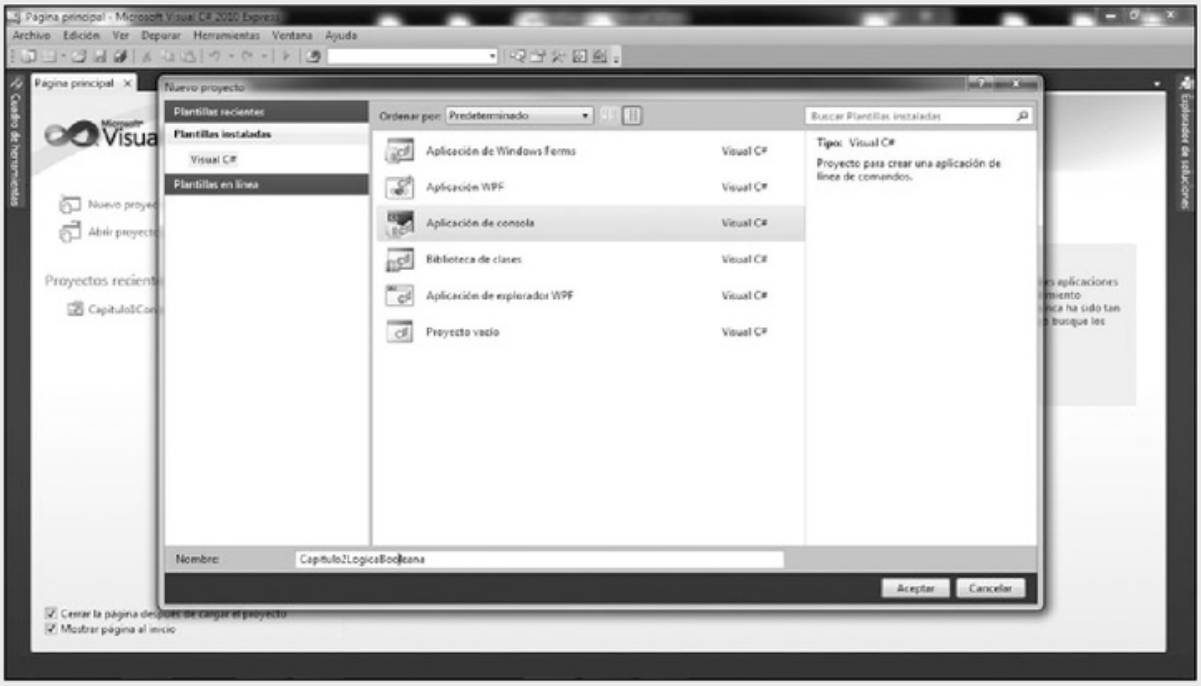
III PALABRAS RESERVADAS

Cuando hablamos de **palabras reservadas** de un lenguaje de programación, hacemos referencia a un conjunto de instrucciones (o palabras) utilizadas por el lenguaje para declarar diferentes funcionalidades las cuales no podremos usar como nombres para nuestras propias implementaciones. Es importante conocer cuáles son para no cometer errores sintácticos.

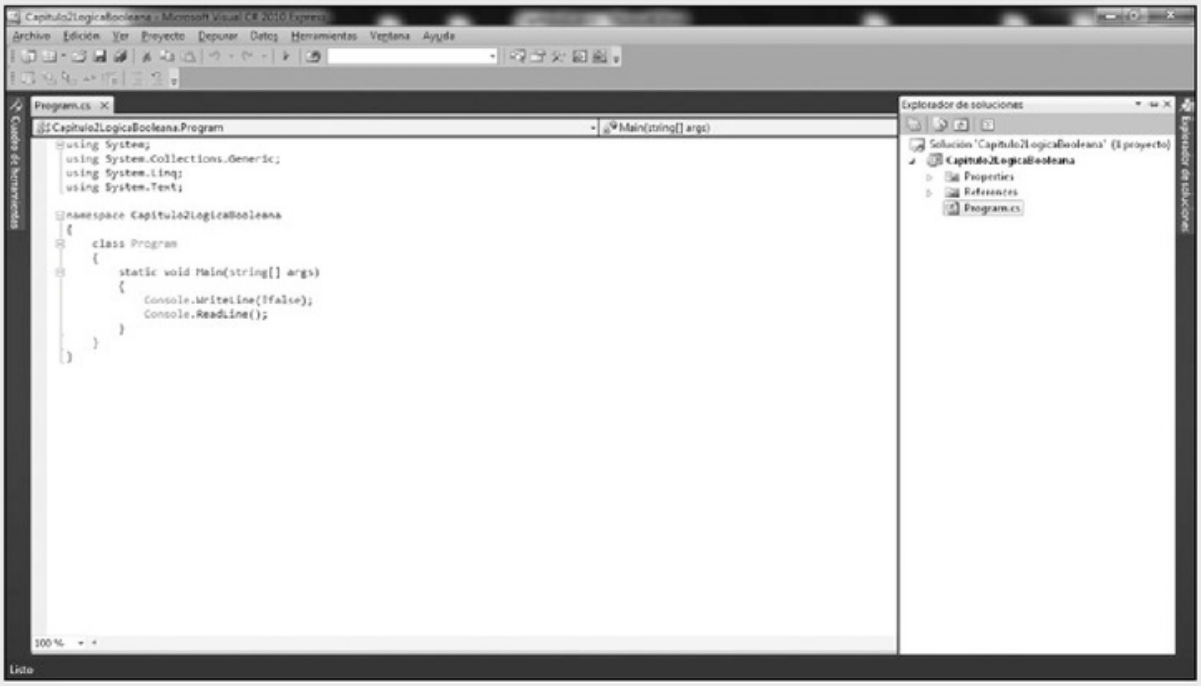
Implementar la lógica booleana

PASO A PASO

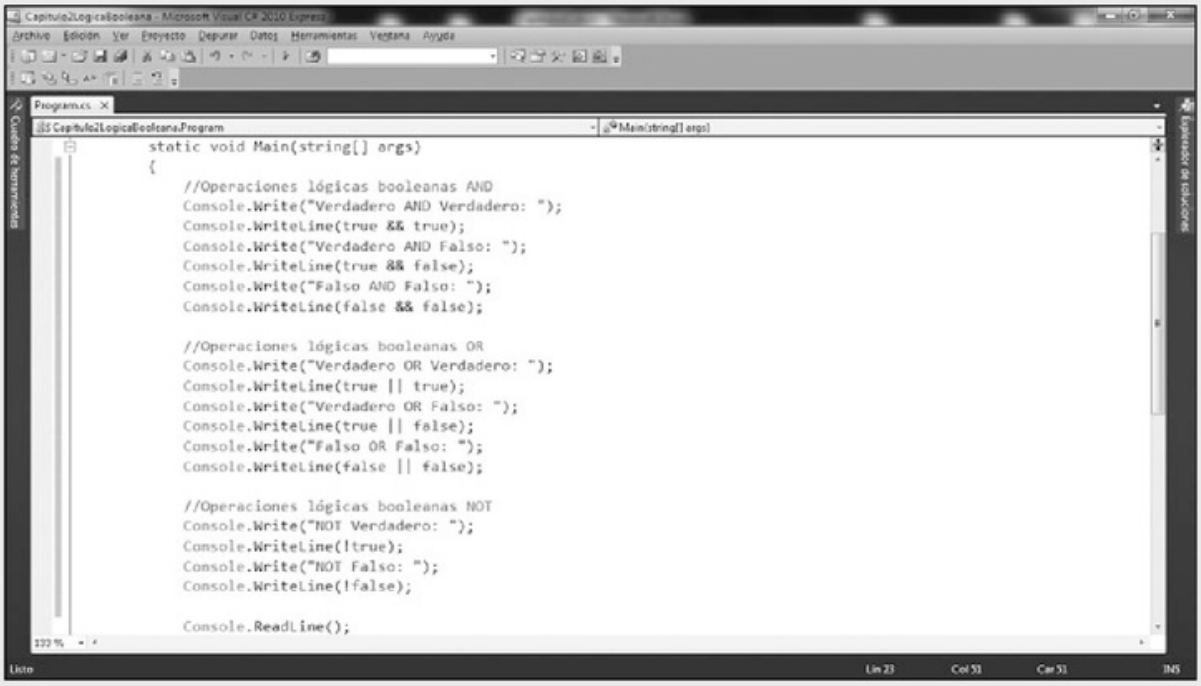
1 Como primer paso, cree una nueva aplicación de consola desde la opción **Nuevo Proyecto**. A continuación, desde la lista de proyectos disponibles, seleccione el proyecto **Aplicación de consola**. Luego presione **Aceptar**.



2 En el explorador de soluciones (izquierda), presione dos veces sobre el archivo **Program.cs** para que este se abra en el editor de código.



3 Para simular las operaciones de lógica booleana, escriba diferentes líneas de código que se apliquen a cada una de las alternativas y escriba sus resultados en la consola, mediante el uso de la línea **Console.write**.



Podemos ver el código de forma más detallada en las siguientes líneas.

```
class Program
{
     static void Main(string[] args)
     {
            //Operaciones lógicas booleanas AND
            Console.Write("Verdadero AND Verdadero: ");
            Console.WriteLine(true && true);
            Console.Write("Verdadero AND Falso: ");
            Console.WriteLine(true && false);
            Console.Write("Falso AND Falso: ");
            Console.WriteLine(false && false);

            //Operaciones lógicas booleanas OR
            Console.Write("Verdadero OR Verdadero: ");
            Console.WriteLine(true || true);
            Console.Write("Verdadero OR Falso: ");
```

```
            Console.WriteLine(true || false);
            Console.Write("Falso OR Falso: ");
            Console.WriteLine(false || false);

            //Operaciones lógicas booleanas NOT
            Console.Write("NOT Verdadero: ");
            Console.WriteLine(!true);
            Console.Write("NOT Falso: ");
            Console.WriteLine(!false);

            Console.ReadLine();
        }
    }
```

Podemos corroborar si hemos implementado correctamente el código anterior comparándolo con lo que es posible observar en la **Figura 3**.

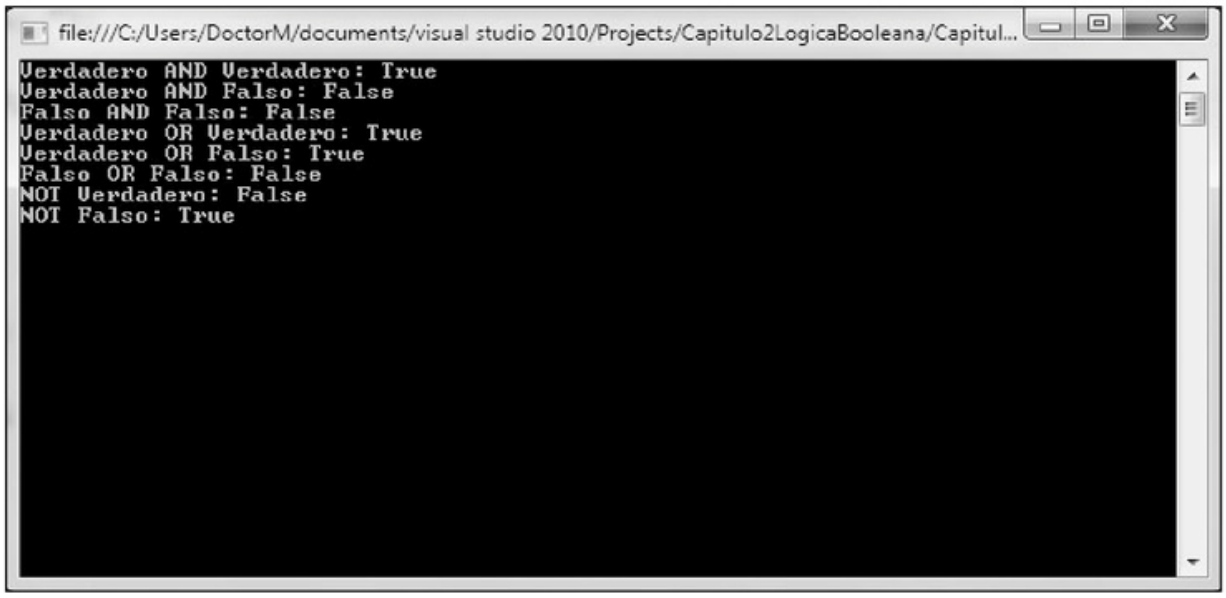


Figura 3. *Salida en la consola de Windows de las operaciones lógicas booleanas implementadas en C#.*

En este ejemplo, hemos agregado también un nuevo elemento; este elemento es el de la doble barra invertida //, el cual representa una línea de comentarios dentro de nuestro código. Esto quiere decir que todo lo que escribamos después de este carácter será tomado como una acotación en el código que no tendrá validez lógica, por lo tanto, no ejecutará ninguna instrucción determinada.

Los comentarios creados a partir del uso del elemento // solo ocuparán una única línea, por lo que, si necesitamos agregar más líneas de comentarios, deberemos

incluir estas barras al principio de cada una de las nuevas líneas de comentarios escritas. También podremos usar **/*** y ***/** para comentar más de una línea.

```
...
//Esto es un comentario de una sola línea
//continuando con otro comentario de una sola línea
...
/*Otro comentario que
ocupará más de una sola línea
y no requiere agregar el carácter de comentario
por cada línea nueva */
...
```

Nótese en las líneas de código anteriores que, en el uso del comentario de muchas líneas, es necesario cerrar el comentario haciendo uso de ***/**.

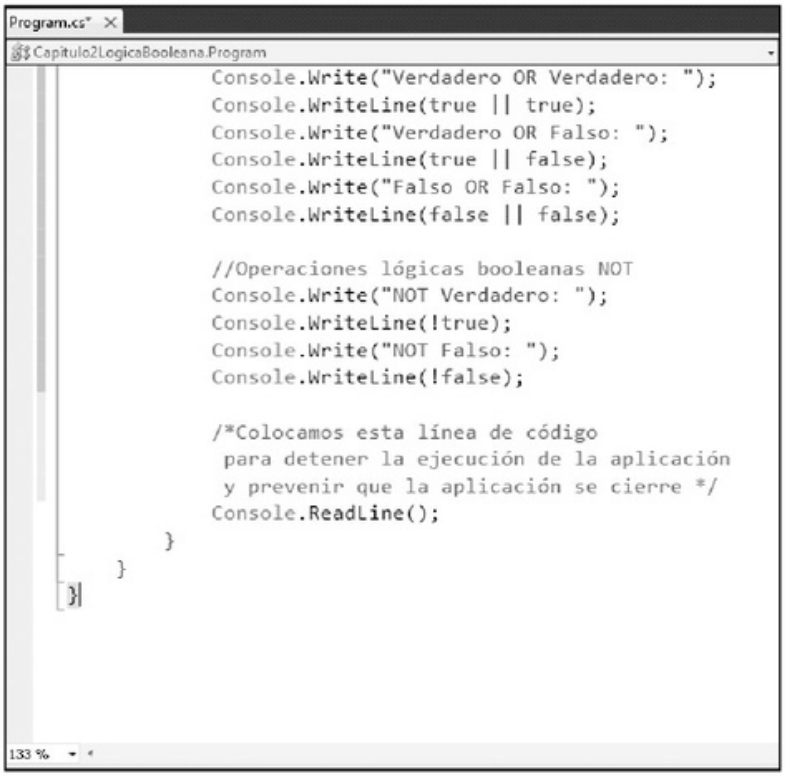


***Figura 4.** Podemos observar el uso de comentarios con múltiples líneas sin la necesidad de caracteres adicionales por cada una de ellas.*

Los comentarios en el código fuente son de gran utilidad al momento de generar documentación sobre el trabajo realizado. Pensemos que otros programadores podrían leer nuestro código para modificarlo o darle mantenimiento, por lo que cualquier evidencia o ayuda sobre su funcionamiento siempre será bienvenida.

Programación estructurada

Al inicio de este capítulo, nombramos los diferentes paradigmas involucrados en la creación de código para computadoras. Uno de estos es el paradigma llamado programación estructurada y, como habíamos propuesto, es el ideal para dar los primeros pasos en la creación de código. Este paradigma se fundamenta en tres pilares lógicos que dan forma a la estructura del programa, su ejecución y definición: secuencial, selección e iteración se conjugan para manipular el flujo de ejecución del programa.

Secuencial

Cuando hablamos de **secuencia** en la programación estructurada, nos referimos a la forma en cómo, cada una de las líneas de código que escribamos se irán ejecutando. Esta secuencia se refiere al orden y secuencia de ejecución. En la programación estructurada, cada línea es ejecutada una a una de forma descendente, no pudiéndose ejecutar la siguiente línea de código hasta que la anterior finalice sus acciones. Si analizamos el código realizado en el ejemplo anterior, notaremos que cada línea de código escrita es mostrada en el mismo orden en la consola de Windows.

```
...
Línea 01 Console.Write("Verdadero AND Verdadero: ");
Línea 02 Console.WriteLine(true && true);
Línea 03 Console.Write("Verdadero AND Falso: ");
...
```

En el caso anterior, primero se escribirá en la pantalla la frase **"Verdadero AND Verdadero: "**, y una vez escrito el mensaje se evaluará la proposición booleana, para luego mostrar el mensaje siguiente. La numeración de las líneas ilustra el orden de ejecución esperado, por lo que no es necesario escribirlas en el código.
La ejecución secuencial también garantiza la integridad del código, permitiendo tener la seguridad de que aquello que esperamos se ejecute después de que alguna otra operación lo haga, y no que nuestro código se ejecute de forma aleatoria. Este pilar también es especialmente importante cuando se involucra el uso de

III VERDADERO Y FALSO

La mayoría de los lenguajes de programación utilizan como idioma para la definición del conjunto de instrucciones, el inglés. Por lo tanto, cuando hablamos de verdadero o falso, estos términos deberán ser traducidos a su equivalente en el idioma inglés, por lo que verdadero será igual a **true** y falso igual a **false**.

métodos y funciones en nuestro código. Hablaremos en particular de los métodos y las funciones al final de este capítulo.
Para terminar de entender este primer pilar, debemos mencionar que el concepto de secuencia, así como la programación estructurada en su conjunto, es comúnmente asociado al uso de los diagramas de flujo por el carácter secuencial en la ejecución de las tareas y procesos.
Estos diagramas son bastante fáciles de entender y muestran, justamente de forma secuencial, la realización de acciones unidas, cada una de estas, por flechas de dirección que irán guiándonos por cada uno de los procesos hasta completar el diagrama.
Podemos ver, entonces, en la **Figura 5**, el mismo código planteado anteriormente y su secuencia, desde la visión de los diagramas de flujo.

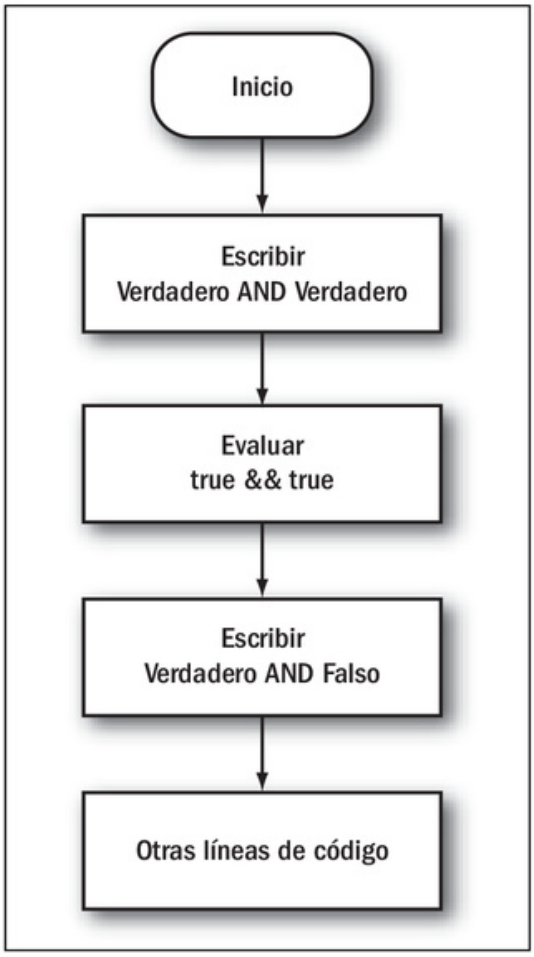


***Figura 5.** Diagrama de flujo que muestra una porción del código que evalúa una expresión booleana y evidencia su resultado en la consola.*

Selección

Cuando hablamos de **selección** dentro de la programación estructurada, hacemos referencia a la capacidad de elegir entre dos posibilidades, por lo tanto, estaremos seleccionando un camino u otro. Así, cuando nuestro código es ejecutado línea a línea, secuencialmente, podremos crear bifurcaciones en él para que ciertas partes se

ejecuten o no sobre la base de alguna condición preestablecida. Para poder crear estas condiciones siempre se utiliza lógica booleana, la que hemos visto previamente. Además del conjunto de operaciones booleanas que ya nombramos, se agregan otras como las de comparación e igualación. En la **Figura 6**, es posible observar la representación de este pilar en un diagrama de flujo.

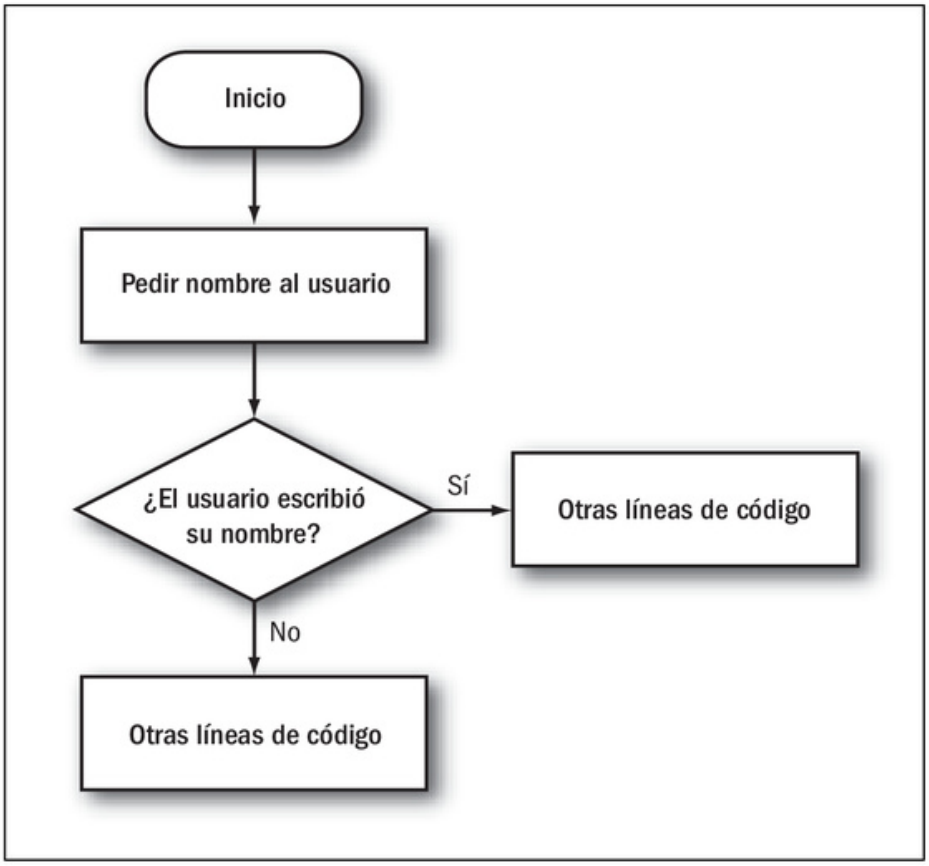


***Figura 6.** El diagrama muestra la selección y bifurcación en caminos alternos sobre la base de una condición específica.*

Como pudimos ver en la **Figura 6**, una bifurcación de este tipo tendrá dos partes: una de ellas representará las acciones o líneas de código en el caso de que la condición sea afirmativa, esto es, se cumpla, y otra, en caso de que esta no sea alcanzada. Luego de que la bifurcación es resuelta, el código seguirá su flujo normal.

INICIOS DE LA PROGRAMACIÓN ESTRUCTURADA

El paradigma de la programación estructurada ve la luz en la década del 60 gracias a Böhm y Jacopini. Al mismo tiempo, uno de los lenguajes bastiones de la programación estructurada fue, posiblemente, el conocido como **Basic**, el mismo que requería que el programador enumerara cada línea para su posterior ejecución.

Iteración

El último pilar de la programación estructurada es la **iteración**. Este se refiere a la capacidad de interactuar con un mismo conjunto de líneas de código de forma repetida por una cantidad de veces definida o por medio de una condición especificada como en el caso de la selección. Este mecanismo es especialmente útil en ocasiones donde necesitemos contar elementos, sumar números, pedir repetidamente información al usuario, o mostrar listas de datos, justamente, iterándolas. En la **Figura 7**, vemos el diagrama de flujo que hace referencia a esta capacidad.

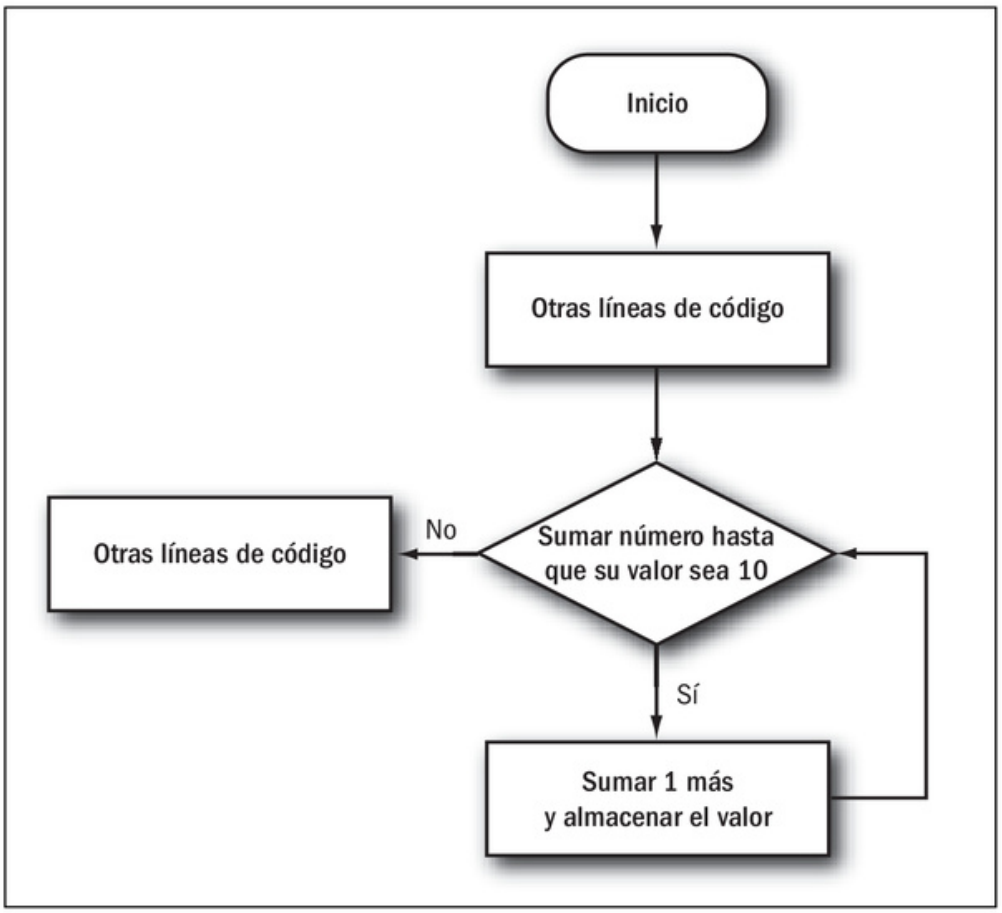


***Figura 7.** En los diagramas de flujo, las iteraciones se representan con un símbolo de condición, cierta lógica de acción y una vuelta a la evaluación de la condición.*

III DIAGRAMAS DE FLUJO

Si no conocemos los diagramas de flujo y deseamos entender mejor lo propuesto en las diferentes figuras que se presentan en este capítulo, podemos tomar un curso rápido ingresando a la siguiente dirección: **http://mis-algoritmos.com/aprenda-a-crear-diagramas-de-flujo**. En este sitio, se muestran las principales figuras y cómo utilizarlas al escribir diagramas de flujos.

En la **Figura 7** también se muestra cómo se sumará un número hasta que su valor sea igual a 10. En el momento en que la condición deje de cumplirse, la iteración llegará a su fin y se continuará con la ejecución normal del código y todo el programa. Es necesario recalcar que, en la programación estructurada, existen diferentes modelos de iteración, fijémonos la diferencia planteada en el diagrama de la **Figura 8**.

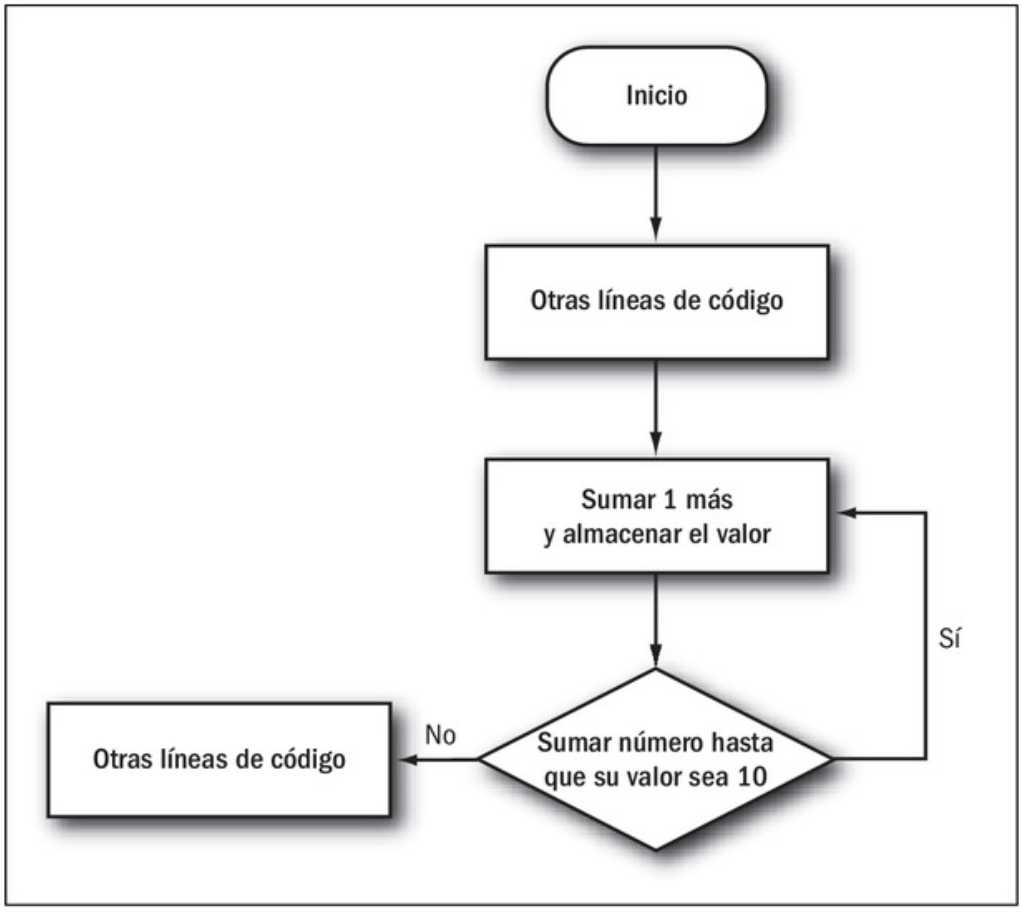


Figura 8. *El diagrama muestra una segunda implementación de una iteración donde, primero, se ejecuta algo de código y, luego, se evalúa una condición para iterar sobre el mismo código mientras la condición se mantenga.*

En el caso de la iteración propuesta en la **Figura 8**, vemos que la acción de sumar 1 a un número hipotético se realizará por lo menos una vez, para luego analizar si la condición se cumple; si el número aún no llegó a valer 10, entonces se volverá a ejecutar la suma. Por el contrario, si el número ya supera el esperado en la condición, entonces el flujo normal del código será ejecutado siguiendo con otras líneas de nuestro programa, finalizando definitivamente el ciclo de interación.
En la programación computacional existen distintos modelos de iteración. Los más conocidos son los que se asocian a las instrucciones **For** y **Do While**, las que podríamos asociar con las figuras **7** y **8** respectivamente; pero encontraremos, sobre la base del lenguaje que utilicemos, otras formas de iterar líneas de código y elementos de nuestro programa. Veremos en detalle estas formas a lo largo de este capítulo.

Variables

Cuando estemos creando un programa basado en código, durante su ejecución, será necesario tener un lugar donde podamos almacenar la información que el programa requiera de forma temporal para luego poder volver a consumir dicha información. Este lugar son las **variables** y, como su nombre lo dice, pueden mutar, cambiar, durante el transcurso de la ejecución del programa, y contener distintos valores, tanto numéricos como cadenas de texto, y colecciones de datos, entre otros.
Para entender la utilidad de las variables, volvamos un poco sobre lo dicho en este capítulo cuando veíamos la iteración en la programación estructurada. Por cada iteración, el valor por comparar era sumado en 1 para luego analizar si la suma total era equivalente a 10. Esta acción requería que se almacenara, en algún lugar, el resultado de la suma para luego poder compararlo, por lo tanto, el lugar ideal es una variable. A continuación, podremos ver esta representación por medio de pseudocódigo:

```
...
Otras líneas de código
Definir la variable resultado y asignarle el valor 0
Si resultado es menor a 10 entonces repetir
     sumar 1 a resultado
Fin de la repetición
Otras líneas de código
...
```

Como podemos ver en el pseudocódigo anterior, primero se define una variable con el nombre de **resultado** y se le asigna el valor de **0**. Luego, en la iteración se pregunta si dicha variable sigue siendo menor a 10. Mientras esta condición se cumpla, se repetirá la línea de código interna que pretende sumar **1** a la variable **resultado**. Al concluir esta línea, se evaluará la condición anterior, para verificar si se cumple. Si la variable **resultado** inicialmente contenía el valor de **0**, en la primera iteración esta contendrá **1**, luego **2**, ya que el valor inmediatamente anterior ya no será 0, sino 1, y la suma de otro 1 arrojará el valor 2, y así hasta que llegue a 10 y finalice.

III PSEUDOCÓDIGO

No es un lenguaje de programación específico, en todo caso, es un conjunto de frases que un programador puede utilizar para representar las acciones esperadas de un programa. Para explicar el funcionamiento de un programa entre desarrolladores, es escrito en pseudocódigo como una forma de ahorrar tiempo y facilitar el entendimiento de aquello que se quiere crear.

Esta capacidad para contener información tiene su impacto en la memoria de la computadora; esto quiere decir que el espacio físico que ocupa una variable, la cantidad de bits que esta representa y su contenido serán alojados en la memoria del ordenador para utilizarlos más adelante.
Por lo tanto, debemos ser cuidadosos en el uso de estas, ya que un uso excesivo e indiscriminado podría causar problemas en el normal funcionamiento de la computadora. Por supuesto, las computadoras modernas cuentan con grandes cantidades de memoria, lo que ha posibilitado crear, entre otras características de las mismas computadoras, programas cada vez más grandes, pero sigue siendo necesario tener cuidado al momento de usar las variables. Las variables pueden ser de diferentes tipos; cada tipo representa la clase de valor que esta puede contener. En la **Tabla 5**, podemos ver la lista de tipos de variables disponibles en C# y su nomenclatura.

TIPO C#	TAMAÑO EN BITS	RANGO
byte	8	0 a 255
sbyte	8	-128 a 127
short	16	-32768 a 32767
ushort	16	0 a 65535
int	32	-2147483648 a 214748647
uint	32	0 a 4294967295
long	64	-9223372036854775808 a 9223372036854775807
ulong	64	0 a 18446744073709551615
bool	8	True o False
char	16 de texto Unicode	0 a 65535

***Tabla 5.** Tipos de variables y su nomenclatura en C#.*

Como podemos ver en la **Tabla 5**, existen variables para contener significativas cantidades y maneras de información, ya sean valores numéricos, como caracteres únicos como es el caso de **char**, o para representar los dos posibles resultados arrojados en las operaciones booleanas mediante la variable **bool**.
Cada uno de estos tipos de variables son utilizadas por los programadores basándose en los datos que pretenden alojar. Así, una aplicación que requiera contener un número

ERROR EN COMA FLOTANTE

En el año 1994, Intel produjo una serie de procesadores, los Intel Pentium de 66 MHz, en los cuales se detectó un error en el cálculo de coma flotante. Así, un simple cálculo matemático realizado desde un lenguaje de programación donde se involucraban números grandes haría que el procesador arrojara un resultado erróneo.

significativo de valores negativos y positivos, posiblemente usaría una variable de tipo **long**; en el caso de los ejemplos vistos con los diagramas de flujo, donde solo contábamos hasta 10, la variable posiblemente hubiera sido del tipo **byte** o **short**. Hacer un buen uso de las variables hará que nuestra aplicación no consuma memoria de forma innecesaria y por lo tanto funcionará de manera óptima. De cualquier forma, aún existen más variables con características adicionales, veamos la **Tabla 6**.

***Figura 9.** Al ejecutar nuestra aplicación con las variables definidas, esta asigna algunos valores por defecto dependiendo del tipo que hayamos elegido para ellas.*

TIPO C#	TAMAÑO EN BITS	PRECISIÓN	RANGO
float	32	7 dígitos	-3.4 x 1038 a + 3.4 x 1038
double	64	15-16 dígitos	±5.0 x 10?324 a ±1.7 x 10308
decimal	128	28-29 lugares decimales	(-7.9 x 1028 a 7.9 x 1028) / (100 a 28)

***Tabla 6.** Tipos de variables y su nomenclatura en C#.*

En el caso de la tabla anterior, la **tabla 6**, podemos ver un conjunto de variables para manipular números enormes. Además, estas brindan la posibilidad de la manipulación de decimales, algo que las anteriores no podían.
Este tipo de variables posiblemente sean comunes en aplicaciones bancarias, donde la precisión en los decimales por cada transacción resulta de carácter vital.

Crear variables

Ahora que ya sabemos cuáles son los tipos de variables disponibles, cuánta memoria requieren, y qué tipo y rango de datos pueden manejar, es momento entonces de entender cómo manipularlas dentro de C#. Lo primero que debemos hacer para

poder trabajar con variables es definirlas; la definición necesita del tipo de variable que queramos crear, el nombre de ella y un valor inicial opcional, este último, como hemos dicho, puede o no estar presente.
Veamos el siguiente código que crea una variable del tipo booleana **bool**, una para almacenar números enteros **int** y una para enteros largos **long**.

```
//Creamos una variable booleana
bool miVariableBooleana;

//Creamos una variable entera
int miVariableEntera;

//Creamos una variable entera larga
long miVariableEnteraLarga;
```

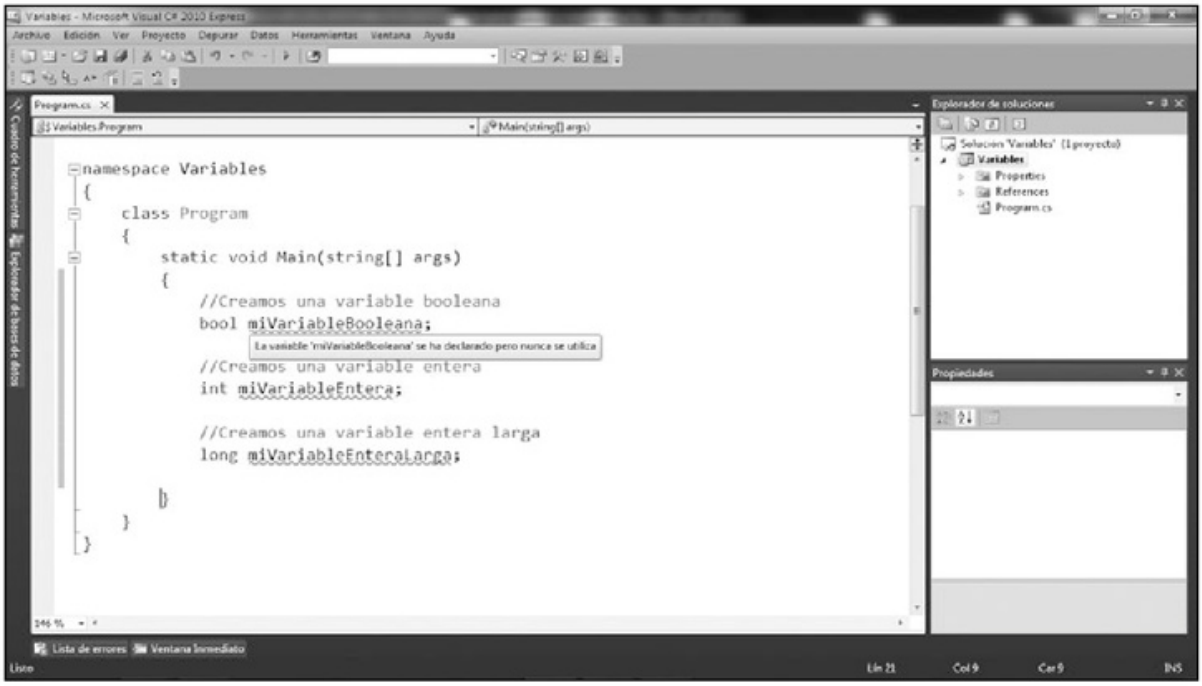


***Figura 10.** Las variables que han sido declaradas, pero no usadas o asignadas con algún valor son analizadas por Microsoft Visual C# 2010 Express y se nos avisa esta situación para que las eliminemos o usemos.*

En C#, así como en los lenguajes con una raíz en C, el tipo de variable es explicitado delante del nombre de la variable por crear. Por otra parte, las variables no pueden comenzar con números o caracteres especiales. El símbolo de **guion bajo** (_) puede ser utilizado para comenzar el nombre de una variable. Además, la notación que usemos para escribir estas variables, por ejemplo, el uso de mayúsculas o minúsculas debe ser respetado, ya que C# discriminará entre ambas.

```
//Creamos una variable booleana
bool MiVariableBooleana;

//Creamos una segunda variable booleana
bool miVariableBooleana;
```

Como vemos en el ejemplo de código anterior, las dos variables se leen de la misma forma. En nuestro idioma, ambas son leídas sin diferencias, pero C# considerará que estas dos variables son diferentes y podrán contener distintos valores independientes debido a que una inicia con mayúsculas y la otra con minúsculas.

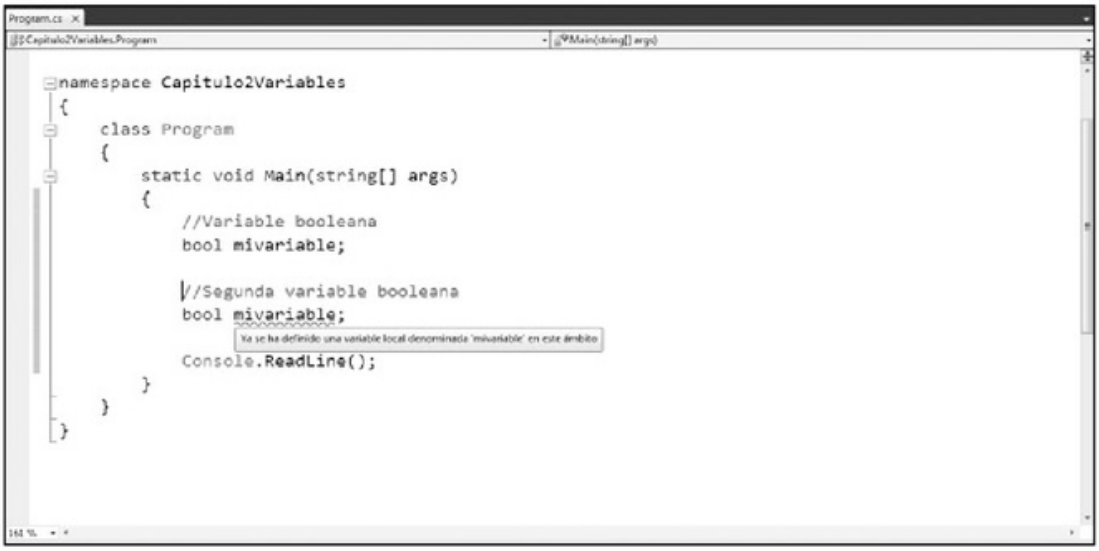


Figura 11. *Si escribimos dos variables que tengan el mismo nombre y la misma nomenclatura, obtendremos un error por duplicidad. No obstante, podremos escribirlas con el mismo nombre, pero no respetando mayúsculas y minúsculas.*

Asignar variables

Ya hemos creado diferentes variables, pero no le hemos asignado ni recuperado valores de ellas. Recordemos que, en el momento de crear una nueva variable, es posible asignarle un valor inicial y que esto es opcional, básicamente debido a que en nuestro código, podremos asignarle valores nuevos o modificar los ya existentes. De cualquier manera, de no asignar un valor inicial, esta tarea quedará a cargo del motor de ejecución y, por lo tanto, no estaremos seguros de cuál será dicho valor. Con esto, nos referimos a que una variable podría contener un número específico, o un valor nulo, o algún otro valor que cause un mal funcionamiento en nuestro programa. Por lo tanto, si no queremos asignarle un valor en el momento de su creación, es recomendable que lo hagamos antes de comenzar a usar esta variable para asegurarnos de que su contenido será el que esperamos. La asignación de valores a variables se realiza por igualación.

Esto quiere decir que deberemos escribir, en nuestras líneas de código, una proposición similar al siguiente modelo: teniendo la variable **A**, esta debe ser igual al número **20**. Para esto usamos el signo **=** (igual). Veamos el siguiente código.

```
bool B = true;
int A = 20;
int C = 30;
```

En el caso anterior, hemos creado tres variables, y en el momento de crearlas le asignamos el valor inicial. Así, la variable **B** será igual a **true** (verdadero), la variable **A** será igual a **20**, y **C** igual a **30**. Para comprobar esto, podemos mostrar el contenido de estas variables en la consola de Windows completando nuestro código con las siguientes líneas.

```
Console.WriteLine(B);
Console.WriteLine(A);
Console.WriteLine(C);
```

El código resultante debería ser similar al de la **Figura 12**.

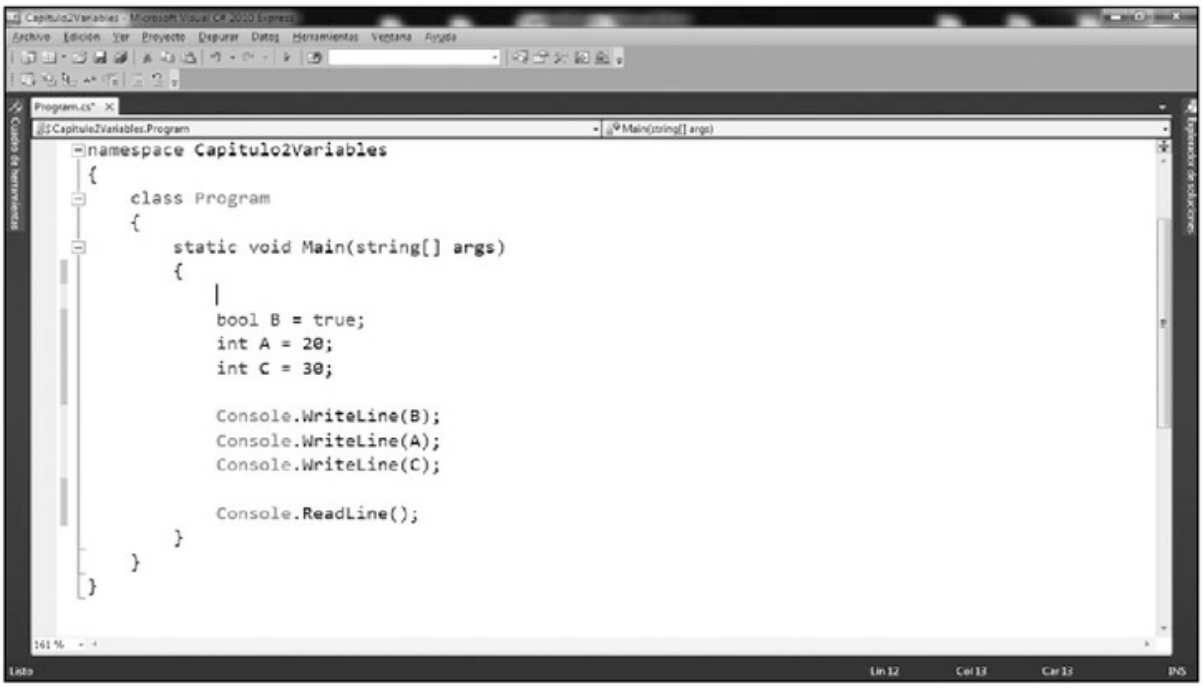


***Figura 12.** En el siguiente código, vemos cómo se crean tres variables y, en el mismo momento, se les asignan valores. Luego se muestran en la consola.*

Como podremos comprobar, la consola nos mostrará los valores **True**, **20** y **30**, respectivamente, lo vemos representado en la **Figura 13**.

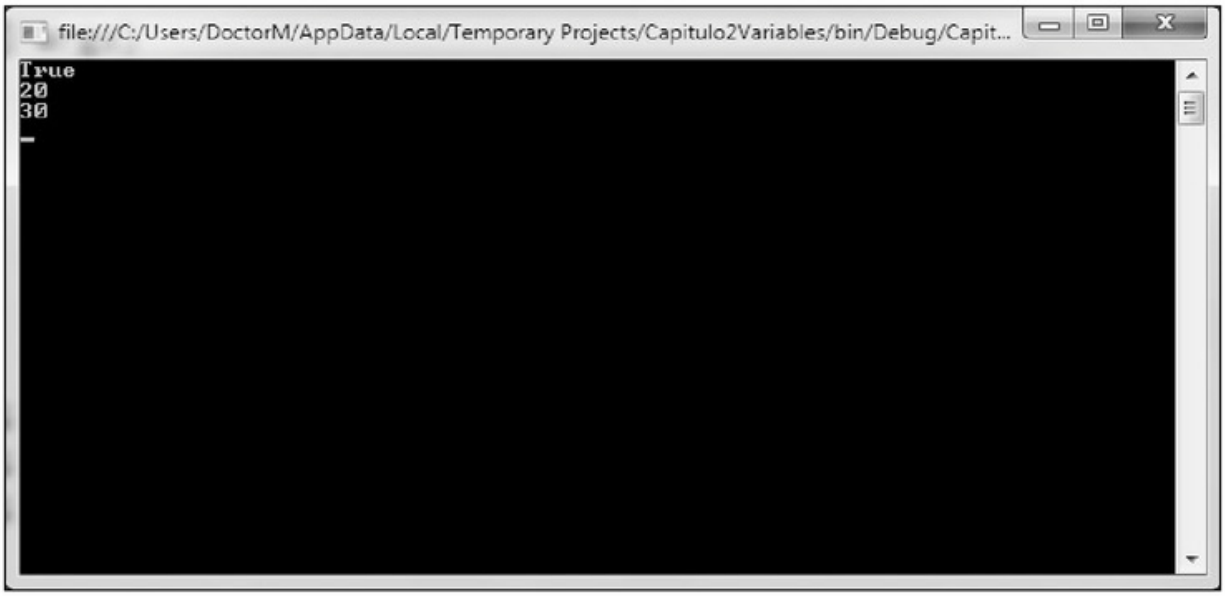


Figura 13. *En la ventana de la consola podemos ver impresos los valores de las variables, en este caso,* ***True****,* ***20*** *y* ***30****.*

No solo es posible asignar valores a una variable mediante su inicialización o en el transcurso del código. También podemos asignar valores a una variable desde otra variable, por lo tanto, teniendo dos variables, mediante una igualación, es posible que una tome los valores de la otra en el proceso.
Esto sucede siempre y cuando sean del mismo tipo o, en su defecto, que el valor por asignar pueda ser contenido por la variable a la cual se le asignará.
Así, un valor entero **int** podrá ser asignado a un valor largo **long**, ya que este último tiene mayor capacidad que el primero, pero no podrá hacerse a la inversa si el valor contenido dentro de la variable **long** supera el máximo permitido en **int**.
Agreguemos algunas líneas a nuestro código, en este caso, para asignar los valores de la variable **C** a la variable **A**. Veamos el ejemplo, a continuación:

```
A = C;
Console.WriteLine(A);
```

{ } SINTAXIS DE LOS LENGUAJES

Cada lenguaje de programación posee su propio conjunto de palabras clave, así como la sintaxis que delimita la forma en la que deberá ser escrito el código. Por esta causa, los distintos lenguajes de programación podrán presentar pequeños cambios en su escritura, dependiendo de la raíz del lenguaje, hasta cambios radicales que provoquen diferencias completas entre ellos.

Como habíamos dicho, la igualación mediante el uso del signo igual hará que los valores del lado derecho sean asignados a la variable del lado izquierdo. Si imprimimos el contenido de la variable **A** después de la asignación, veremos que esta contiene el mismo valor que **C** (**Figura 14**).

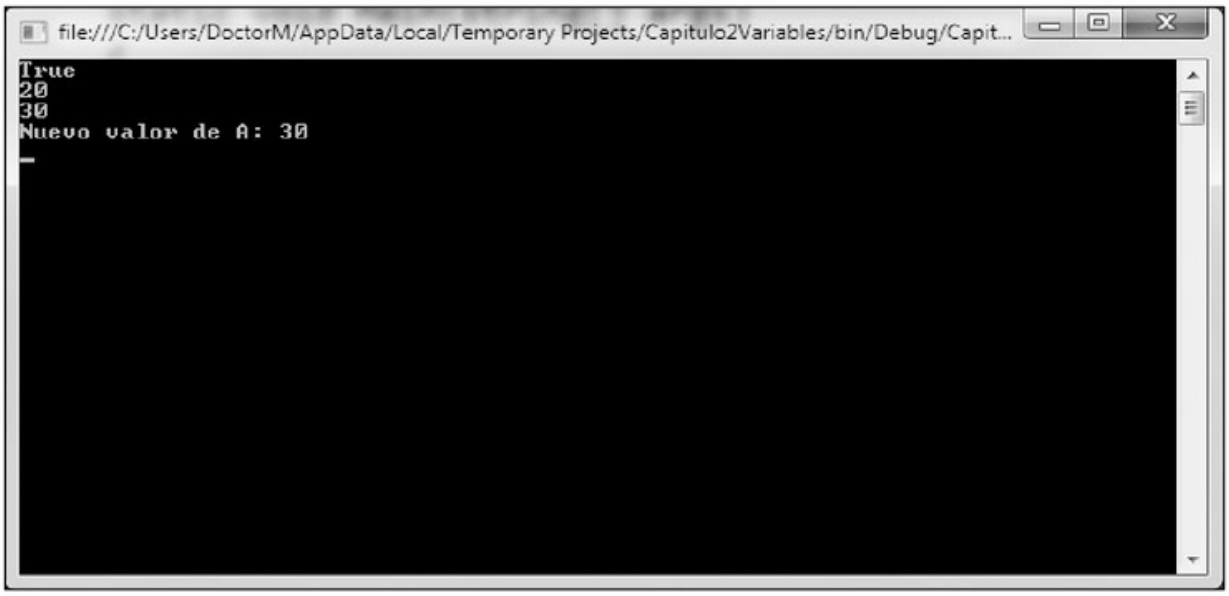


Figura 14. *En la consola podemos ver el nuevo valor de la variable A. Inicialmente 20 y luego de la asignación 30.*

Operaciones con variables

Además de asignar valores de una variable a otra, es posible realizar operaciones con ellas. Dependiendo del tipo de variables con la que estemos interactuando, podremos hacer uno y otro tipo de operación. Por ejemplo, las variables con capacidades de almacenar números serán sujeto de operaciones aritméticas como la suma, la resta, divisiones y demás. Sobre las variables booleanas, se podrán realizar operaciones booleanas y, sobre las que manejen cadenas de texto, tendremos disponibles operaciones que manipulen dicho texto.

```
int A;
int B;
int resultado;

A = 10;
B = 20;

//Sumamos A y B y su resultado se almacena
//en la variable RESULTADO
resultado = A + B;
```

```
Console.Write("Resultado de la operación: ");
Console.WriteLine(resultado);
```

En el código anterior, el cálculo de suma se realiza utilizando el operador de **suma** (+), obteniendo como resultado la suma de las dos variables y su impresión en la consola (**Figura 15**). De la misma forma, otros operadores como la división, multiplicación y demás, son válidos, al mismo tiempo que los cálculos respetarán las agrupaciones de operaciones con el uso de paréntesis. Esto último resulta de vital importancia, ya que la lógica aplicada a la aritmética también será válida en el contecto del desarrollo de código.

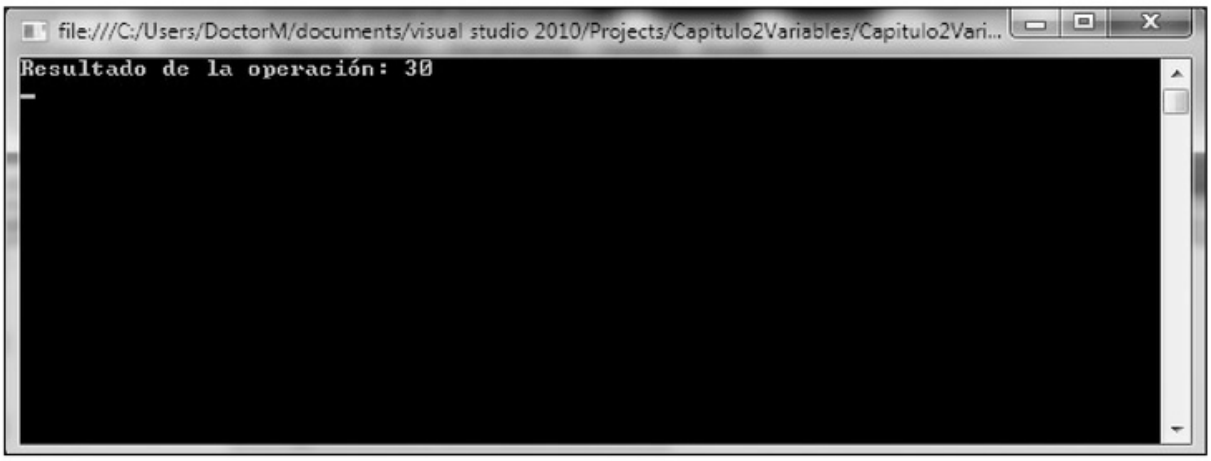


Figura 15. *Se muestra, en la consola, el resultado de la suma de las variables A y B; como resultado obtenemos el valor 30.*

En el siguiente código, vemos cómo podemos realizar más operaciones basadas en la agrupación de funciones. De la misma forma, las operaciones serán resueltas en base a los principios aritmétricos de los cálculos.

```
int resultado = 0;
resultado = (10 + 20) + 2 * (5 + 10);
Console.WriteLine(resultado);

resultado = 10 + 20 + 2 * 5 + 10;
Console.WriteLine(resultado);
```

En el código podemos ver que las operaciones realizadas son iguales, salvo por la forma en cómo están agrupadas sobre la base de los paréntesis utilizados. En la **Figura 16**, podemos ver cómo la simple remoción de los paréntesis hace que el resultado sea diferente, en la primera cuenta aparece el valor **60,** y en la segunda **50**.

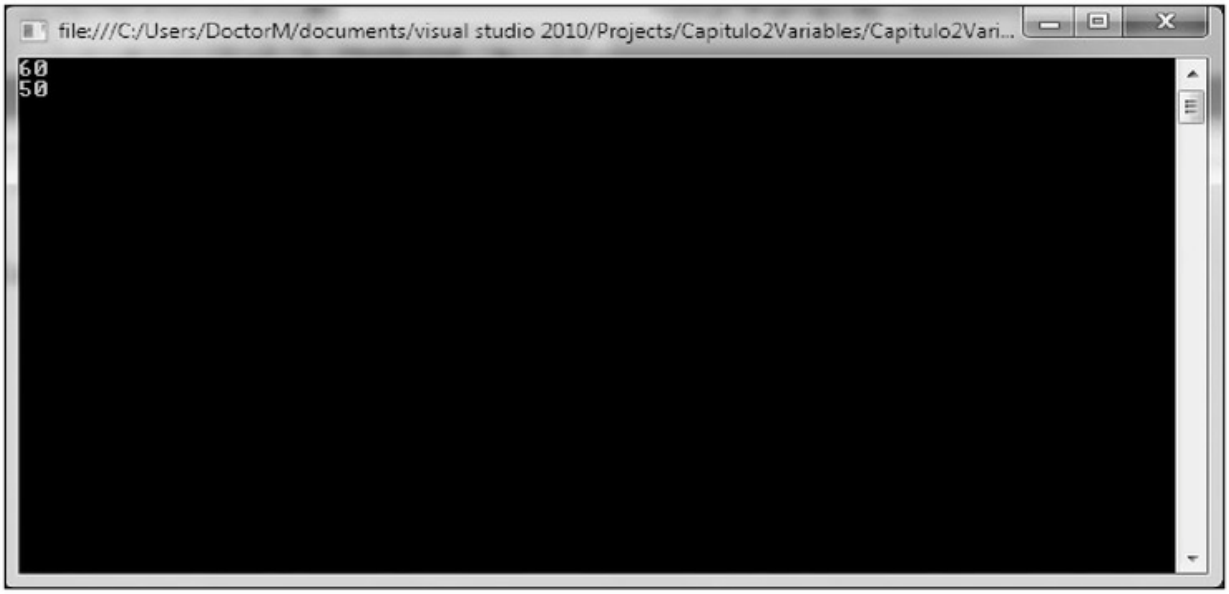


***Figura 16.** El uso de paréntesis puede alterar el resultado debido a la forma en cómo se agrupan las operaciones.*

A diferencia de lo que pudiéramos conocer de aritmética, nosotros no usaremos el símbolo de **[ ]** (corchetes) o **{ }** (llaves) para realizar agrupaciones mayores. Estos caracteres están reservados para otro tipo de operaciones, por lo que, si necesitáramos mayor nivel de agrupamiento, deberíamos usar más paréntesis.
En la **Tabla 7** que se encuentra a continuación, vemos la lista de operadores disponibles para cálculos aritméticos y sus diferentes atajos para el lenguaje C#.

CATEGORÍA	OPERADORES
Aritméticos	+, -, *, /, %
Concatenación de cadenas de texto	+
Incremento y decremento	++, –
Asignación	=, +=, -=, *=, /=, %=
Lógicos booleanos	!

***Tabla 7.** Lista de operadores aritméticos, de manipulación de cadenas de texto y lógica booleana.*

Existen muchos otros operadores disponibles, que iremos viendo a medida que los necesitemos. Por el momento, estos serán suficientes para poder realizar diferentes cálculos, además de manipular cadenas de texto. Notaremos, por otra parte, que muchos de estos operadores parecieran estar repetidos, por ejemplo, el símbolo + es aplicable para sumar números y además, para unir cadenas de texto. Esto se debe a que estos operadores están sobrecargados, lo cual quiere decir que, para el mismo operador, se ejecutarán diferentes códigos, dependiendo del tipo de variable que se use, numérica o de texto. Por lo tanto, debemos tener en cuenta no confundir las funciones de los diferentes operadores, siempre es bueno tener alguna tabla de referencia cerca.

Para el primer caso, cuando sumemos dos números se ejecutará la acción de sumar, pero cuando sumemos dos textos se ejecutará la acción de unir estas cadenas. En la misma **Tabla 7**, notamos algunos operadores curiosos, como el **++** y los símbolos aritméticos seguidos del símbolo **=**. Estos pueden ser considerados atajos en la escritura de código. Para el primer caso, la suma consecutiva hará que al valor contenido en la variable se le adicione 1, o se le quite 1 en el caso de usar el símbolo menos. Para el segundo caso, el operador aritmético seguido del igual logrará que se realice la operación propuesta basada en el argumento seguido del igual. Así, si queremos sumar dos variables, no será necesario escribir la suma en la forma **resultado = resultado + A**, se puede realizar simplemente **resultado += A**.

```
int A;
int B;
A = 0;
B = 10;

Console.Write("Valor de A: ");
Console.WriteLine(A);
A++;
Console.Write("Valor de A++: ");
Console.WriteLine(A);

A += B;
Console.Write("Valor de A: ");
Console.WriteLine(A);
```

En la **Figura 17**, podemos ver el resultado de las operaciones de las líneas de código anteriores. En el primer caso, visualizamos la variable A contiene el valor de **0**; al aplicar el operador **++** e imprimir su contenido, vemos que contiene **1**, luego, al realizar **A += B**, sumamos el contenido de **B** al contenido actual de **A**, obteniendo como resultado final el valor de **11**, como vemos a continuación.

III LOS BITS

Para entender el modelo computacional, su funcionamiento, así como la transmisión de la información dentro de la arquitectura de los ordenadores, es necesario saber qué son los **bits**. En la siguiente dirección podremos encontrar una descripción sobre estos: **http://es.wikipedia.org/wiki/Bit**.

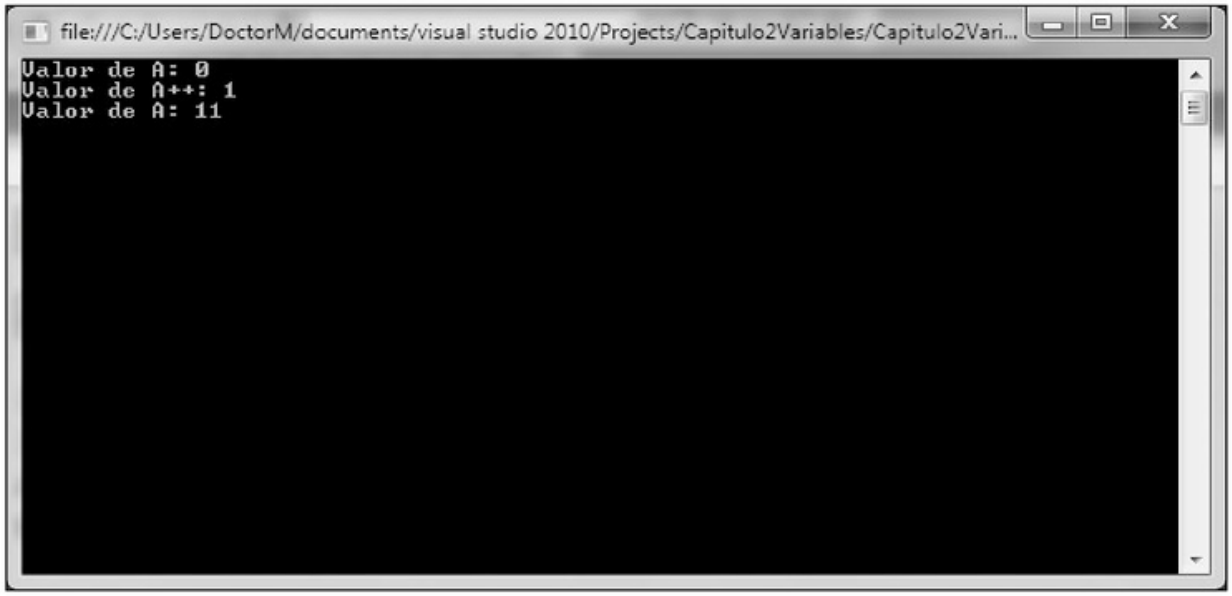


Figura 17. *En la ventana de la consola, podemos ver los distintos resultados de las operaciones realizadas sobre la variable A.*

Por otra parte, los operadores **++** y **--** realizarán el cálculo aritmético en diferentes momentos dependiendo si se escribe antes o después de la variable, por lo tanto, obtendremos diferentes resultados si escribimos **++A** o **A++**.
En el primer caso, si lo aplicamos en un cálculo compuesto, estaremos provocando que primero se sume 1 al contenido de **A**, y después se realicen las otras posibles operaciones. Si por el contrario colocamos el operador como lo hemos hecho hasta ahora, primero se ejecutarán las operaciones contenidas en todo el cálculo y luego se sumará 1 al contenido de la variable.

```
int A;
int B;
A = 0;
B = 10;
Console.WriteLine(A++ + B);
Console.WriteLine(++A + B);
```

III VARIABLES BOOLEANAS

Una buena práctica, si necesitamos modificar el estado de una variable booleana basada en ella misma es utilizar el operador de negación sobre ella en vez de asignar los valores **true** o **false**. Escribimos la asignación de la siguiente forma **A = !A**, donde **A** es una variable booleana.

```
Console.WriteLine(A);
```

En la **Figura 18** podemos ver el resultado del código anterior.

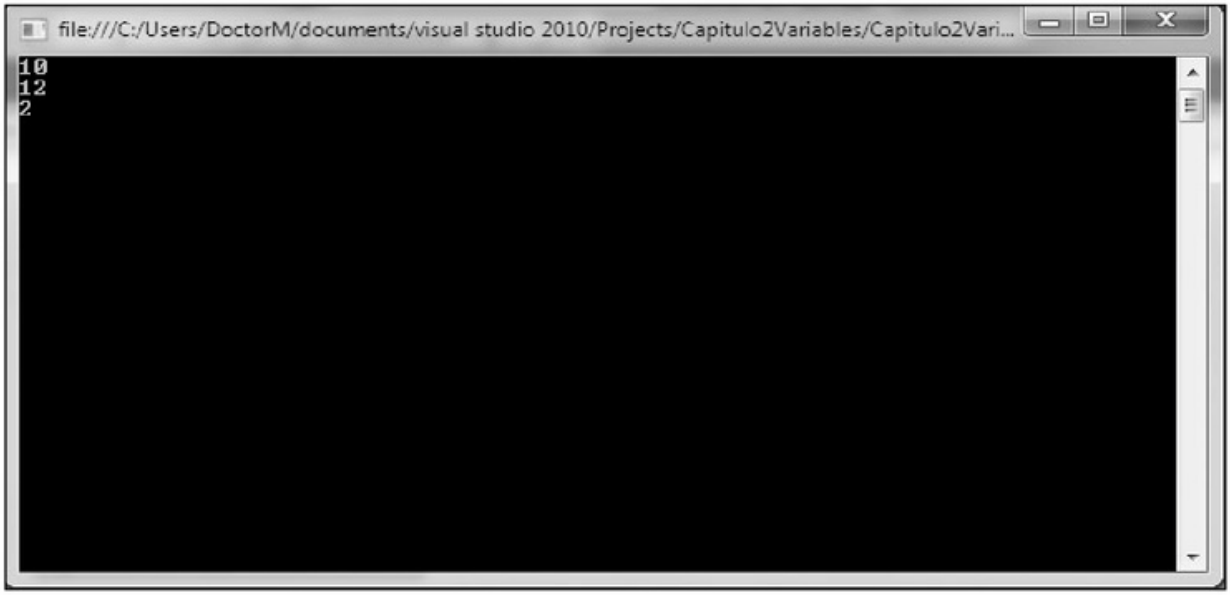


Figura 18. *Dependiendo de dónde coloquemos el operador, obtendremos resultados significativamente diferentes.*

En la primera operación **A** es igual a **0**, y por lo tanto, el resultado es igual a **10** (0 + 10), luego de esta operación se adiciona **1** a **A**. En la segunda operación, **A** ya vale **1**, y antes de realizar el cálculo se le suma **1**, y luego los **10** contenidos en **B**. Como resultado obtenemos **12**. El valor final de **A**, como es de esperarse, será igual a **2**.
El último operador de la **Tabla 8** es uno que ya habíamos visto cuando hablamos de operaciones booleanas. Este operador puede, por consiguiente, ser utilizado en distintas operaciones con variables. Supongamos que tenemos dos variables booleanas; estas contienen un valor **verdadero** (**True**) y necesitamos que, al asignarse una de las variables a la otra, su valor se transforme en **falso** (**False**).

```
bool A = true;
bool B = true;
A = !B;
Console.WriteLine(A);
```

Recordemos que el operador **!** representa una negación lógica. En este caso, al usar este operador en una asignación entre variables booleanas obtendremos el valor contrario al contenido en dicha variable. En este caso, **A** será igual a **falso** a pesar de que **B** es **verdadero**, como vemos en la **Figura 19** que se encuentra a continuación:

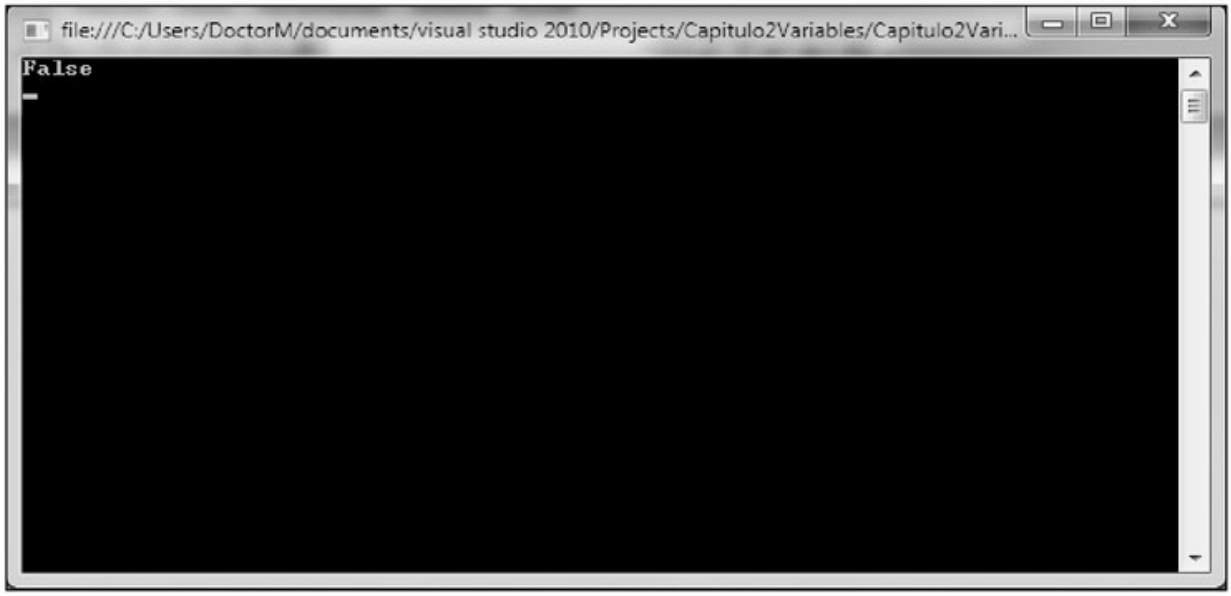


Figura 19. *Al realizar la asignación mediante una negación, la variable A contendrá el valor de la variable B, pero en su forma negada.*

Variables de texto

El último conjunto de variables son las variables capaces de contener cadenas de texto o caracteres. Ya hemos visto, en la lista de variables, el tipo **char**, junto a esta adicionaremos el tipo **string**. Las trataremos de forma diferente a las anteriores ya que tienen un comportamiento distinto, incluso a nivel de memoria. La primera solo puede contener un único carácter, mientras que la segunda puede contener múltiples caracteres. Asimismo, podemos considerar las variables de tipo **string** como una secuencia de caracteres, por lo que es posible acceder a cada uno de sus elementos como si se tratasen de caracteres o variables **char**. Otra de las características de las variables del tipo **string** es su inmutabilidad, esto quiere decir que cada vez que realicemos una operación sobre ellas, ya sea eliminando, adicionando o uniendo otras cadenas de texto, nuevas variables serán creadas para contener este nuevo resultado. A diferencia de las variables que hemos visto hasta ahora, la modificación de sus valores solo alteraría su contenido y no el espacio que ocupan en la memoria.

Por el contrario, como una variable del tipo **string** puede variar en su contenido necesitando más memoria para almacenar nuevos caracteres, la única posibilidad de

{} BUCLES INFINITOS

Aunque un bucle infinito puede causar problemas de ejecución en la aplicación, el desarrollo de videojuegos usa esta técnica para no finalizar el juego abruptamente. En cada ciclo del bucle, la pantalla es redibujada; las teclas, analizadas, y los demás periféricos manejados. La diferencia radica en cómo se manipula internamente.

lograr esto es que el motor de ejecución busque nuevos espacios disponibles en la memoria para alojar esta información. Imaginemos que la memoria es una hoja cuadriculada de un cuaderno; cada cuadradito representa un espacio de ella. Al crear una nueva variable del tipo **short**, esta requerirá de **16** de esos cuadraditos en una misma fila, o que tengan cierta continuidad. Cada uno de estos cuadraditos representa un bit, el mismo que puede contener los valores 0 o 1. Las diferentes combinaciones de 1 y 0 darán la posibilidad de alcanzar el rango que veíamos en la **Tabla 5**. Un **string**, por el contrario, requerirá por cada carácter nuevo una serie de estos espacios de memoria, pero la memoria está ocupada de forma aleatoria por otros datos, por lo que, si tenemos una variable **string** que contiene una sola letra y justo después, en la memoria, se encuentra una variable con otros datos, la primera no podrá expandirse, salvo que esta sea descartada y se busque un nuevo espacio libre en la memoria donde quepa toda la información (**Figura 20**).

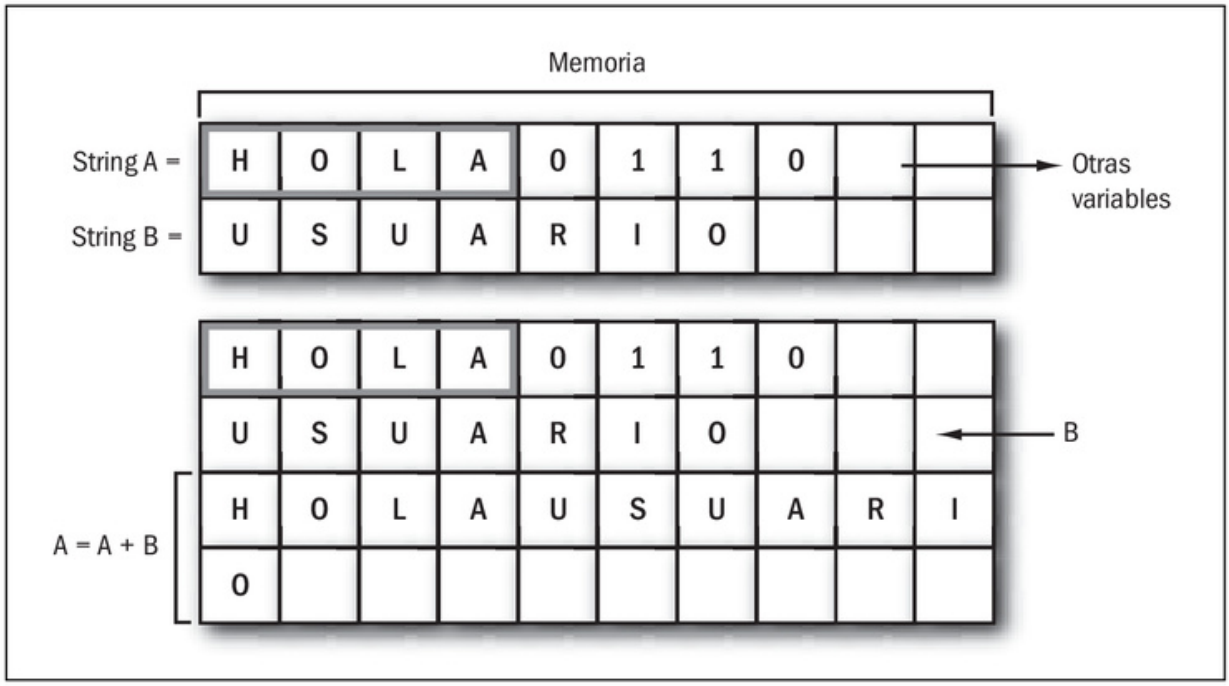


***Figura 20.** Cada variable necesita de una cantidad de memoria para alojar sus datos. Los strings presentan un comportamiento diferente: cada vez que hacemos una acción con ellos una nueva memoria debe ser alojada.*

III STRINGBUILDER

Microsoft .Net provee funcionalidades adicionales para el manejo de variables con comportamientos no tradicionales. Los **strings** son uno de estos tipos. Su inmutabilidad hace que el mal manejo de ellos pueda incidir en problemas de memoria. El objeto **StringBuilder** provee, entonces, funcionalidad para suplantar el uso de **string** en casos donde pueda ocasionar problemas.

Como vemos en la **Figura 20**, al tener dos variables **string**, **A** y **B**, que contiene cada una de estas una cadena de texto, cada espacio de la memoria es ocupado. En el ejemplo, otras variables podrían estar ocupando espacios contiguos, por lo que al unir el contenido de las dos variables en la variable **A**, es necesario buscar más espacio en la memoria que pueda contener dicha información.
Por lo tanto, la referencia que se tenía de la memoria donde estaba contenida la información de **A** es dejada para referenciar a la nueva sección de la memoria que contendrá el resultado de la unión. Como podemos darnos cuenta, este trabajo resultará costoso para el ordenador, al mismo tiempo que requerirá de memoria libre dejando también memoria ocupada, aunque no esté siendo usada por ninguna parte de nuestro programa. Por lo tanto, resulta una buena práctica en el desarrollo de software el no abusar de las transacciones realizadas con cadenas de texto, en este caso, con variables del tipo **string**. Imaginemos este movimiento de memoria si agregáramos diez, cincuenta, mil o un millón de cadenas de texto a la variable **A**: quedaría mucha basura en la memoria sin que nadie la usara. Por suerte, Microsoft .Net Framework posee dos mecanismos para ayudarnos a solucionar en parte este problema.
Por un lado, encontraremos el **Garbage Collector**, el cual se encargará de limpiar la memoria de datos que no están siendo utilizados por nuestra aplicación, y, por otro lado, el objeto **StringBuilder**, que nos permite manipular cadenas de texto, pero con memoria reservada previamente. Por lo tanto, si necesitamos adicionar más información a nuestra cadena de texto, esta tendrá el espacio en la memoria reservado para tal fin y no habrá que hacer los malabares que hemos visto hasta ahora. Hablaremos más del objeto **StringBuilder** en el **capítulo 5**.

CARÁCTER	DESCRIPCIÓN
\n	Nueva línea
\t	Tab
\0	Valor nulo
\r	Retorno de carro
\\	Carácter de barra invertida
\"	Carácter de comillas
\'	Carácter de comillas simples

***Tabla 8.** Lista de caracteres especiales en las cadenas de texto.*

Iniciemos con el tipo de variable **char**. Esta variable representa un único carácter. En la **Tabla 8**, vemos una lista de caracteres especiales para representar comportamientos que, de otra forma, no podrían ser representados en el texto. Así, encontramos el carácter **/0**, el mismo que representa un valor vacío, nulo.
Este carácter, es escrito al final de una cadena de texto para representar su límite. Así como veíamos cómo se almacenaban las cadenas de texto en la memoria, cuando se produce alguna operación sobre esta, como por ejemplo contar su largo, es

necesario recorrer cada uno de los caracteres involucrados en dicha cadena, por lo que, si no contáramos con algún símbolo especial que nos avisara cuándo esta cadena ha finalizado, podríamos contar con toda la memoria.

```
char nuevaLinea = '\n';
string mensaje1 = "Línea superior";
string mensaje2 = "Línea inferior";
Console.Write(mensaje1);
Console.Write(nuevaLinea);
Console.Write(mensaje2);
```

Existe una diferencia entre las variables **char** y **string**; en el primer caso, la asignación de datos hacia estas variables debe realizarse englobando el contenido por asignar entre comillas simples. Estas comillas simples dan a entender que estamos hablando de un único carácter. A diferencia de las variables **string**, las cuales pueden contener muchos caracteres, estas engloban el texto por asignar entre comillas dobles. Al mismo tiempo, notemos que la variable **char** contiene un carácter especial; este es el de la nueva línea, por lo que, al escribir cada una de las variables en la consola, notaremos que en vez de escribir los dos mensajes en una única línea, cada texto será escrito en una línea diferente. Y esto se debe a que estamos escribiendo, entre los dos mensajes, una nueva línea mediante el uso de **\n**.

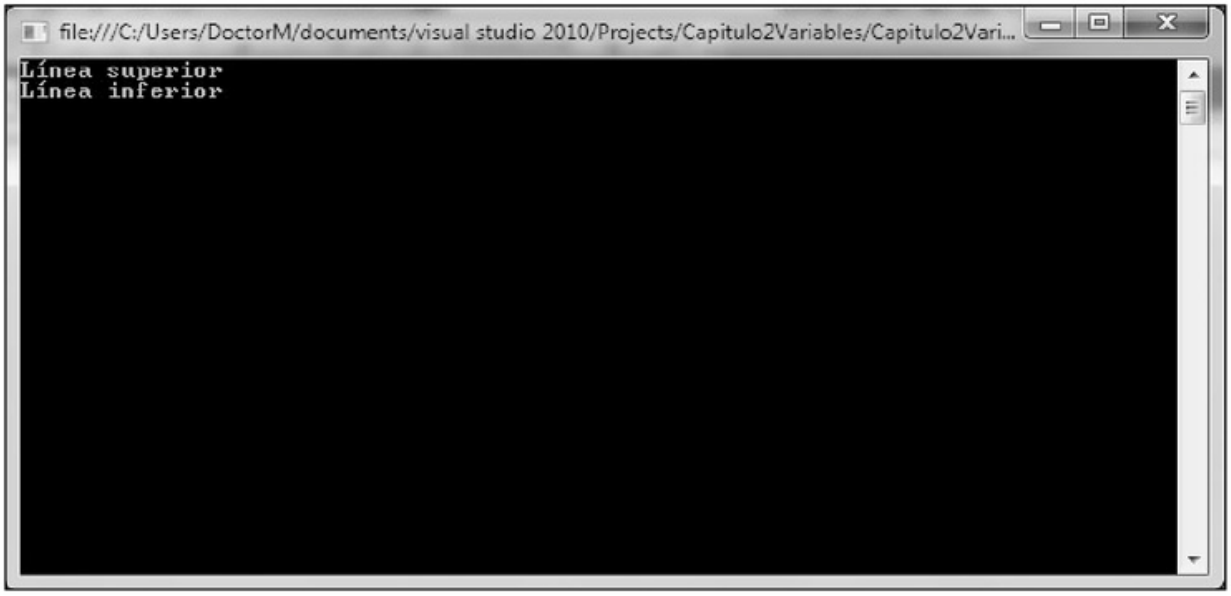


Figura 21. *Una variable del tipo char que contiene el carácter de salto de línea crea un salto de línea en la salida por consola.*

Estos caracteres también pueden ser utilizados en las cadenas de texto. Recordemos que una cadena de texto **string** es considerada una sucesión de **char**, por lo tanto,

un carácter de retorno de carro, nueva línea o comillas puede ser aplicado dentro de la misma cadena. Así, vemos cómo la línea de texto se divide en dos renglones debido al carácter de nueva línea (**Figura 22**).
Para esto, debemos escribir la cadena de texto completa, entre comillas, y dentro de ella, lo distintos caracteres especiales, también llamados, caracteres de escape, como podemos ver a continuación.

```
string mensaje = "Texto en la primer línea\nTexto en la segunda línea";
Console.WriteLine(mensaje);
```

Logrando el mismo comportamiento del ejemplo anterior donde escribimos, por un lado, la primer línea del mensaje seguido de un caracter de retorno de carro y a continuación de la segunda línea. En la **Figura 22**, podemos ver que aparece en la consola "**Texto en la primer línea**" y "**Texto en la segunda línea**".

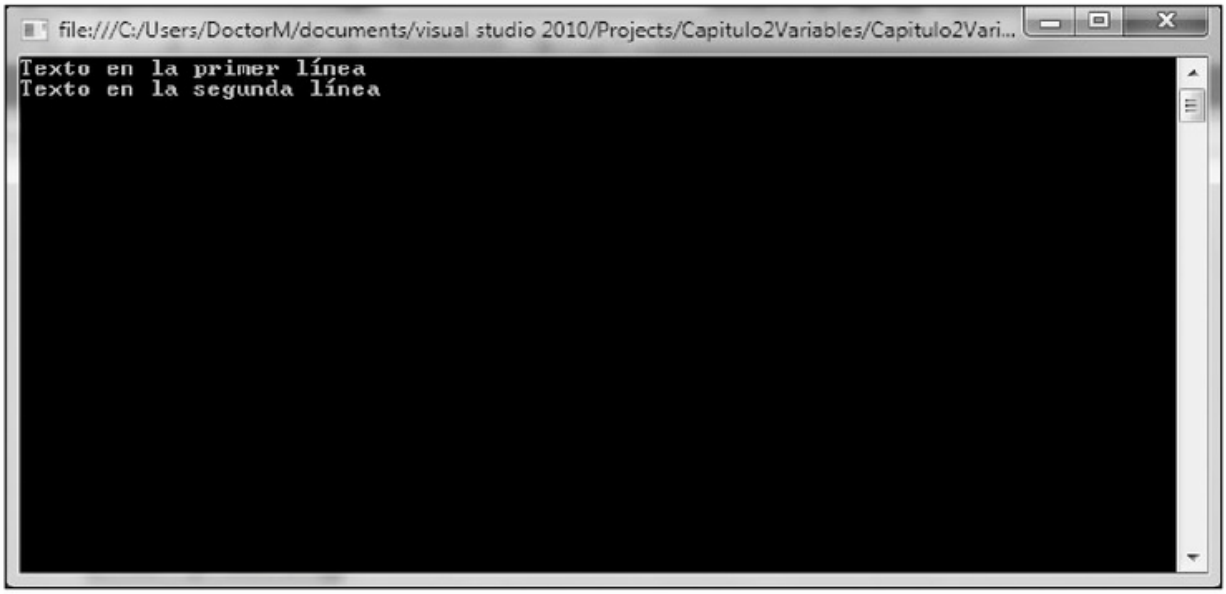


Figura 22. *Los caracteres especiales pueden ser usados en cadenas de texto y son tratados como un carácter más. Si escribimos uno para representar una nueva línea, al imprimirse en la consola este carácter también se representará.*

III VECTORES DE CARACTERES

Las cadenas de texto o variables de tipo **string** son tratadas como una sucesión de caracteres enlazados; esto quiere decir que podemos ver los **strings** como vectores de caracteres. Siguiendo este concepto, es posible que podamos obtener carácter por carácter de una cadena de texto mediante un índice de vector con la forma **Variable String (índice)**.

Las variables de tipo **string** poseen un gran conjunto de utilidades para manipular el texto contenido en ellas. Así podremos obtener secciones de este, buscar una serie de caracteres y averiguar si existe dentro de esta, o eliminar los espacios al principio o al final del texto. En la **Tabla 9**, podemos ver la lista de estas operaciones.

OPERACIÓN	DESCRIPCIÓN	USO
Contains	Busca en la cadena de texto si contiene el carácter o cadena dada.	variableString.Contais(Carácter por buscar)
IndexOf	Busca el carácter dado y retorna la posición de este.	variableString.IndexOf(Carácter)
LastIndexOf	Busca el carácter dado empezando por el final de la cadena de texto.	variableString.LastIndexOF(Carácter)
Insert	Inserta texto a partir de una posición dada.	variableString.Insert(Posición, Texto por insertar)
Length	Cuenta el largo de la cadena de texto.	variableString.Lenght
Remove	Elimina parte del texto sobre la base de una posición y la cantidad de caracteres por remover.	variableString.Remove(Posición, Total por remover)
Replace	Reemplaza una porción de texto sobre la base de las coincidencias encontradas.	variableString.Replace (Cadena por buscar, Cadena de reemplazo)
Substring	Obtiene una parte de la cadena de texto.	variableString.Substring (Posición de inicio, Cantidad de caracteres)
ToLower	Transforma el texto en minúsculas.	variableString.ToLower()
ToUpper	Transforma el texto en mayúsculas.	variableString.ToUpper()
Trim	Elimina los espacios anteriores y posteriores de la cadena de texto.	variableString.Trim()

***Tabla 9.** Herramientas para la manipulación de cadenas de texto dentro de variables string.*

El siguiente código implementa todas las herramientas vistas en la **Tabla 9**.

```
string mensaje = " Este es un mensaje de texto ";

Console.WriteLine(mensaje.Contains("a"));
Console.WriteLine(mensaje.IndexOf('E'));
Console.WriteLine(mensaje.Insert(2, "Insertado"));
Console.WriteLine(mensaje.LastIndexOf('o'));
Console.WriteLine(mensaje.Length);
Console.WriteLine(mensaje.Remove(0, 10));
Console.WriteLine(mensaje.Replace("e", "A"));
Console.WriteLine(mensaje.Substring(0, 10));
Console.WriteLine(mensaje.ToLower());
Console.WriteLine(mensaje.ToUpper());
Console.WriteLine(mensaje.Trim());
```

Podemos ver el resultado de cada una de las operaciones en la **Figura 23**.

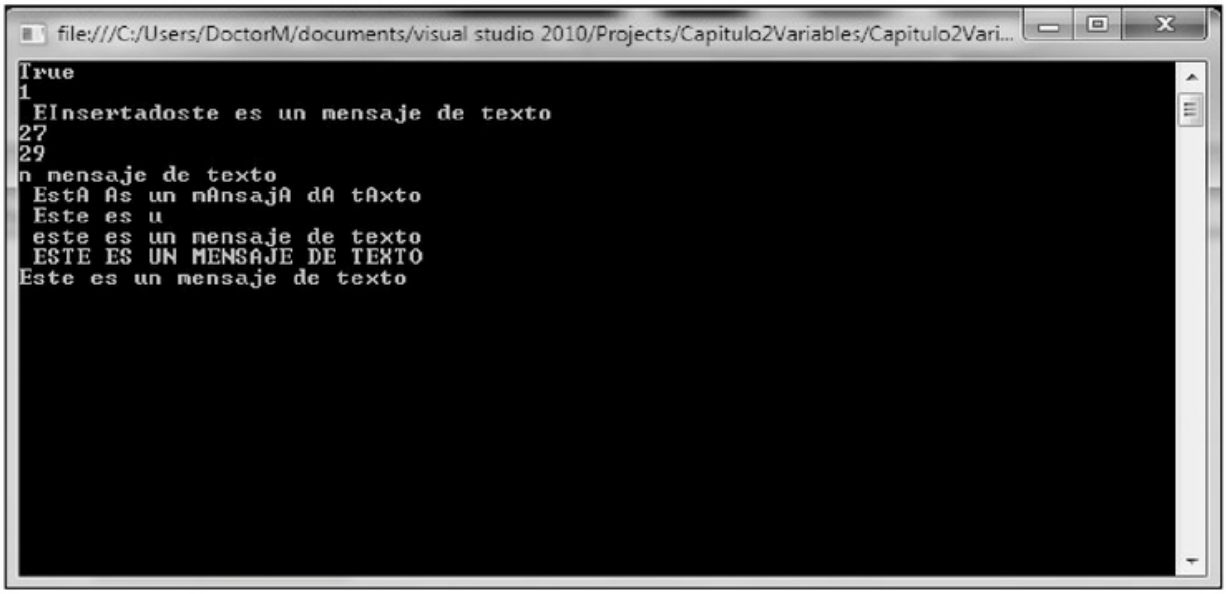


***Figura 23.** Los tipos string poseen funciones para la manipulación del texto contenido en ellos. Podemos reemplazar parte de un texto, extraer letras o buscar por coincidencias dentro de ellos sobre la base de otro texto.*

Podemos notar que las diferentes operaciones sobre las variables **string** son accesibles mediante el uso de un punto (.). Esta es una característica de acceso a miembros, funciones y propiedades de objetos.
No nos debemos preocupar si, en este momento, no comprendemos su funcionamiento; hablaremos de esto en el próximo capítulo, cuando veamos el siguiente paradigma de programación: la orientación a objetos. En todo caso, para finalizar con el uso de variables, construyamos una pequeña aplicación que pida información al usuario, la almacene en variables y, luego, muestre su contenido. Veamos el siguiente código.

```
string nombre;

Console.Write("Escriba su nombre: ");
nombre = Console.ReadLine();

Console.WriteLine("Bienvenido " + nombre);
```

El primer paso es la declaración de una variable que contendrá lo ingresado por el usuario en la consola. A continuación, escribimos en la consola un mensaje. Notemos el uso de **Console.ReadLine()**; esta línea detiene la ejecución del código de forma momentánea para que el usuario pueda escribir algo desde el teclado. En el momento en que el usuario escribe un texto y acepta la entrada presionando la tecla **ENTER**,

lo escrito por este será asignado a la variable **nombre**, así podremos contener el valor escrito dentro de nuestra variable. Finalmente, escribimos un mensaje y sumamos, en el texto de salida, el contenido de la variable. No habíamos usado aún el operador de suma con este tipo de variables; de cualquier manera, veremos que en la consola se escribirá el texto inicial seguido del contenido de nuestra variable (**Figura 24**).

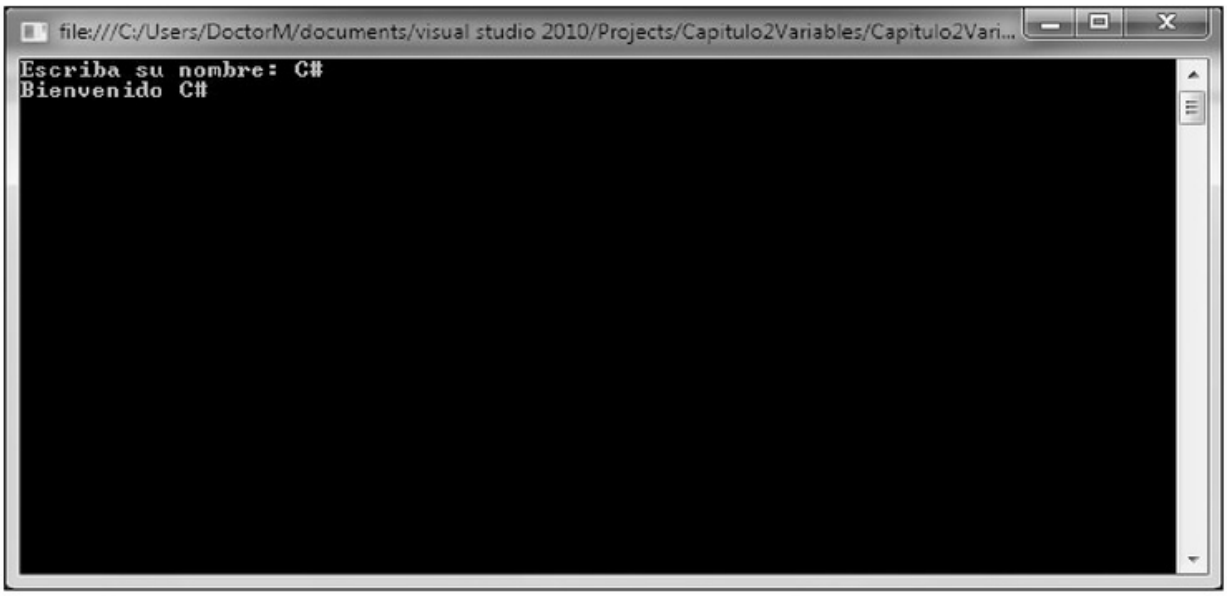


Figura 24. *Podemos pedir por la introducción de texto al usuario desde la consola, almacenarla en una variable para luego procesarla y mostrar los resultados en la misma consola.*

Estructuras de control

Hasta aquí, en este capítulo, hemos visto cómo crear las primeras líneas de código, crear espacios donde almacenar información llamados variables, y conocemos los principios de la programación estructurada.

Uno de esos principios es el de selección, el cual nos permitía generar caminos alternativos en la ejecución del código basados en alguna condición dada.

Por ejemplo, supongamos que estamos realizando una aplicación bancaria, en la que un usuario necesita retirar un monto determinado de su cuenta, pero solo podrá realizarlo siempre y cuando el balance de su cuenta contenga la cantidad solicitada. Si por el contrario, el usuario no contara con los fondos necesarios, debería recibir un mensaje avisándole sobre esto y, por lo tanto, no retirar ningún dinero. Esto es posible conseguirlo mediante la estructura de control **if** (si condicional). Veamos su sintaxis en las líneas de código que se encuentran a continuación.

```
if (Expresión booleana)
     Única línea de código si la expresión fue verdadera
```

La palabra reservada **if** es seguida de una expresión booleana, ejecutará la siguiente línea de código contenida dentro de la estructura de control. En el pseudocódigo anterior, solo una línea de código será ejecutada. Esta será la más próxima a la sentencia **if**. Si necesitamos ejecutar más de una línea de código después de la evaluación booleana será necesario que coloquemos **{ }** para encerrar todas las líneas que quieran ser ejecutadas.

```
if (Expresión booleana)
{
     ...
     Líneas de ejecución
     ...
}
```

Recordemos que una expresión booleana es cualquier expresión que arroje dos posibles valores, o **verdadero** o **falso**. Para esto, deberemos ampliar los operadores ya conocidos de análisis de expresiones booleanas (**Tabla 10**).

OPERADOR	DESCRIPCIÓN
==	Representa una igualdad. 10 == 10 es igual a verdadero.
!=	Representa una desigualdad. No es igual. 5 != 10 es igual a verdadero.
>	Mayor que. 10 > 5 es igual a verdadero.
<	Menor que. 5 < 10 es igual a verdadero.
>=	Mayor o igual que. 10 >= 5 es igual a verdadero. 10 >= 10 es igual a verdadero.
<=	Menor o igual que. 5 <= 10 es igual a verdadero. 10 <= 10 es igual a verdadero.

***Tabla 10.** Operadores utilizados en operaciones booleanas dentro de la estructura de control if.*

Con esto, implementemos el ejemplo propuesto al inicio de esta sección.

```
int balance = 1000;
int retiro = 500;
```

```
if (balance >= retiro)
{
    balance -= retiro;
    Console.WriteLine("Se ha producido el retiro de forma correcta");
}

Console.WriteLine("Su saldo es: " + balance);
```

Creamos dos variables enteras, una contendrá el balance del usuario, la otra, el monto por retirar. Antes de realizar el retiro de dinero, se verifica si el saldo es suficiente mediante el uso de la sentencia **if** utilizando uno de los operadores vistos en la **Tabla 10**. En el ejemplo, **mil es mayor o igual que quinientos**, por lo tanto, el resultado de la operación booleana es verdadero, por lo que el flujo de código seguirá por dentro de la condición produciendo el retiro, escribiendo un mensaje de aviso al usuario para finalmente, escribir el saldo final de la cuenta del usuario (**Figura 25**).

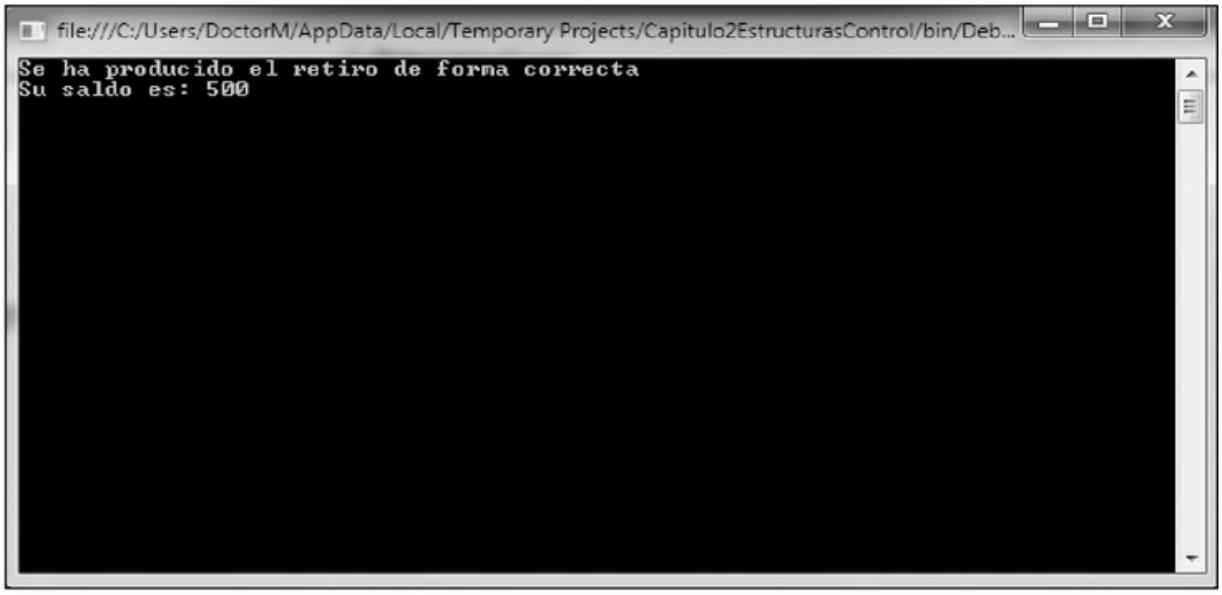


Figura 25. *Al ejecutar el código, la transacción que realiza el retiro de dinero se ejecuta exitosamente. La variable balance terminará por contener el valor de 500.*

La estructura de control **if** posee además un camino alternativo si la condición no se cumple o el resultado de su evaluación es igual a **falso**. Esto es útil si necesitamos ejecutar algunas líneas de código antes de que el flujo normal siga su curso.

```
if (Condición booleana)
{
```

```
        ...
}
else
{
        ...
        Líneas de código ejecutadas si la expresión evaluada es falsa
        ...
}
```

El uso de la palabra reservada **else** (si no) habilita el camino alternativo para una evaluación falsa. Agreguemos este camino a nuestra aplicación.

```
if (balance >= retiro)
{
    balance -= retiro;
    Console.WriteLine("Se ha producido el retiro de forma correcta");
}
else
{
    Console.WriteLine("Usted no tiene suficientes fondos");
}
```

Si en este caso intentáramos retirar más dinero de aquel que tuviéramos en nuestro balance, el segundo mensaje será mostrado en vez de realizar el retiro (**Figura 26**).

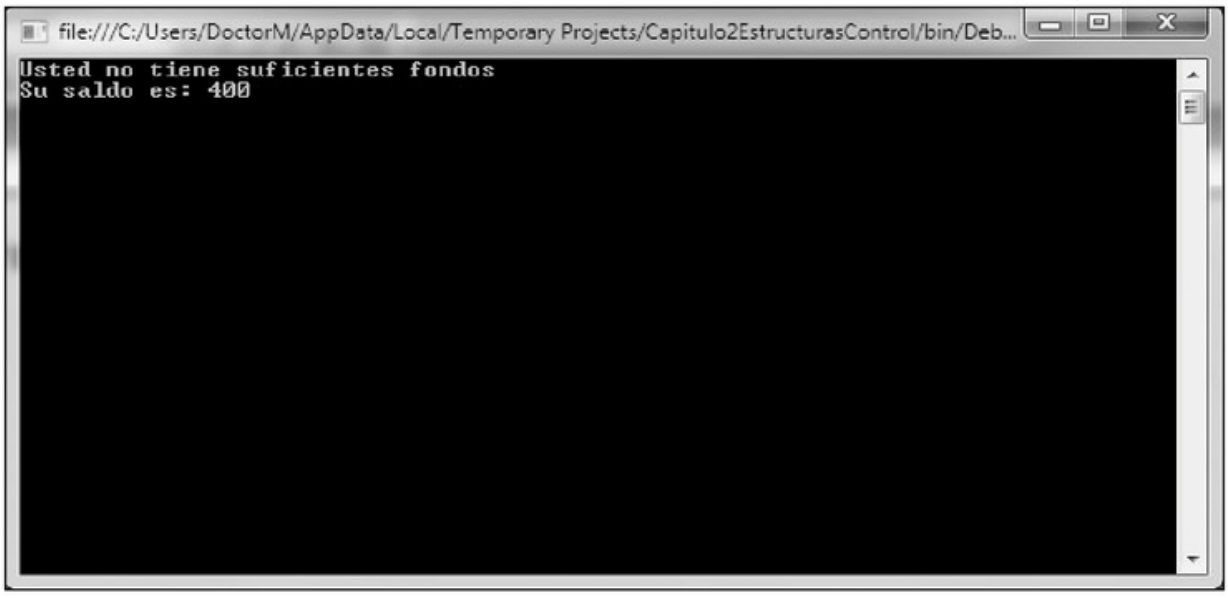


Figura 26. *Nuestro balance es menor al monto que se intenta retirar.*

Es posible encadenar una secuencia de condicionamientos. Esto quiere decir que podríamos colocar tantas instrucciones **if** como veamos necesarias.

```
if (Condición booleana)
{
     ...
     if (Condición booleana)
     {
            if (...)
}
else
{
     ...
}
}
else
{
     if (...)
}
```

Como podemos ver, el uso de las estructuras de control no están restringidas a un solo nivel, sino que es posible anidar tantas como creamos necesarias. De cualquier manera, sí tenemos una forma diferente de anidar estas condiciones en el caso del camino falso. Esto se logra mediante el uso de **else if**.

```
if (Condición booleana)
{
     ...
}
else if (Condición booleana)
```

III VARIABLES

Existe un tipo de variable llamada **object**. Debido a que esta es el tipo base de construcción de todos los objetos y tipos en Microsoft .Net, si utilizamos esta clase de dato para nuestras variables, estas podrán contener cualquier otro tipo, incluso podrán incluir instancias de clases.

```
{
     ...
}
else
{
     ...
}
```

De esta forma, en cada instrucción **else**, es posible agregar otro **if** con una condición, la cual contará de dos partes: la verdadera inmediatamente siguiente, y la falsa con un nuevo **else**, el mismo que puede ser completado con otro **else if** si lo viéramos necesario, y el ciclo se repetiría. Agreguemos algunas líneas más a nuestro código.

```
if (balance >= retiro)
{
     balance -= retiro;
     Console.WriteLine("Se ha producido el retiro de forma correcta");
}
else if (retiro == 500)
{
     Console.WriteLine("Su retiro de 500 no puede ser completado");
}
else
{
     Console.WriteLine("Usted no tiene suficientes fondos");
}
```

En este código, si los fondos son insuficientes, pero el monto por retirar es de quinientos, se mostrará un mensaje particular. Si fuese cualquier otro monto mayor al del balance, se mostrará el mensaje de fondos insuficientes (**Figura 27**).

III VARIABLES

Si usamos variables de tipo **object**, por más que estas permitan almacenar cualquier otro tipo dentro de ellas, podemos crear una sobrecarga de procesos al guardar y recuperar desde ella. En especial, si la usamos para almacenar tipos primitivos como **int**, **bool**, etcétera, ya que estos se almacenan de forma diferente en la memoria que los tipos **object**.

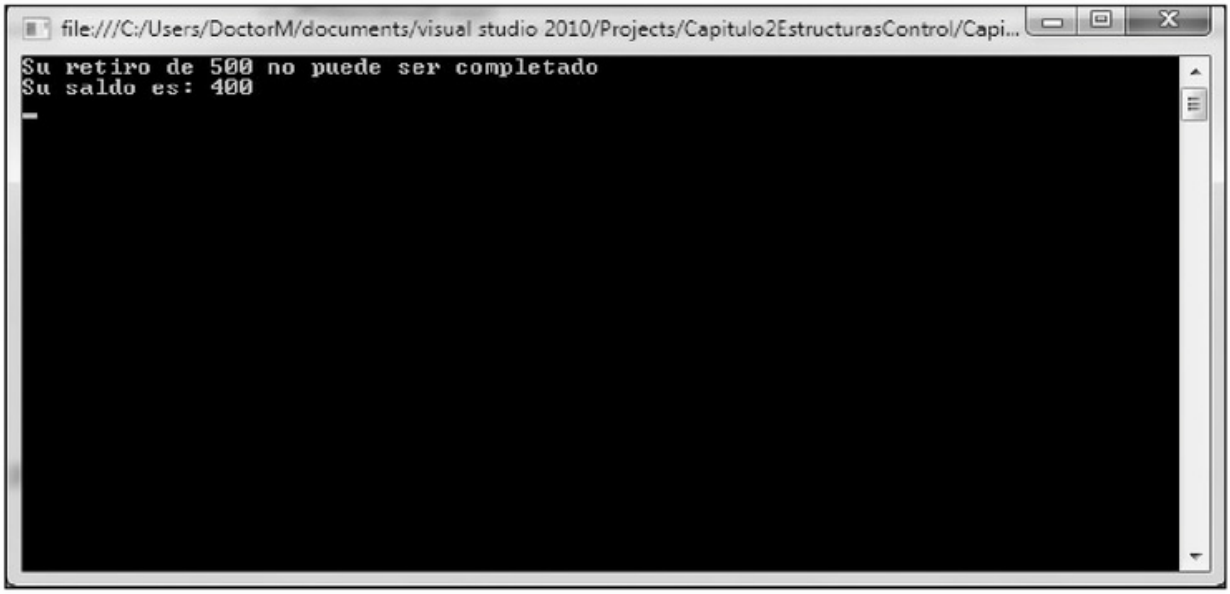


Figura 27. *Distintos mensajes son mostrados dependiendo de las condiciones utilizadas.*

Por último, la condición booleana esperada por la sentencia **if** puede ser compuesta. Recordemos los operadores **&&** (AND), || (OR) y **!** (NOT).
Estos operadores, nombrados en el párrafo anterior, pueden ser usados para crear expresiones más complejas y así evaluar más condiciones al mismo tiempo.

```
int balance = 400;
int retiro = 500;
string usuario;
Console.Write("Escriba nombre de usuario: ");
usuario = Console.ReadLine();
if (balance >= retiro)
{
    balance -= retiro;
    Console.WriteLine("Se ha producido el retiro de forma correcta");
}
else if (retiro == 500 && usuario == "administrador")
```

III CONDICIONES MÚLTIPLES

Cuando confeccionamos la condición booleana en una instrucción **if**, podemos encontrar un comportamiento extraño si tenemos dos expresiones para ser evaluadas, pero unidas por un || (OR). Si la primera expresión da un resultado verdadero, la segunda no será evaluada. Esto puede ayudarnos en la prevención de errores cuando evaluamos objetos nulos colocados después del ||.

```
{
    balance -= retiro;
    Console.WriteLine("Como administrador se ha retirado el monto
            solicitado");
}
else
{
    Console.WriteLine("Usted no tiene suficientes fondos");
}
Console.WriteLine("Su saldo es: " + balance);
```

En el ejemplo final, vemos que se le pide al usuario su nombre. Si este es igual a **administrador**, el monto por retirar será aprobado, incluso, si no tiene saldo y si el monto es igual a quinientos. Esto se logra con el uso de **&&** (AND), ayudándonos a unir dos expresiones booleanas para que sus resultados sean evaluados en conjunto. Podemos entender esto mediante la siguiente expresión.

```
Si el usuario es igual a "administrador" Y el retiro es igual a 500
     entonces aceptar el retiro.
```

El pseudocódigo anterior debería ayudarnos a entender la condición planteada. El resultado podemos comprobarlo viendo la **Figura 28**. En la consola nos aparece "**Escriba nombre de usuario: administrador/ Como administrador se ha retirado el monto solicitado/ sus saldo es: -100**".

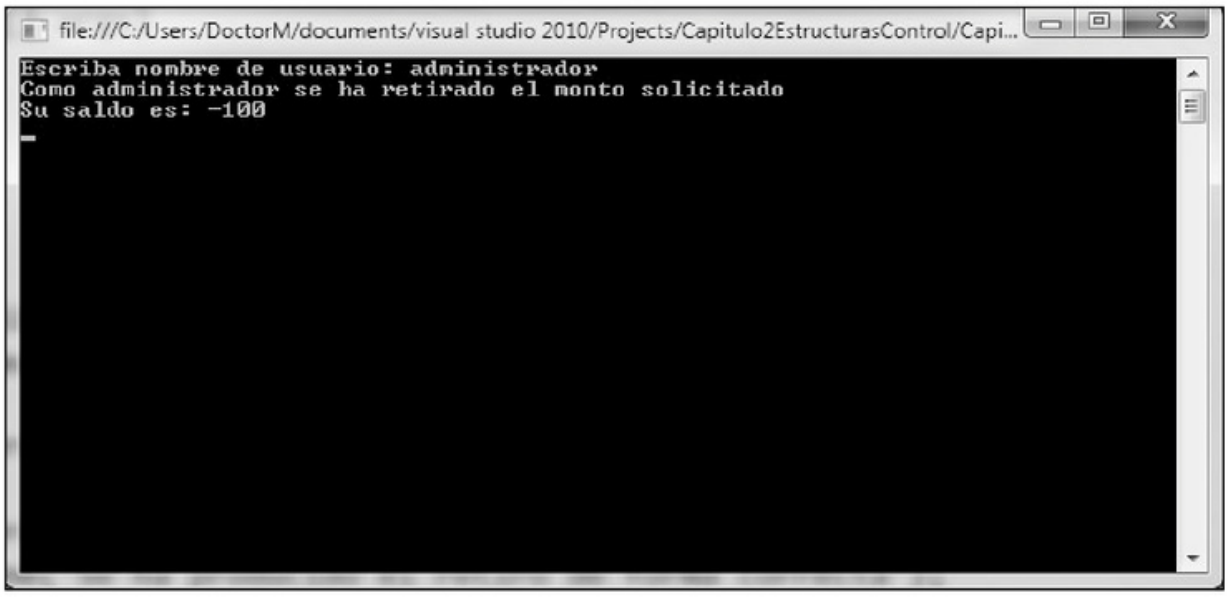


Figura 28. *Solo el administrador del sistema puede realizar retiros mayores a los de su balance.*

Estructura switch

La estructura **switch** es otra estructura de selección basada en una única evaluación booleana. Dado un valor para evaluar, se procede a ejecutar el código que se encuentre dentro del resultado único esperado. Para entender el **switch** (interruptor), imaginemos una serie de interruptores que encienden o apagan diferentes bombillas eléctricas a lo largo de una fila de bombillas. Nuestro código es equivalente a la electricidad que recorre los cables. Solo puede haber un interruptor activado al mismo tiempo y, dependiendo de cuál de los interruptores esté activado, se encenderá una u otra bombilla.

```
switch (Expresión)
{
     case Evaluador:
            break;
     case Evaluador 2:

            break;
     ...
     ...
     default:
            break;
}
```

Como podemos ver en el código anterior, cada opción por evaluar se destaca por el uso de la palabra reservada **case** (caso). Además, notaremos el uso de otra palabra reservada, **default** (por defecto). Tanto **case** como **default** pueden contener diferentes líneas de código que se ejecutarán si la expresión por evaluar coincide con el valor esperado, pero, en caso de que ninguna de las evaluaciones expresadas coincida con la expresión, entonces se ejecutará lo contenido en **default**.
Veamos el caso de las bombillas eléctricas en el código fuente que se encuentra en la página siguiente de este libro.

III CONDICIONES BOOLEANAS

Así como en las expresiones aritméticas, en las condiciones booleanas de las estructuras de control es posible el uso de paréntesis como agrupador evaluativo. Cada operación booleana comparativa deberá estar contenida por paréntesis, pudiendo se agrupadas mediante el uso de los operadores **&&**, **||** y **!**.

```
string bombilla;
Console.Write("Elija el nombre de la bombilla: ");
bombilla = Console.ReadLine();
switch (bombilla)
{
     case "bombilla1":
          Console.WriteLine("Bombilla 1 activada");
          break;
     case "bombilla2":
          Console.WriteLine("Bombilla 2 activada");
          break;
     case "bombilla3":
          Console.WriteLine("Bombilla 3 activada");

        break;
     default:
          Console.WriteLine("Bombilla general activada");
          break;
}
```

Como podemos ver, si el usuario escribe **bombilla1**, **bombilla2** o **bombilla3**, se mostrará un mensaje de acuerdo con cada una de las posibilidades; por el contrario, si fuese cualquier otra opción, se mostraría un mensaje genérico al no encontrar una coincidencia válida, recomendamos revisar la **Figura 29**.

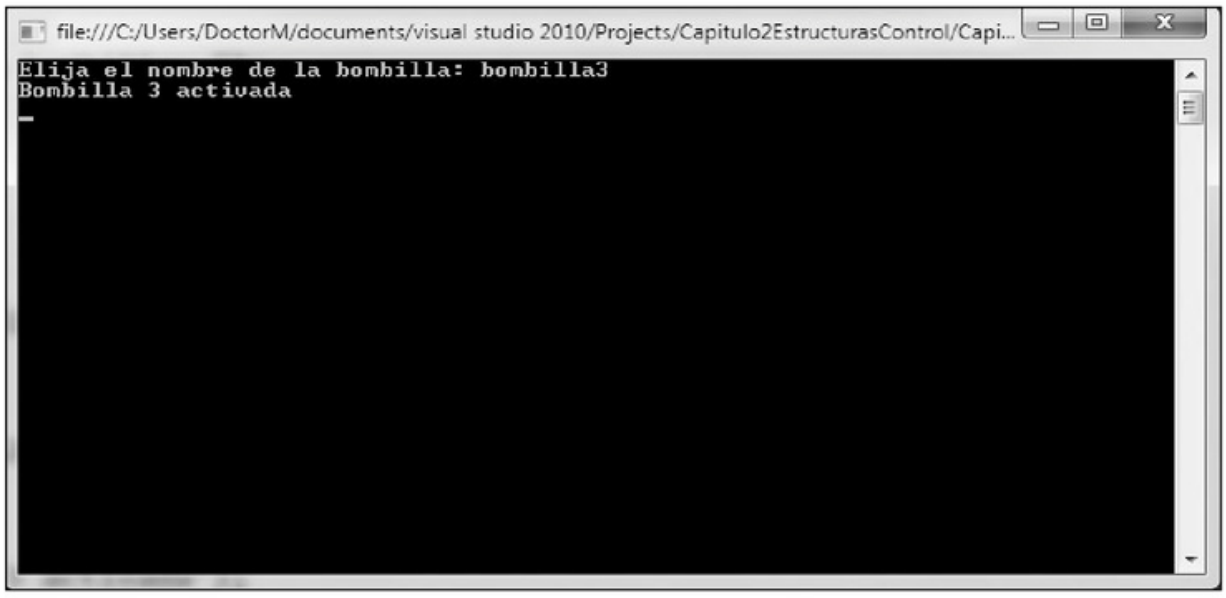


Figura 29. *El código atraviesa solo la sección que coincide con la evaluación del texto introducido por el usuario.*

Incluir la sección **default** no es obligatorio para la estructura del **switch**, pero es recomendable. Esto se debe a que, si por algún motivo nos hemos olvidado de colocar alguna coincidencia entre la expresión por evaluar y la lista de casos posibles, por lo menos, tendremos control al realizar algún código que nos alerte de ello sobre la base de lo escrito dentro de la sección **default** (**Figura 30**).
Por otra parte, podemos incluir tantas líneas como creamos necesarias después de cada **case**; no estamos limitados a solo una, por lo que podríamos plantear diferentes acciones por cada una de las opciones disponibles.

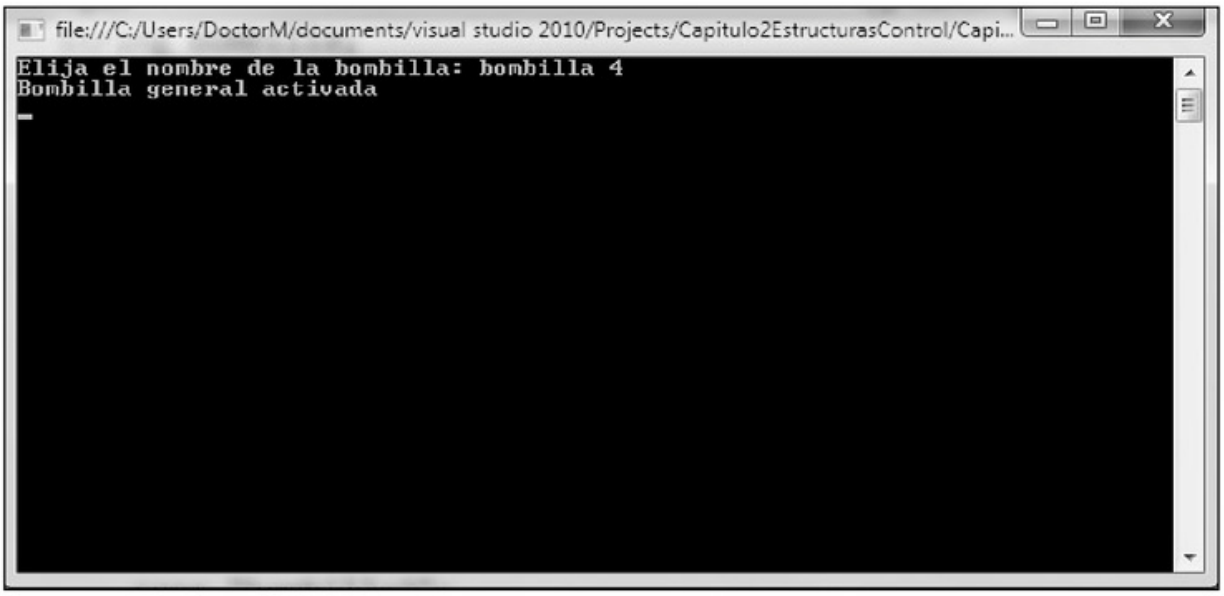


Figura 30. *Al no encontrarse coincidencia entre lo introducido por el usuario y las opciones, se muestra el código dentro de* **default**.

Otra palabra reservada que resalta en la estructura **switch** es **break**. Esta palabra reservada delimita hasta dónde llegará cada grupo de líneas bajo un mismo **case**. Si no incluyéramos esta palabra reservada, el código seguiría normalmente desde el punto en el cual se ingresó al **case**. Finalmente, es posible agrupar diferentes **case** bajo un mismo código; esto, por un lado evitará que realicemos código repetido, y, por otro, porque podríamos necesitar que bajo las diferentes condiciones, se requiera agrupar código ejecutable similar.

{ } EXPRESIONES BOOLEANAS Y SWITCH

La estructura **switch** requiere para su funcionamiento de una expresión por evaluar para luego elegir qué sección **case** ejecutar. Si lo que estamos evaluando es una expresión booleana o una variable booleana, además de evaluarla tal cual esta se presenta, podríamos utilizar el identificador de negación **!** (NOT) delante de esta expresión para hacer que el flujo del código se invierta.

```
case "bombilla4":
case "bombilla5":
    Console.WriteLine("Bombilla 4 y 5 activadas");
    break;
```

Si el usuario escribiera cualquiera de las dos opciones del código anterior, el mensaje sería el mismo (**Figura 31**), debido a cómo tenemos agrupadas las opciones del switch y la ausencia de la palabra reservada **break**.

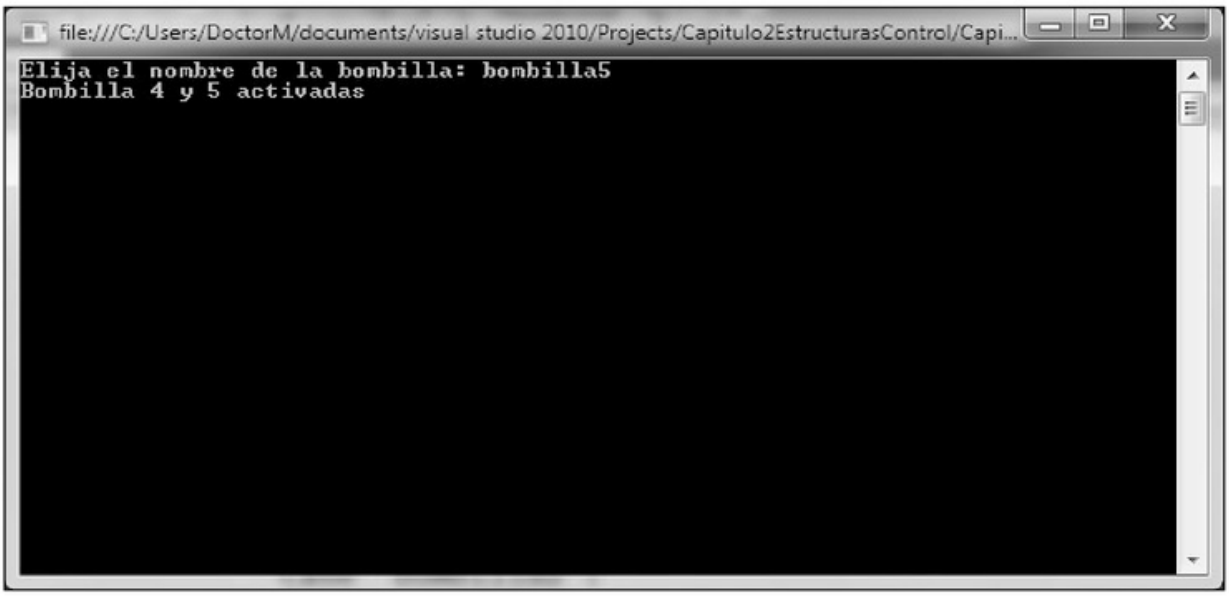


Figura 31. *Al agrupar más de un case, cualquier opción escrita por el usuario que coincidiera con alguno de ellos haría que se ejecutara el mismo código.*

Es posible también utilizar otro tipo de variables como parte de la expresión del **switch**. No solo las variables de tipo **string** están permitidas. Podremos utilizar variables numéricas como **int** o enumeraciones.
Dependiendo de qué tipo coloquemos en la expresión **switch**, los distintos case deberán adaptarse para manejar ese tipo.

{} CONSULTAS SQL

Las consultas a las bases de datos, también llamadas consultas T-SQL, fundamentan su funcionamiento en la aplicación de la teoría de conjuntos. Las uniones, las intersecciones y demás operaciones no nos deben resultar ajenas cuando nos sentemos a escribir en este tipo de lenguajes.

Estructuras de iteración

El último pilar al que hacíamos referencia es el de iteración. Este nos permite interactuar con código de forma repetitiva. En C# tenemos cuatro estructuras de iteración disponibles, **do ... while**, **for**, **while** y **foreach**. En esta sección, solo veremos las primeras tres y, luego, en los siguientes capítulos trabajaremos con la cuarta.

Estructura do ... while

Esta estructura de iteración es aquella que veíamos en el diagrama de la **Figura 8**, donde se realizarán, por lo menos una vez, las líneas de código contenidas en ella y, luego, se evaluará la condición para saber si se seguirá iterando sobre el mismo código.

```
do
{
     ...
     Líneas de código a iterar
     ...
} while (Condición booleana)
```

Como vemos, las condiciones booleanas son imprescindibles para estas estructuras de iteración, y así como en las de control, podremos utilizar todo el conjunto de operaciones booleanas que necesitemos dentro de la expresión esperada por la estructura en cuestión, vistas al inicio de este capítulo.

```
int i = 0;
do

{
    i++;
    Console.WriteLine("El valor de I es: " + i);
} while (i <= 10);
```

{} PARADIGMAS DE PROGRAMACIÓN

Existen tantos paradigmas como los lenguajes que los soporten, por lo tanto, es común que encontremos diferentes formas de desarrollar código según el lenguaje que estemos usando. Muchas veces, los principales paradigmas se conjugan para formar uno nuevo, en consecuencia, es recomendable saber aquellos básicos para luego poder aplicarlos correctamente en cada uno.

El código anterior suma 1 a la variable **i** hasta que esta sea **menor o igual a 10**. Como podemos notar en la **Figura 32**, la iteración no se detiene en el número 10, sino que la variable **i** llega a valer **11**. Esto se debe a que, de acuerdo con esta estructura, primero se suma el valor en **i** para luego ser analizado por la expresión booleana.

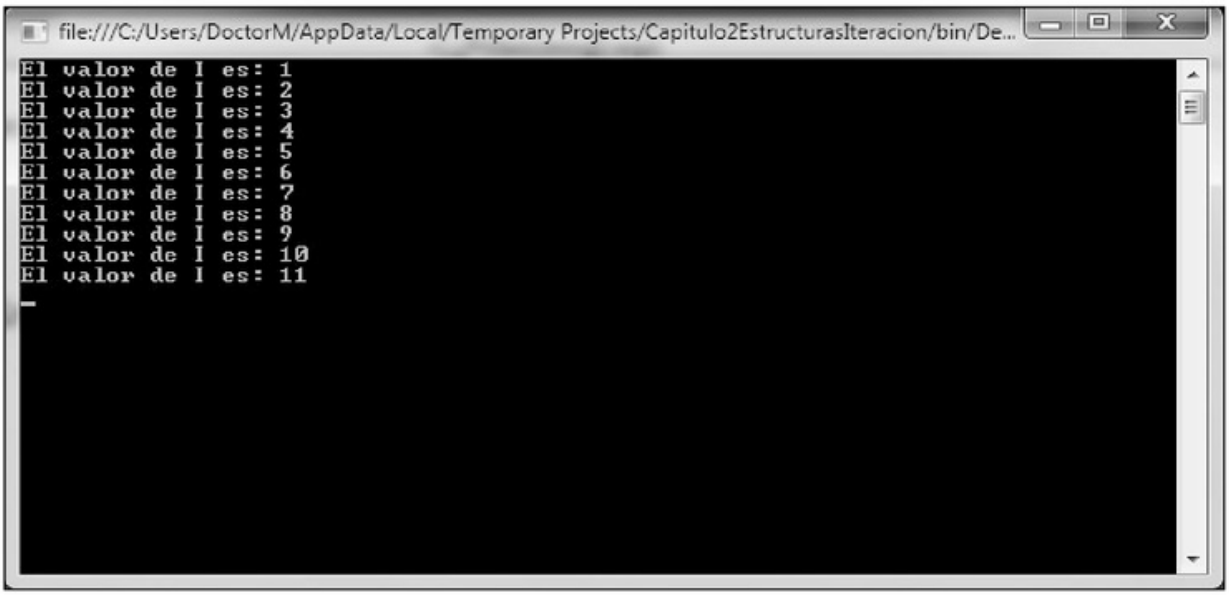


***Figura 32.** Al iterar sobre la variable i, esta acumula el valor por cada iteración mostrando el resultado en la consola.*

Esta estructura, entonces, ejecuta primero y luego analiza si la condición dada se ha alcanzado. Para este caso, si quisiéramos llegar solo hasta 10, deberíamos colocar un número más chico en la condición booleana por evaluar debido a que el código es ejecutado una vez, como mínimo. Podemos ver que funciona de forma inversa en la siguiente estructura de control, la estructura **while**.

Estructura while

La estructura **while**, a diferencia de la anterior, evalúa primero la condición booleana y, luego, ingresa al bucle de iteración sobre el código. Si la condición no es alcanzada en primera instancia, nunca se ejecutará el código dentro del **while**. Podemos ver esta estructura aplicada en la línea de código siguiente:

VARIABLE I

Encontraremos que, en casi todos los ejemplos que incluyan la estructura **for**, el uso de **i** como variable es predominante. Este convencionalismo se arrastra, según se cree, desde los primeros lenguajes de programación como **Fortran** y el uso de un identificador para el tipo de dato utilizado por la variable en cuestión, siendo esta variable un entero (**int**) y su inicial igual a **i**.

```
while (Condición booleana)
{
     ...
     Líneas de código a iterar
     ...
}
```

Supongamos que **i**, a la salida de la iteración anterior contiene el valor de **11**, por lo tanto, el bucle de iteración **while** evaluará la condición dada, no cumpliéndose y, por ende, no ingresando al bucle.

```
int i = 11;
while (i <= 10)
{
    i++;
    Console.WriteLine("El valor de I es: " + i);
}
```

En la **Figura 33**, podemos visualizar de qué manera nada es mostrado en la consola debido a que el bucle no se dispara en el proceso.

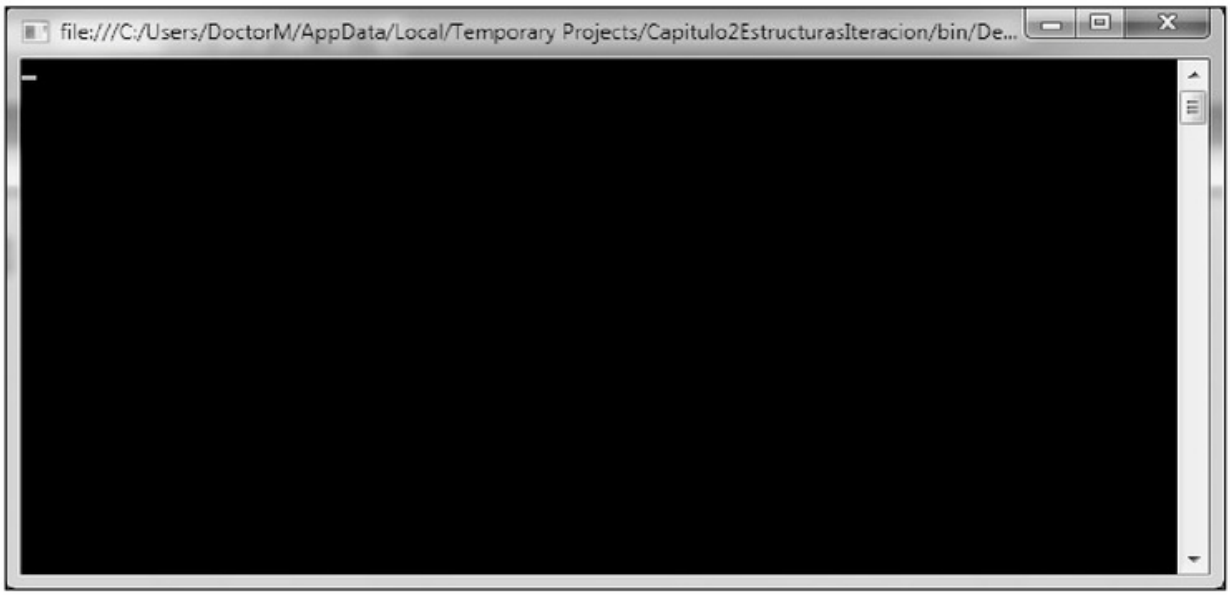


Figura 33. *El bucle while solo se ejecuta mientras que la condición se cumpla. A diferencia de do ... while, ninguna línea de código es ejecutada en primera instancia.*

Por supuesto, si la variable **i** tuviera cualquier otro valor menor o igual a 10, el bucle **while** se ejecutaría con normalidad.

Estructura for

Es otra estructura de iteración, pero presenta mayor versatilidad; a la hora de formularla y usarla permite definir un inicializador, una condición booleana por evaluar y un algoritmo de modificación de la condición y de la variable inicializada.

```
for (inicializador; condición booleana; expresión)
{
     ...
     Líneas de código a iterar
     ...
}
```

Cada una de las partes de la estructura **for** debe ser separada por **;** (punto y coma), y cada una de estas realiza una función específica.

```
for (int i = 0; i <= 10; i++)
{
    Console.WriteLine("Valor de I: " + i);
}
```

En el ejemplo anterior, la primera sección crea una variable **i** con un valor inicial de **0**. Luego, define la condición por la cual el bucle de iteración se mantendrá en funcionamiento. Finalmente, la tercera parte es una expresión que actuará sobre la variable creada en la primera parte sumando **1** a la variable **i**. En la **Figura 34,** vemos cómo obtenemos el mismo resultado que en las anteriores estructuras de iteración, pero mediante la declaración de las tres secciones del **for**. La variable creada como primer parámetro del bucle **for** es destruida en el momento en que la iteración termina, ahorrándonos memoria y sin la necesidad de que una variable declarada anteriormente perdure durante toda la ejecución del programa y es lo suficientemente versátil para que no tengamos que declarar ninguna o podamos prescindir de alguna de ellas.

III VARIABLES

En las nuevas versiones de C# y Microsoft .Net, se han adicionado novedosos identificadores de variables. Ya no solo contamos con los tipos básicos como **int**, **bool**, etcétera, sino que además podemos declarar nuestras variables mediante el uso de **var**. Este tipo de variables se utilizan para que, al compilar la aplicación, se infiera el tipo que esta contendrá y se transforme en una de esta clase.

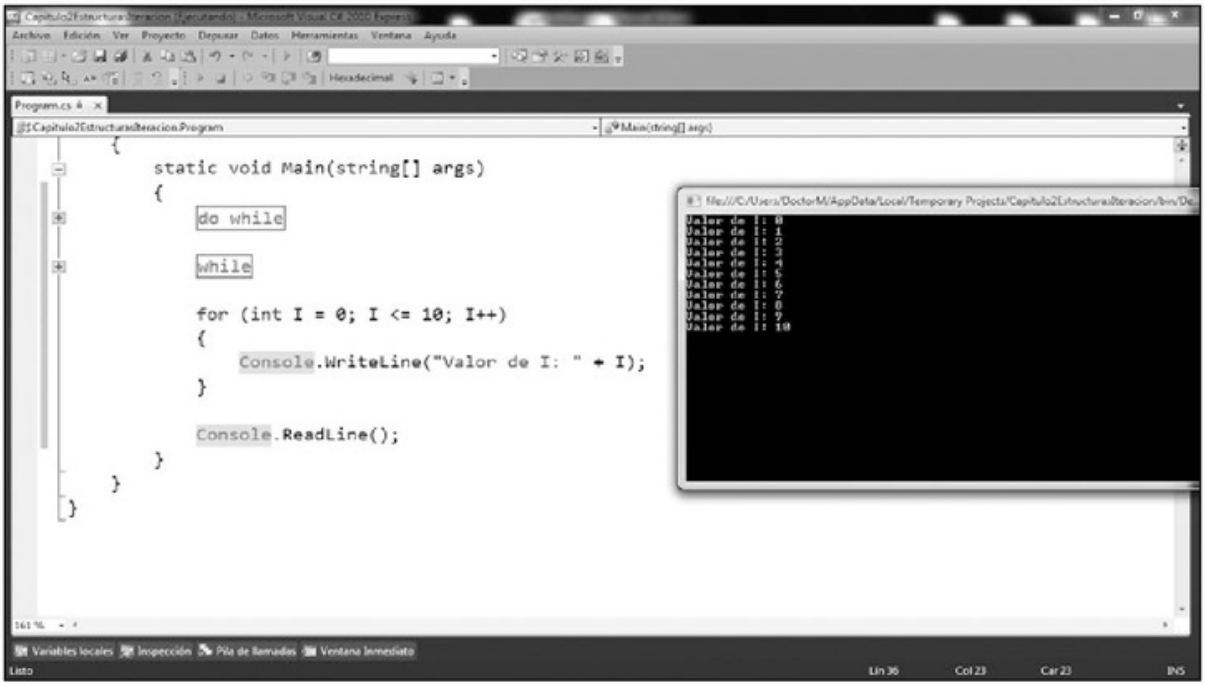


Figura 34. *El bucle for es ideal para tener el control de los estados de la variable iterada. Además, puede ser declarado en su conjunto en una sola línea de código.*

```
int contador = 0;

for (; contador < 10; )
{
    contador++;
    Console.WriteLine(contador);
}
```

Tengamos en cuenta que la condición booleana no necesita depender de una variable numérica, sino que simplemente requiere de una expresión que, al ser evaluada, retorne **verdadero** o **falso**, por lo que el uso de una variable numérica debe ser tomada simplemente como un ejemplo de las tantas posibilidades que tenemos a disposición. Por otra parte, la sección correspondiente a la función que anteriormente

III TEORÍA DE CONJUNTOS

La teoría de conjuntos es fundamental para el entendimiento de los operadores booleanos así como para lenguajes de programación de alto nivel. Es recomendable que volvamos a los libros de matemáticas para recordar estos conceptos, de cualquier manera, podemos leer algunos principios en la siguiente dirección: **http://es.wikipedia.org/wiki/Teor%C3%ADa_de_conjuntos**.

acumulaba **i**, también ha desaparecido, y la hemos trasladado dentro del bucle, por lo que, por cada iteración la variable contador sumará uno más, pudiendo salir del bucle en algún momento. El resultado es el mismo que en el caso anterior.

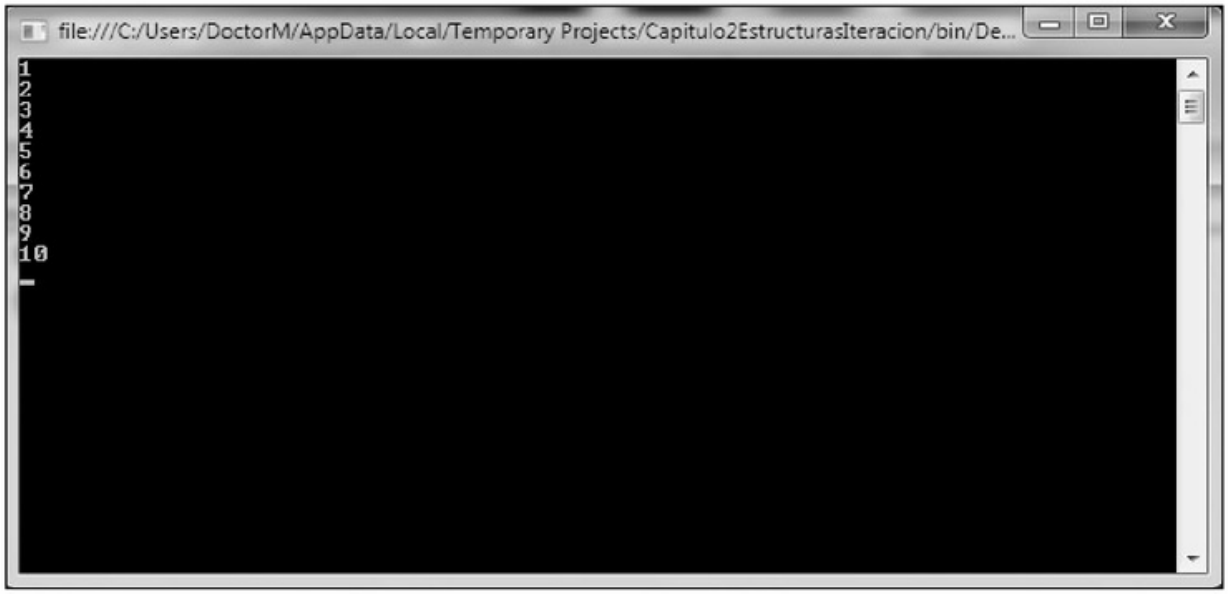


***Figura 35.** El bucle for sin variable inicializadora ni acumulador se comporta de igual manera que cuando los tenía.*

Finalmente, podemos quitar todos las partes del bucle **for** haciendo que este gire infinitamente. Si lo hacemos, debemos tener cuidado de que en algún momento este termine, de lo contrario podría causar un problema en nuestra aplicación al impedir que las demás líneas de código se ejecuten.

```
int contador = 0;

for (; ; )
{
    contador++;
    Console.WriteLine(contador);
    if (contador >= 10)
        break;
}
```

Este código presenta un bucle **for** sin parámetros en su ejecución, lo que hará que se ejecute de forma infinita. Para prevenir esto, debemos trasladar parte de la lógica que prevenga este problema dentro del mismo bucle.
Como vemos, el uso del **if** para verificar si el contador llega a determinado número hará, mediante el uso de la palabra reservada **break**, que paremos la ejecución del

bucle y las demás líneas de código sigan ejecutándose en forma normal (lo podemos ver ejemplificado en la **Figura 36** que se encuentra a continuación).

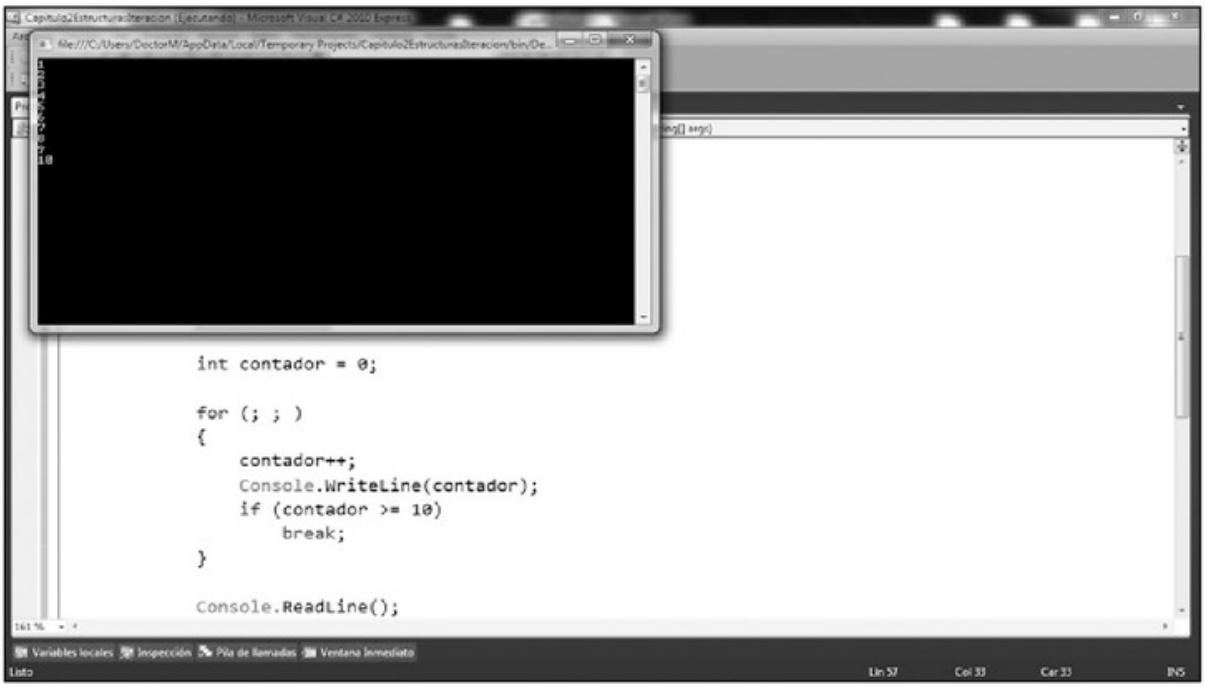


Figura 36. *El bucle sin parámetros se ejecutará infinitas veces, salvo que detengamos su ejecución mediante el uso de la palabra reservada* ***break****.*

La palabra reservada **break** provee la función de escapar de diferentes estructuras de control e iterativas, como en el caso de **switch**. Si nos encontramos usando bucles anidados, esto es, un **for** o más de uno dentro de otro, cada **break** terminará la ejecución del bucle más cercano, dejando que el superior siga su ejecución.

Vectores y matrices

Los **vectores**, también conocidos como **arreglos**, son un tipo de dato compuesto; una sucesión secuencial de datos de un tipo definido. Así, podríamos tener una sucesión de valores enteros **int**, **string**, o cualquier otro, pero trabajando en conjunto. ¿Nos acordamos de la disposición de la memoria? Una serie de casilleros contiguos donde cada uno podría contener un dato, aunque la diferencia, en este caso, es que cada casillero ocuparía tantos espacios en memoria como el tipo de variable que estemos usando, y así, la sucesión de este. Para entenderlo mejor pensemos en una fila de una caja de supermercado: cada persona que espera en esa fila para pagar representa un elemento del vector de clientes compuesto por personas. Por lo tanto, cada uno de ellos tendría un índice dentro de dicha fila. Si quisiéramos llamar a la tercera persona de esa fila, simplemente podríamos llamarla por su número de posición, y ella respondería. De la misma forma, un vector será este continuo de elementos, accesibles por un índice y de un tipo de dato definido (**Figura 37**).

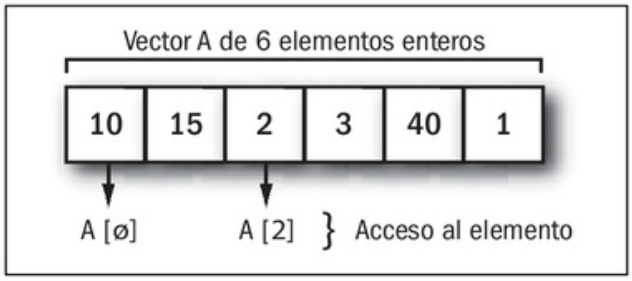


Figura 37. *Vector A de seis elementos enteros. El número entre los corchetes representa el índice de acceso al elemento deseado.*

La declaración de un vector difiere un poco de la declaración de variables comunes. Veamos la declaración de un vector de **6** elementos del tipo **string**.

```
string[] fila = new string[6];
```

Hay una significativa diferencia en la declaración. Por una parte, el tipo de dato es acompañado de **[ ]**. Esto significa que la variable contendrá un índice, podrá ser una sucesión de elementos del mismo tipo. Se crea un nuevo elemento de ese tipo especificando la cantidad de elementos que contendrá. La palabra reservada **new** es vital para esta acción. Asociemos dicha palabra reservada con el concepto de creación. Una vez creada la variable de tipo vector de **string**, es necesario poder asignarle información a cada uno de los elementos. El primer elemento contiene el índice **0**.

```
fila[0] = "Cliente 1";
fila[1] = "Cliente 2";
fila[2] = "Cliente 3";
fila[3] = "Cliente 4";
fila[4] = "Cliente 5";
fila[5] = "Cliente 6";
```

Como vemos, para cada uno de los elementos del vector, un texto es asignado. Recordemos que hemos creado un vector de **string**. Así como hemos usado el símbolo **[ ]** para acceder al índice del vector que necesitamos asignar, podemos usar el mismo mecanismo, pero del lado contrario de la igualdad para tomar sus valores.

```
Console.WriteLine(fila[3]);
```

Para este caso, accedemos al elemento **4** (índice 3) de nuestro vector y lo mostramos en la consola de la aplicación.

Por último, gracias a las propiedades de los vectores, es posible conocer el total de elementos que este contiene mediante el uso de **Length**, de igual forma que en el caso de un tipo **string**. Junto con una de las estructuras de iteración vistas, en este caso **for**, podríamos recorrer el total del vector.

```
for (int i = 0; i < fila.Length; i++)
{
    Console.WriteLine(fila[i]);
}
```

Matrices

Las **matrices** también son estructuras de datos compuestas, pero en este caso podemos considerarlas como vectores de vectores. Para imaginar esto, pensemos en un tablero de ajedrez en el que tenemos filas y columnas, y, en la conjunción de estas dos, un espacio donde poner las piezas, o si se trata de la programación, datos. Por supuesto, en este caso nos referimos a matrices de dos dimensiones, pero es posible tener de tres, cuatro o más dimensiones.

```
string[,] tablero = new string[8, 8];
```

La diferencia en la declaración de una matriz con un vector es la coma en la especificación del índice inicial. Vemos que, en el código anterior, existe una coma como si esta separara las filas y las columnas. Luego, al definir el tamaño de cada uno, también deberemos especificar con una coma cuántos elementos tendrán las filas y cuántas, las columnas. Si necesitáramos más dimensiones en nuestra matriz, solo deberíamos agregar más comas y tamaños para cada una de las dimensiones. Para recorrer todas las filas y columnas, podemos usar estructuras de iteración anidadas y así asignarle valores a la matriz. Esto recorrerá tanto las filas como las columnas, colocando un valor a cada intersección de la matriz.

```
for (int i = 0; i < 8; i++)
{
    for (int j = 0; j < 8; j++)
    {
        tablero[i, j] = "Pieza " + i;
    }
}
```

Para poder localizar un valor y recuperarlo, será necesario hacer referencia a la intersección entre fila y columna. Nuevamente, el índice inicial es igual a **0** y no a **1**.

```
Console.WriteLine(tablero[2, 1]);
```

Los vectores y las matrices son útiles para almacenar datos en forma grupal, pero, con la evolución del lenguaje, estos han sido dejados de lado dentro de la programación de aplicación para dar paso a las colecciones. Nos encargaremos de las colecciones en los siguientes capítulos.

Métodos y funciones

Al principio de este capítulo, hacíamos referencia a los paradigmas de programación; uno de ellos era el de la programación modular. Los métodos y funciones son parte de los pilares de la programación modular. Si bien la división entre métodos y funciones remite a lenguajes como Basic o Visual Basic, en C# solo las llamaremos funciones, ya que para este lenguaje no existe diferencia entre los dos casos. Entonces, las **funciones** representan la separación de funcionalidad de código en subrutinas de procesamiento. Pensemos que tenemos unas líneas de código que realizan la suma de dos números y necesitamos en nuestro programa utilizarlas repetidas veces. Sería lógico que, en lugar de tener que escribir cada vez que necesitemos realizar la suma, consignemos la fórmula. Si bien este caso puede resultar bastante simple, pensemos en una funcionalidad más compleja, que, en algún momento pudiera requerir alguna modificación; si este código se encontrara desperdigado por toda la aplicación, deberíamos buscar cada una de estas líneas y modificarlas, lo que implicaría, por supuesto, un costo innecesario.

```
Función principal del programa
     ...
     Líneas de código del programa principal
     ...
```

MATRICES MULTIDIMENSIONALES

Es posible crear matrices multidimensionales en C#, donde generalmente intentan simular un aspecto de la vida real. De dos dimensiones, simulando filas y columnas, de tres, un punto en el espacio, de cuatro, un punto en el espacio y tiempo. Si bien podemos seguir agregando dimensiones, resulta difícil para nuestro cerebro poder comprenderlas y manejarlas.

```
     Llamada a función secundaria
     ...
Fin de la función principal del programa

Función secundaria
     ...
     Líneas de código de la función secundaria
     ...
Fin de la función secundaria
```

Como vemos en el pseudocódigo anterior, una función es una pieza de funcionalidad secundaria del programa principal. Hasta ahora, todo el código que habíamos estado realizando estaba contenido en la función principal de nuestra aplicación de consola, por lo tanto, estas líneas también estaban dentro de una función.
Por otra parte, en el mismo pseudocódigo, podemos ver que, desde cualquier función, es posible llamar a otras funciones, en este caso, una función secundaria. Estas funciones secundarias, a las que desde ahora llamaremos simplemente funciones, sea cual fuere su naturaleza, al igual que cualquier otra función deben seguir una nomenclatura; dicha nomenclatura consta de un nombre de función, parámetros de entrada y valor de retorno.

```
int funcionSumar(int parametro1, int parametro2)
{
     ...
     Código de la función
     ...
}
```

En el código anterior, vemos cómo podemos definir una función tipo. Por un lado, tenemos el valor de retorno. Veamos que este valor de retorno puede ser cualquiera de los tipos conocidos hasta ahora, desde un entero a un **string**, así como una matriz o

III INMUTABILIDAD

Las matrices y los vectores son considerados, en sus dimensiones, como **inmutables**. Esto quiere decir que, una vez declarados, su tamaño no podrá modificarse, y ser más grande o más chico. Para conseguir un nuevo vector de diferente tamaño, tendremos que crear otro y copiar cada uno de los elementos del anterior al nuevo.

un vector. Si no pretendemos retornar valor alguno, deberemos usar la palabra reservada **void** en vez de un tipo conocido, como podemos ver en el siguiente código.

```
void funcionSinRetorno()
{
     ...
     Código de la función
     ...
}
```

Para este caso, la función **funcionSinRetorno**, además, no acepta parámetros de entrada, algo que la anterior sí hacía. Los parámetros de entrada también pueden ser de cualquier tipo conocido y, una vez que se ejecute la función, estos pasarán a ser tratados como unas variables más dentro de ella. Veamos entonces cómo podemos crear nuestra propia función para sumar dos números.

```
int Sumar(int parametro1, int parametro2)
{
     int resultado = 0;
     resultado = parametro1 + parametro2;
     return resultado;
}
```

La función **Sumar** retorna un valor entero, el mismo que servirá para retornar el resultado de dicha suma; además, acepta dos parámetros, cada uno para cada número por sumar. En su interior, el código simplemente realizará la suma de los dos parámetros, asignando estos últimos en la variable **resultado**; finalmente, este valor es devuelto por la función mediante el uso de la palabra reservada **return**. Esta palabra reservada es de vital importancia debido a que, sin esta, la función nunca devolvería un valor, por lo tanto, no la debemos olvidar.
El valor retornado debe ser del mismo tipo del que la función ha dicho que retornará. Por lo tanto, notaremos que la variable **resultado** es del mismo tipo (**int**) que el valor de retorno de la función. A continuación, implementaremos la llamada a esta función, en la siguiente línea de código.

```
int resultadoSuma = 0;
resultadoSuma = Sumar(10, 20);
Console.WriteLine("Resultado de primera suma: " + resultadoSuma);
```

```
resultadoSuma = Sumar(55, 100);
Console.WriteLine("Resultado de segunda suma: " + resultadoSuma);
```

Notemos que la función **Sumar** utiliza los paréntesis para decir cuáles serán los valores para cada uno de los parámetros que esta necesita. Cada parámetro es separado por el carácter de coma. Una vez llamada la función, siendo que esta retornará un valor entero, asignaremos dicho resultado a otra variable que contendrá el resultado de la operación. Veremos más abajo, en el código, que la función **Sumar** es llamada nuevamente, pero con valores distintos, y, como es de esperar, la función se ejecutará otra vez, aunque con estos valores se obtendrá un resultado diferente al anterior. Esto no es todo en relación con funciones y su uso; aquí solo hemos visto lo esencial para poder crear nuestras primeras aplicaciones. En los siguientes capítulos, volveremos a hablar del tema funciones para aumentar nuestro conocimientos sobre ellas.

... RESUMEN

En este capítulo, hemos visto en detalle los pilares fundamentales del desarrollo de software; conceptos como variables, bucles, lógica booleana y funciones son ahora identificables por nosotros. Para cada uno de los temas vistos, existe una profundidad mucho mayor e iremos ahondando en ellos en el transcurso de los distintos capítulos. Por lo tanto, es recomendable que practiquemos modificando los diferentes ejemplos ofrecidos hasta ahora para sentirnos más cómodos con cada uno de los elementos vistos; si es necesario, repasemos los términos en los que encontremos alguna duda ya que, como hemos dicho, esta es la base fundamental de la escritura y desarrollo de código.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Las funciones solo pueden retornar tipos string?

2 ¿Cuál es el rango de valores que puede contener un tipo int?

3 ¿Con qué carácter se accede a un índice en un vector?

4 ¿Cuál cree que es la forma más fácil de modificar el estado de una variable booleana usando la misma variable como dato de partida?

5 Indique la diferencia entre un if y un switch?

6 Determine si las funciones pueden no retornar valores.

7 ¿Cuál es la palabra reservada que se utiliza para retornar valores desde una función?

8 ¿Cuál es el operador equivalente en C# para representar un OR booleano?

9 ¿Cuántos parámetros puede recibir una función?

10 Indique si variables creadas dentro de funciones, estructuras de iteración o de control siguen activas luego de que estas funciones y estructuras hayan finalizado.

EJERCICIOS PRÁCTICOS

1 Tomando como base el ejemplo realizado con matrices, intente asignar un valor diferente a cada uno de los elementos de esta matriz.

2 Intente crear una función para restar dos valores.

3 En el ejemplo de la extracción bancaria, pruebe de agregar una nueva condición para que, si un usuario común intenta extraer mayor dinero que lo contenido en su balance, siempre y cuando no sea superior a 200, pueda hacerlo.

4 Teniendo una variable de tipo long, asígnele el valor máximo que esta pueda contener e intente asignar su contenido a otra variable de tipo int. Si obtiene un comportamiento extraño, ¿podría explicar el porqué?

5 Con el código de ejemplo realizado para simular transacciones bancarias, intente invertir las condiciones booleanas dentro de cada if para que la primera condición por evaluar sea la última de la lista. De esta forma, el mensaje de fondos insuficientes deberá ser el primer mensaje, luego la verificación de usuario, y finalmente el mensaje de extracción correcta.

Programación orientada a objetos

En el anterior capítulo, hemos aprendido dos de los principales paradigmas utilizados en el desarrollo de software, estos paradigmas nos servirán de base para el tercero, la programación orientada a objetos.

En este capítulo, nos enfocaremos en este nuevo concepto expandiendo nuestros conocimientos para así poder escribir código de mayor calidad.

PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

La **POO** (Programación orientada a objetos) es un paradigma que intenta plantear su fundamento en la representación de comportamiento en forma de objetos desde las líneas de código que simulen objetos de la vida real; de esta forma, un objeto creado en nuestro código tendrá características, propiedades y comportamiento semejantes al objeto que esté intentando representar del mundo real.

Para entender esto, imaginemos que necesitamos simular las características de un automóvil en nuestro código; si separamos en partes dicho vehículo, notaremos que contamos con un motor, puertas, ruedas, asientos, entre otros. Pero, como sabemos, cada uno de estos elementos posee un comportamiento único; el motor no funciona como una puerta ni una rueda, así como las ruedas delanteras no poseen el mismo comportamiento que las traseras; al mismo tiempo, la suma de todas estas partes componen el automóvil y su comportamiento como tal.

Por otro lado, nos damos cuenta de que existen partes del automóvil que son idénticas en su composición, aunque no lo sean en su comportamiento, por ejemplo, las cuatro ruedas son idénticas, pero dependiendo de dónde se encuentren podrán comportarse de forma diferente; por lo tanto, si extrapolamos esta idea al código, podríamos tener una única descripción de lo que es una rueda, pero debemos crear cuatro copias, o **instancias**, de esta con cuatro comportamientos, atributos y propiedades diferentes basándonos en la posición que ocuparán o el trabajo que realizarán. Dentro de este comportamiento, además estará incluida la interacción que pudiera tener con las demás partes del vehículo.

En esta conjunción de objetos, también vemos cómo algunos de estos tendrán contacto con el usuario, mientras otros se remitirán a proveer de **servicios de conexión**, **transporte de datos**, o **funcionalidad específica entre objetos**. Por lo tanto, al crear nuestros objetos en el código, así como en la construcción de objetos de la vida real, deberemos ser cuidadosos de no duplicar funcionalidad, saber colocar en el objeto correcto dicha funcionalidad y saber interconectar todas las partes con cuidado para no consumir recursos en forma innecesaria (**Figura 1**).

LA ORIENTACIÓN A OBJETOS

La programación orientada a objetos se hizo popular en los años 90. De cualquier manera, el concepto es mucho más antiguo. El primer lenguaje orientado a objetos fue **Simula 67**. Creado por dos ingenieros noruegos, Simula 67 ve la luz en 1967. A pesar de ello, debieron pasar casi treinta años para que este paradigma fuera ampliamente aceptado.

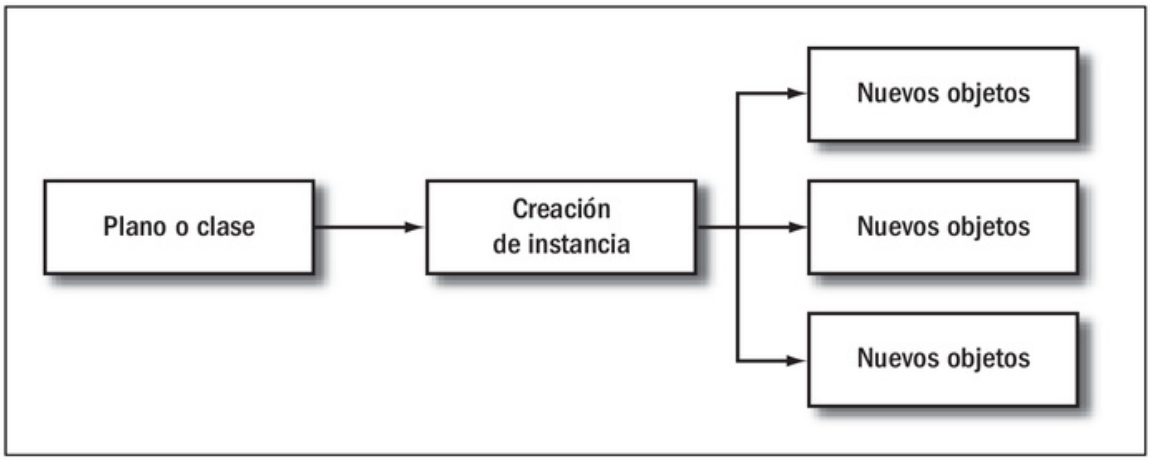


Figura 1. *Para crear un objeto determinado, necesitamos un plano, una forma de construirlo. Con este, es fácil replicarlo, transformándolo en nuevos objetos tangibles, con una misma base, pero únicos entre sí.*

El plano de construcción

Para construir nuestros objetos y tantas instancias o copias como necesitemos, es fundamental contar con un **plano** o **molde guía**. Este plano contendrá toda la información necesaria para replicar los distintos objetos, desde sus propiedades hasta su comportamiento. Cuando nos refiramos a planos o moldes guía, en realidad, estaremos hablando de **clases** y, si bien ya hemos visto estas clases en el código realizado hasta el momento, aún no las hemos tratado en detalle.

```
class Rueda
{
     ...
     int diametro;
     int identificador;
     string marca;
     ...

}
```

Una clase en C# comienza con la palabra reservada **class** (clase) seguida de un nombre que la identifique; en el código anterior, nuestra clase recibe el nombre de **Rueda**. También podemos ver en las líneas de código anteriores que la clase **Rueda** contiene una serie de variables declaradas dentro de ella, que serán utilizadas para definir las características de cada una de las instancias que podamos crear de esta clase. Esto quiere decir que cada una de las copias que realicemos de la clase **Rueda** podrá alojar información propia e independiente en estas variables.

Una vez definida nuestra clase, podemos crear un objeto tangible del tipo **Rueda** de esta clase de la siguiente forma (**Figura 2**). Por lo tanto, cada uno de los objetos que creemos basándonos en esta clase, contendría cada una de estas variables.

```
//Creamos una instancia
//de la clase Rueda

Rueda rueda1 = new Rueda();
```

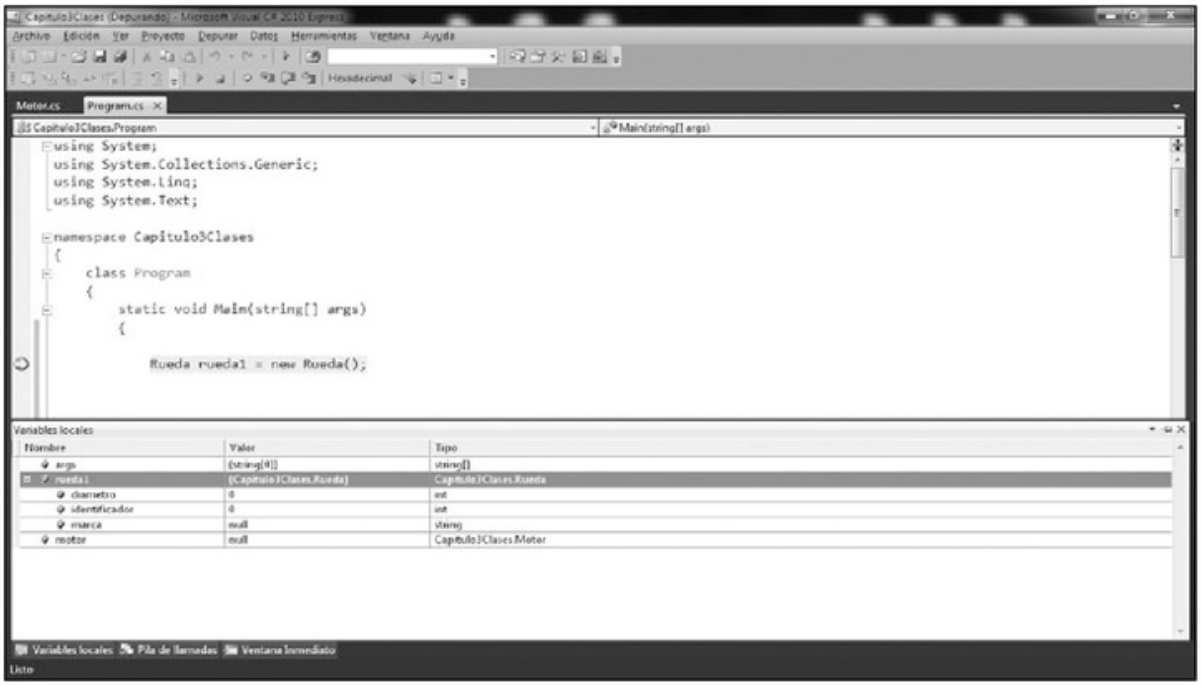


***Figura 2.** La palabra reservada new junto al nombre de la clase construye un objeto de esa clase. En el inspector de variables, vemos el contenido de la variable rueda1 con el tipo creado.*

En el código anterior, podemos ver que la declaración e instanciación de un objeto del tipo **Rueda** se asemeja al de la creación de una variable cualquiera. Por un lado, especificamos el tipo de la variable, el cual hará referencia al nombre de la clase pre-

III NUEVAS INSTANCIAS

Para crear una variable que contendrá una instancia de un nuevo objeto, no solo tenemos la posibilidad de usar la palabra reservada **new**. Si contamos con otra variable que ya contiene una instancia del mismo tipo de objeto, podemos asignar esta a la nueva variable y así tener una copia instanciada de ella.

viamente creada y el nombre de la variable que usaremos para contener este objeto, en este caso en la variable llamada **rueda1**.
El siguiente paso es asignar la creación de la instancia en sí mediante el uso de la palabra reservada **new** seguida del identificador o nombre de clase de la cual crearemos el objeto. Esto podría parecer redundante ya que necesitamos decir que el tipo de variable será del mismo tipo que de aquel del que crearemos el nuevo objeto. A pesar de esto, lo cierto es que podríamos declarar nuestra variable de un tipo y recibir una instancia de un objeto de un tipo diferente, pero que compartieran, en algún punto, ciertas similitudes. Por otro lado, sería posible tener una variable del tipo **Rueda**, pero que no contuviera instancia de objeto alguna.

```
//Creamos una instancia
//de la clase Rueda

Rueda rueda2;
...
//Otras líneas de código
...
rueda2 = new Rueda();
```

En este caso, la variable **rueda2** es del tipo **Rueda**, pero no contiene una instancia de objeto, por lo tanto, no podremos utilizarla hasta que no le asignemos un nuevo objeto **Rueda**. El problema principal al cual nos enfrentaremos si hacemos esto es que, a diferencia de las variables que hemos visto hasta el momento, donde si no se le asignaban valores iniciales al momento de su declaración, el motor de ejecución asignaba por nosotros uno por defecto, estas variables no contendrán un valor por defecto que pudiera relacionarse con el objeto que contendrá, por el contrario, estas contendrán el valor **null** (nulo) como podemos ver en la **Figura 3**.
Si una variable de las características de las de la **Figura 3** alojara un valor de tipo **null**, ninguna de las operaciones relacionadas con ese objeto serían posibles ya que el objeto mismo nunca habría sido instanciado.

III DECLARACIÓN Y ASIGNACIÓN

Como tantos otros lenguajes de programación, C# posee la capacidad de realizar muchas instrucciones de código en una única línea. Como en el caso de la declaración y asignación de variables, donde en la misma línea de declaración es posible asignar un valor inicial y no se necesita recurrir a líneas de código adicionales para su posterior asignación.

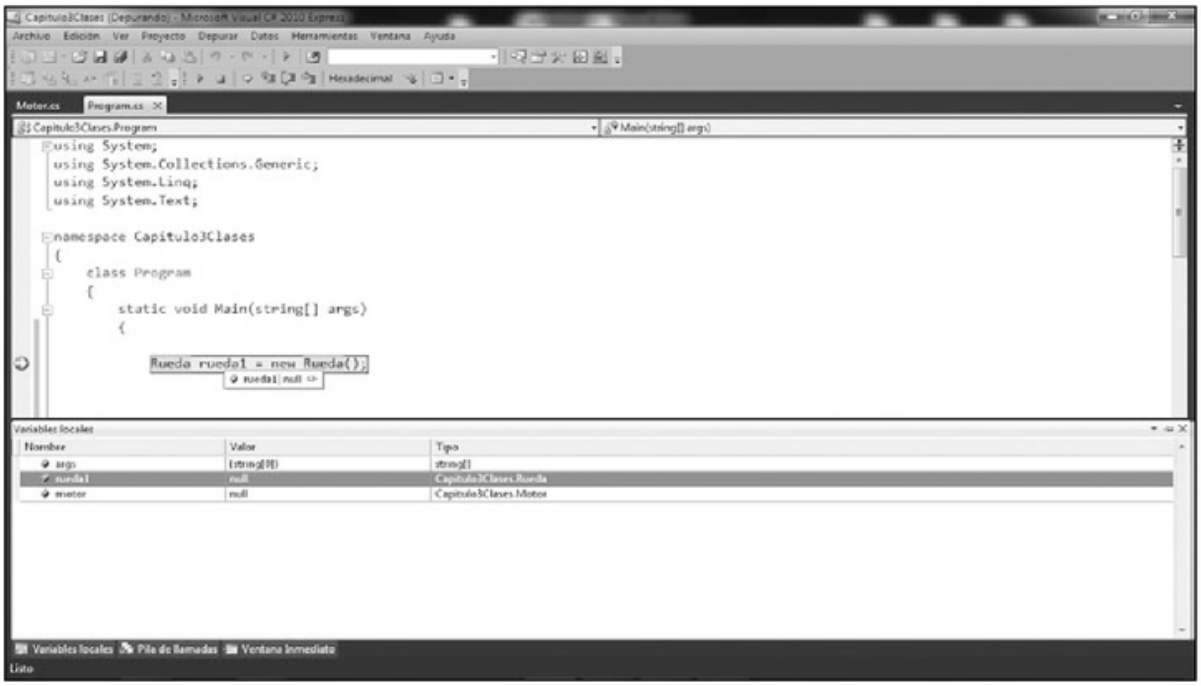


Figura 3. *Mientras una variable que contendrá un objeto no reciba una instancia de un nuevo objeto, seguirá manteniendo el valor* ***null****.*

Cada una de las variables que contengan una nueva instancia de la clase **Rueda** funcionarán de forma independiente, por lo tanto, cualquier modificación realizada a los atributos de una de estas, no modificará los atributos de las demás.

Esto es un punto clave en la programación orientada a objetos ya que garantiza que cada objeto sea independiente uno del otro, aunque sean una copia de la misma clase. Por otro lado, esto puede resultar algo confuso debido a que podríamos entender que la misma clase se está replicando en diferentes variables y que la funcionalidad está contenida en la clase y no, en el objeto.

Sin embargo, debemos ver el funcionamiento desde el punto de vista del objeto y no de la clase, por lo que podríamos tener cuatro objetos del tipo **Rueda**, uno para cada una de las ruedas de nuestro vehículo, y estos se comportarían de forma totalmente independiente uno de los otros, ya que la funcionalidad estaría replicada en cada uno de los objetos trabajando en forma independiente.

```
//Creamos cuatro
//objetos de la clase Rueda

Rueda rueda1 = new Rueda();
Rueda rueda2 = new Rueda();
Rueda rueda3 = new Rueda();
Rueda rueda4 = new Rueda();
```

Por lo tanto, teniendo cada uno de los objetos **Rueda** del anterior código, es posible asignar valores a cada uno de ellos, que resultarán valores únicos para cada uno de los objetos creados (**Figura 4**). Por más que los nombres de las variables sean idénticas para cada objetos, estas no serán compartidas entre objetos creados desde la misma clase.

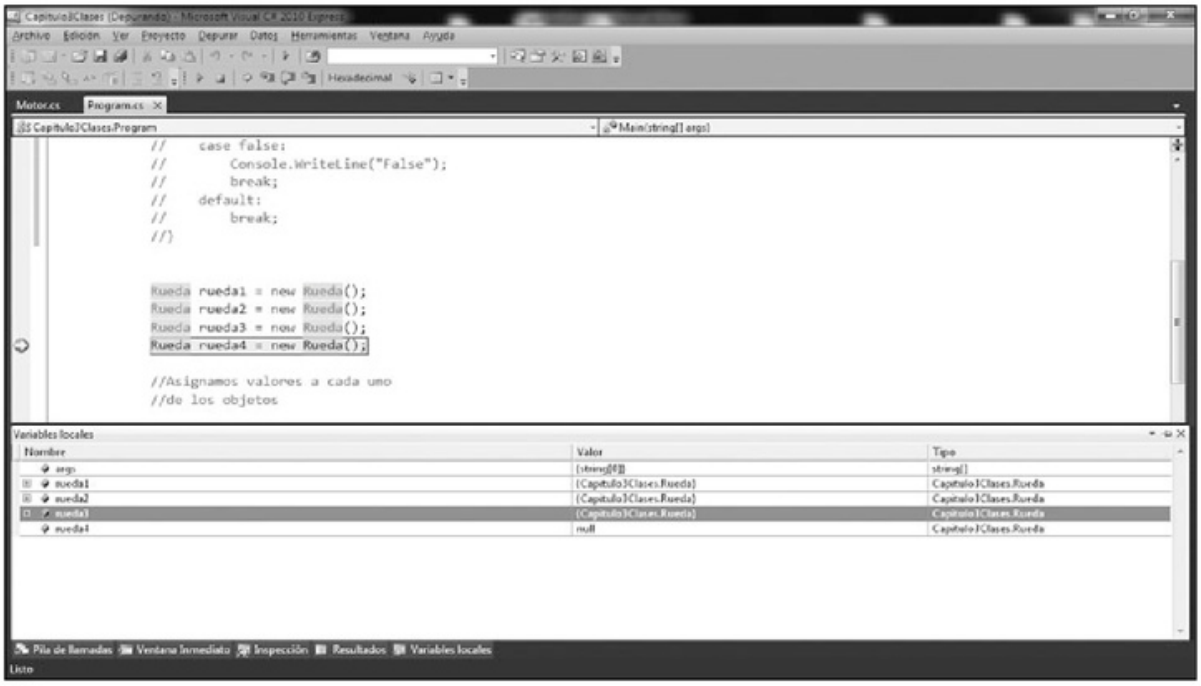


Figura 4. *Habiendo instanciado las tres primeras variables de tipo Rueda, la cuarta permanece no instanciada a pesar de ser del mismo tipo. Esto muestra la independencia de los objetos al momento de su creación.*

```
//Asignamos valores a cada uno
//de los objetos

rueda1.diametro = 10;
rueda2.diametro = 20;
rueda3.diametro = 15;
rueda4.diametro = 5;

//Mostramos cada valor
//conservando su independencia
//entre objetos

Console.WriteLine("Diámetro Rueda 1: " + rueda1.diametro);
Console.WriteLine("Diámetro Rueda 2: " + rueda2.diametro);
Console.WriteLine("Diámetro Rueda 3: " + rueda3.diametro);
Console.WriteLine("Diámetro Rueda 4: " + rueda4.diametro);
```

Podemos ver que cada objeto mantiene correctamente los valores asignados en cada caso, sin que alguna de las asignaciones sobrescriba los valores de los demás objetos como podemos ver en la **Figura 5**.

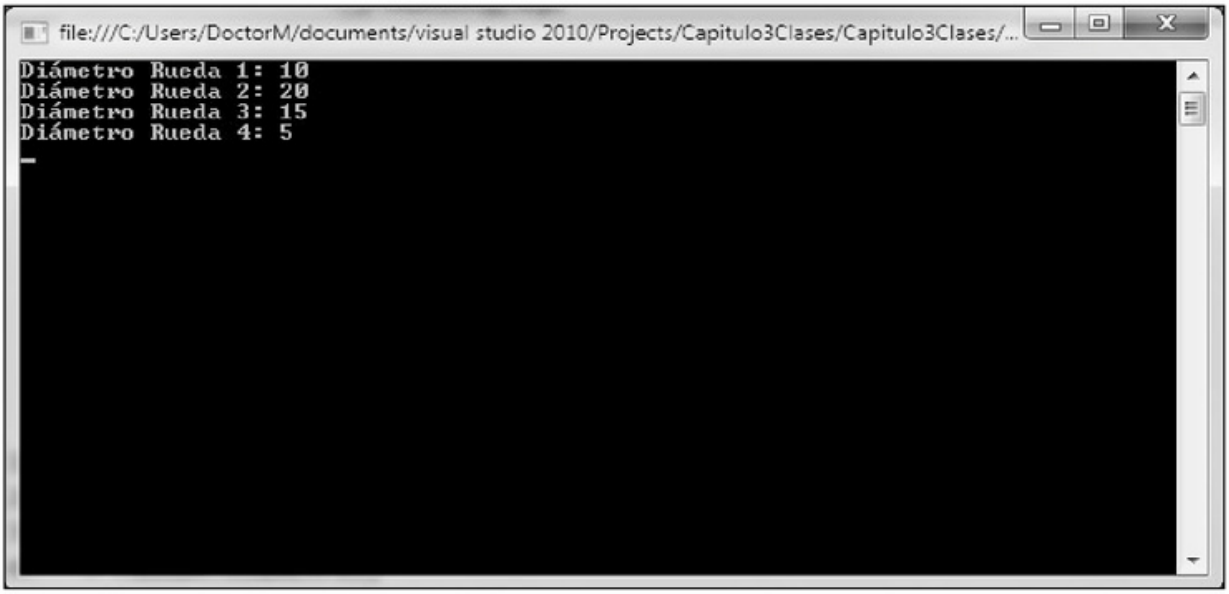


***Figura 5.** A pesar de que cada objeto es una instancia de la misma clase, estos respetan sus características como elementos únicos. La funcionalidad del objeto será la misma, pero su información será independiente.*

Junto con los nuevos operadores como el **new** para crear una nueva instancia de objetos, incorporamos otro elemento para la manipulación de los miembros de un objeto, en este caso, el símbolo de **.** (punto) seguido del nombre de la función, propiedad o atributo del objeto al que queremos acceder. Una ventaja del uso de las herramientas de desarrollo es que estas nos proveen de ayuda contextual a medida que escribimos el código, por lo tanto, notaremos que, al escribir el nombre de la variable por utilizar, y escribir el carácter de **.** (punto) se desplegará la lista de miembros, propiedades y funciones de ese objeto, a los cuales tenemos acceso y con los que podremos interactuar de alguna forma (**Figura 6**), esto puede ser un diferenciados en la rapidez con la cual podamos desarrollar nuestro código, además de la reducción de errores de escritura, más en lenguajes que diferencian mayúsculas de minúsculas.

III MIEMBROS DE UN OBJETO

Para poder interactuar con los distintos objetos creados en el código, estos deben exponer algún tipo de punto de contacto que sirva para ejecutar funcionalidad, leer datos o configurar el objeto. Algunos de estos puntos de contacto suelen ser variables accesibles desde fuera del objeto; estas reciben el nombre de **miembros del objeto**.

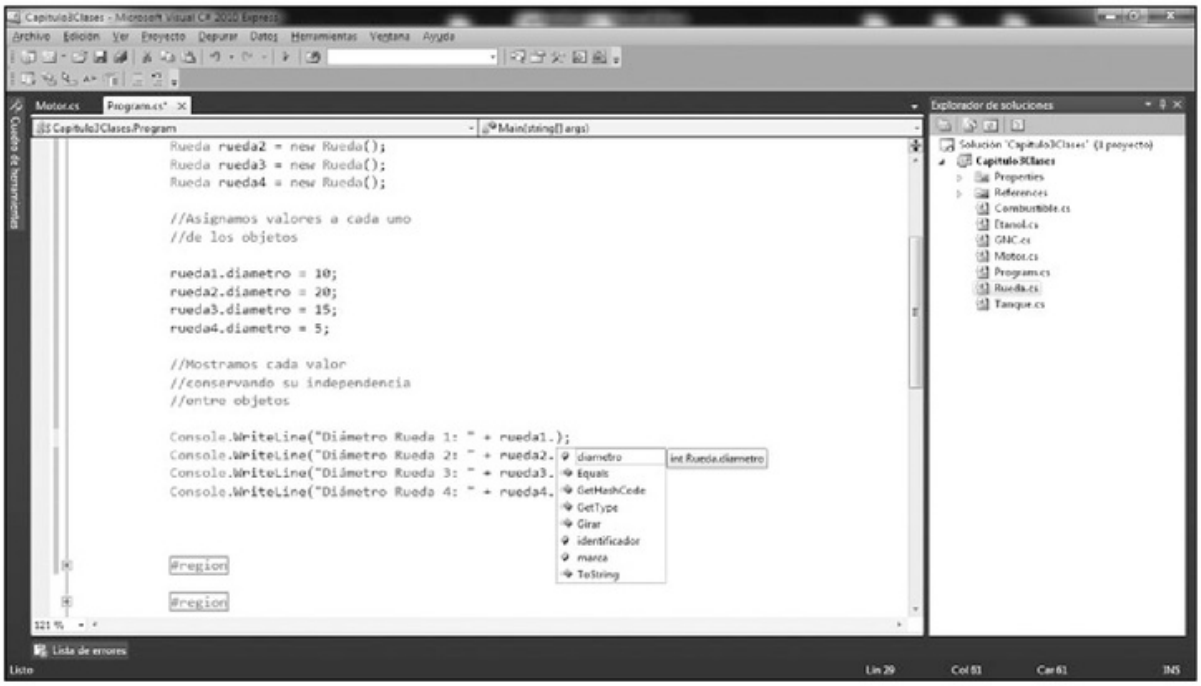


***Figura 6.** Al presionar el carácter de punto luego del nombre del objeto, la interfaz de desarrollo nos mostrará la lista de miembros disponibles del objeto con los que podremos interactuar desde nuestro código.*

Tipos por valor y referencia

Antes de proseguir con los conceptos de programación orientada a objetos es necesario que hagamos un breve repaso sobre los tipos de variables, sus valores y cómo se manejan internamente en la memoria, para entender las diferencias existentes entre las variables vistas en el **Capítulo 2** y las variables que contienen instancias de objetos como las que hemos desarrollado en este capítulo.

Por un lado, encontramos los **tipos por valor**. Estos tipos, o variables que contengan y estén enmarcadas dentro de esta categoría, serán manejados dentro de la memoria de la computadora de distinta forma que si fuese un **tipo por referencia**.

Para el primero grupo, los **tipos por valor**, las variables y sus valores son almacenados en la memoria de forma directa. Esto quiere decir que la memoria reservada para la variable estará ocupada desde el momento en el cual declaremos la variable, y sus valores ocuparán el espacio en ella en forma automática.

Esto los diferencia de los **tipos por referencia**, donde estos no ocuparán directamente el espacio en la memoria, sino que serán referenciados por medio de un **puntero** a los datos que estos representan. Para entender este último concepto, imaginemos a la memoria dividida en secciones, cada una de ellas representada por un número de acceso; este número es el punto de entrada a la información almacenada en la memoria, por lo que, para poder ver los datos en ese punto de la memoria, deberemos hacer referencia a dicho número y, por ende, a dicha posición. Por lo tanto, un **puntero de memoria** tiene la capacidad de apuntar a ese

espacio en memoria y así encontrar el camino hacia la información almacenada, y, si un puntero aún no apunta a ninguna información, la variable que contenga la representación de dicho puntero incluirá un valor nulo o **null** por su identificador sintáctico usado en C# (**Figura 7**).

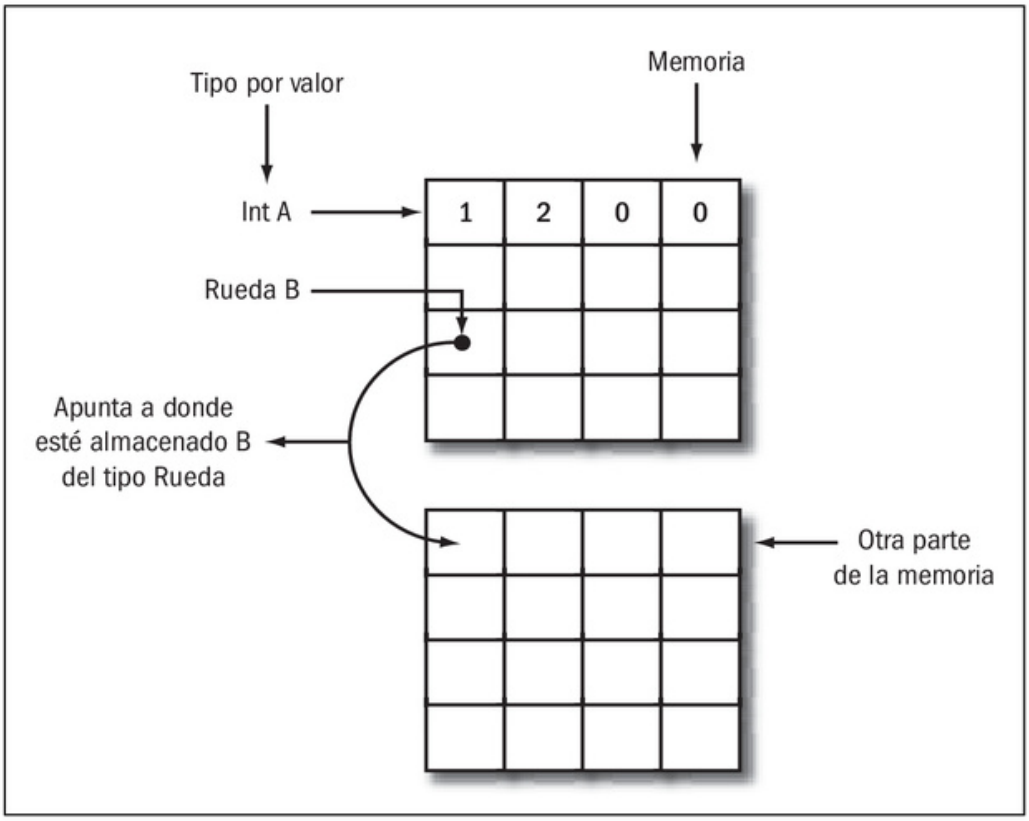


***Figura 7.** Los tipos por valor se almacenan directamente en la memoria, mientras que los tipos por referencia mantienen una referencia a la posición de la memoria donde están los datos.*

En la **Tabla 1**, podemos ver la lista de tipos tanto por referencia como por valor.

TIPOS POR VALOR	TIPOS POR REFERENCIA
Todos los tipos de datos numéricos.	El tipo de dato **string**.
Tipos **bool**, **char** y **date**. Este último para el manejo de fechas.	Todos los vectores y matrices. Incluso aquellos que sean creados a partir de tipos por valor.
Todas las estructuras de datos. Incluyendo aquellas que contengan tipos por referencia.	Todas las clases.
Las enumeraciones.	Los delegados.

***Tabla 1.** Una lista completa de los tipos por referencia y por valor encontrados dentro de Microsoft .Net Framework.*

Si tenemos esto en cuenta, notaremos comportamientos diferentes al momento de trabajar con variables creadas a partir de **tipos por valor** y **tipos por referencia**. Para

el primer caso, si igualamos una variable a otra, los valores de una son copiados a la segunda. Pero, en el caso de los tipos por referencia, si realizamos una operación de igualación, lo que se copia es la referencia de la primera a la segunda, por lo tanto, las dos variables apuntarán al mismo conjunto de datos (**Figura 8**).

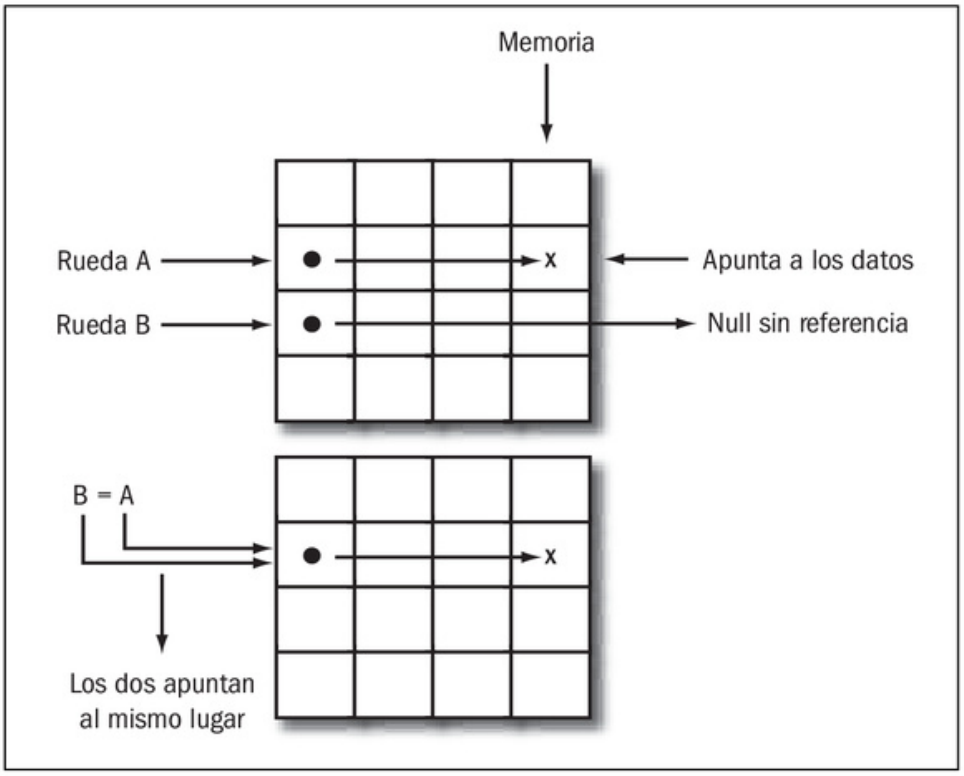


Figura 8. *La variable A posee una instancia y apunta a los datos, mientras que B no posee una instancia y apunta a null. Al copiar A en B, B apunta al mismo lugar que A, compartiendo los datos.*

Por ejemplo, veamos el código, que se encuentra a continuación, con la clase **Rueda** previamente creada y un tipo por valor cualquiera.

```
int tipoPorValor1 = 0;
int tipoPorValor2 = 0;
```

III LA MEMORIA EN C#

A diferencia de otros lenguajes orientados a objeto, C# y los demás lenguajes soportados por Microsoft .Net se consideran lenguajes manejados. En estos lenguajes, su motor de ejecución es el encargado de administrar la memoria, buscando los espacios libres así como liberando aquellos que ya no están en uso.

```
Rueda rueda1 = new Rueda();
Rueda rueda2 = new Rueda();
rueda1.diametro = 10;
rueda2.diametro = 20;

//Asignamos un valor a tipoPorValor1
//e igualamos a tipoPorValor2
tipoPorValor1 = 4;
tipoPorValor2 = tipoPorValor1;
Console.WriteLine("Valor de tipoPorValor2: " + tipoPorValor2);

//La modificación de tipoPorValor1
//no afecta tipoPorValor2, los valores fueron copiados
//no su referencia
tipoPorValor1 = 50;
Console.WriteLine("Valor de tipoPorValor2: " + tipoPorValor2);

//Mostramos el valor del objeto Rueda2
Console.WriteLine("Valor de rueda2: " + rueda2.diametro);

//Igualamos el objeto Rueda2 a Rueda1
//y mostramos su contenido
rueda2 = rueda1;
Console.WriteLine("Valor de rueda2: " + rueda2.diametro);

//Modificamos Rueda1 y mostramos el valor de Rueda2
//Los valores son modificados en las dos variables
rueda1.diametro = 50;
Console.WriteLine("Valor de rueda2: " + rueda2.diametro);
```

Como podemos ver en el código anterior, realizamos las mismas operaciones tanto para tipos por valor como por referencia, pero los resultados son completamente diferentes. Mientras que para el primer caso solo los valores contenidos por las variables son copiados a la segunda, en el caso de los tipos por referencia lo que se copia es el puntero que apunta a los datos, por lo tanto, si modificamos los valores en la primera variable, esta estará modificando los valores a los que apunta mediante el puntero, y por eso, todas las demás variables que apunten a la misma zona de la memoria también verán el cambio realizado.

Por último, vemos cómo la variable **rueda2** contiene los valores modificados por la variable **rueda1** sin necesidad de igualarlos (**Figura 9**).

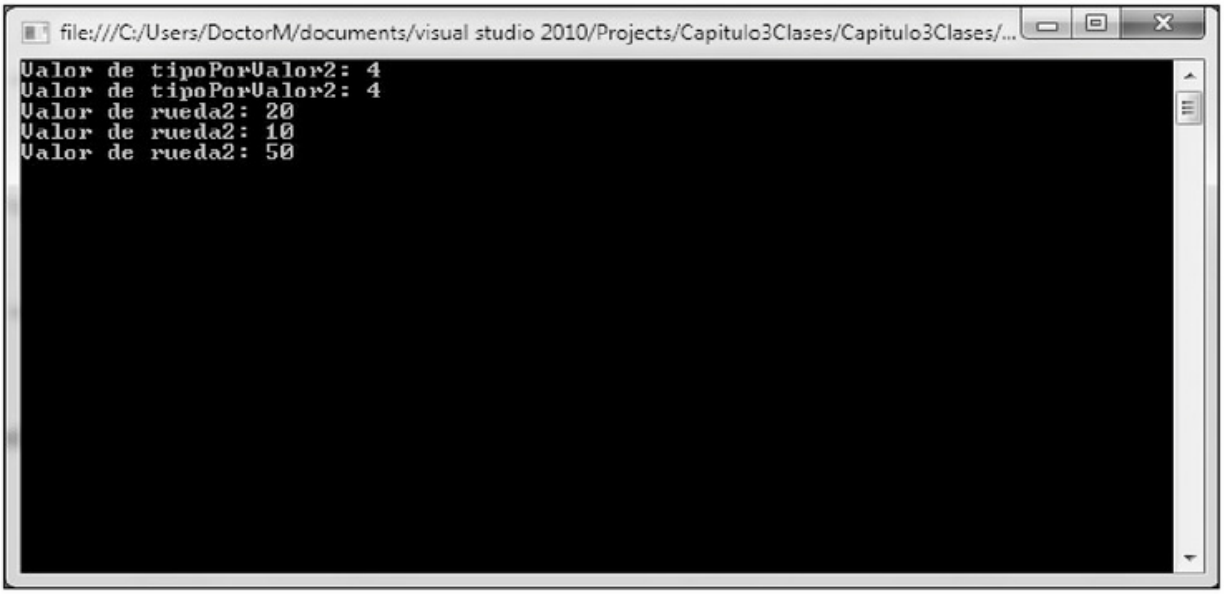


Figura 9. *Después de asignar los valores a la variable del tipo por valor, esta conserva su independencia.*

Parámetros por referencia

Si bien hemos visto la diferencia entre tipos por valor y tipos por referencia, es posible manipular los tipos por valor para que, en vez de copiar su contenido, lo que se manipule sea la referencia hacia la posición en la memoria en la cual se almacenan los datos, así, cualquier modificación realizada repercutirá en la variable original. Esta característica es posible lograrla cuando pasamos parámetros a una función y resulta de gran utilidad cuando, en el contexto desde donde se llama a la función, no queremos reemplazar el valor de una variable sino, simplemente alterar su contenido desentendiéndonos de lo que pase dentro de la función.

```
void Funcion1(ref int valor)
{
    valor += 10;
}
```

III VERIFICACIÓN POR VALOR

Los tipos numéricos por valor, como los tipos enteros, poseen un rango máximo y mínimo que pueden contener. Esto dependerá del tipo que elijamos. Si intentamos asignar un valor superior al que puede ser contenido, esto arrojará un error. Para omitir el error es posible enmarcar las operaciones aritméticas en un bloque **unchecked**.

Como vemos, la función toma un valor entero como parámetro de entrada. Podemos ver que el parámetro de entrada posee un descriptor antes del tipo del parámetro. El descriptor **ref** (referencia) hace que, al llamar a esta función, el parámetro pasado no envíe una copia de su contenido, sino que será pasada la referencia de la variable que contiene el valor, por lo tanto, al modificar el contenido de esta referencia, también se verá modificado el contenido de la variable que fuese pasada por referencia.

```
int valorReferencia = 10;
ParametrosReferencia param = new ParametrosReferencia();

//Llamamos a la función Funcion1
//y pasamos el valor por referencia
param.Funcion1(ref valorReferencia);

Console.WriteLine(valorReferencia);
```

Al momento de llamar a la función que acepta parámetros por referencia, también es necesario aclarar que dicho parámetro es pasado por referencia mediante el uso de la misma palabra reservada **ref**. El resultado final de la ejecución de este código será que, al valor inicial de **10** contenido por la variable **valorReferencia**, se le adicionen **10** más, y siendo que esta variable es pasada por referencia, la suma afectará el contenido de dicha variable, terminando con el valor de **20** al imprimir su contenido (**Figura 10**).

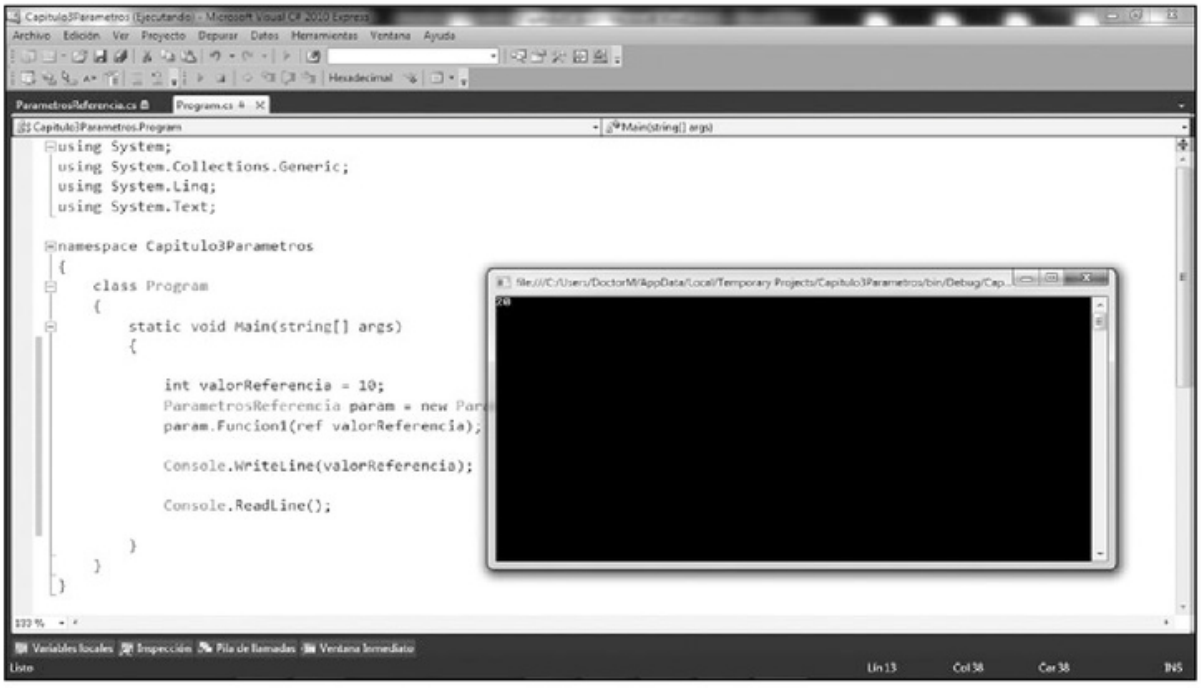


Figura 10. *La variable pasada por referencia con un valor inicial de 10, al salir de la función, contiene el resultado de la suma.*

Parámetros de salida

Recordemos que las funciones solo pueden retornar un único valor, por lo tanto, si quisiéramos devolver más información en una llamada a una función deberíamos retornar, por ejemplo, un vector o alguna colección de datos. Pero esto podría requerir de líneas de código adicionales para verificar la cantidad de parámetros retornados o intentar adivinar en qué posición de dicho vector se encuentra el parámetro que estamos buscando. Para los casos más simples, en los que no necesitemos complicarnos con lo planteado antes, contamos con los parámetros de salida. Estos parámetros son declarados junto con los parámetros normales de una función, pero utilizando el identificador **out** (salida) por delante del tipo del parámetro de entrada.

```
public bool IntentarSumar(int valor1, int valor2,
    out int resultado)
{
    resultado = 0;

    if (valor1 > 0)
    {
        resultado = valor1 + valor2;
        return true;
    }
    else
    {
        return false;
    }
}
```

En el código, se propone una función que intente sumar dos números siempre y cuando el primer valor sea mayor a cero. Notemos que retornará un tipo booleano diciendo si pudo o no realizar la suma. En este caso, si la suma fuese realizada correctamente, además de avisar que se ha realizado con éxito mediante el resultado booleano, es necesario devolver el resultado de la operación. En este caso debemos usar el parámetro del tipo **out**, donde almacenaremos el resultado de la operación.

```
int resultado = 0;
bool pudoSumar = false;

//Hacemos que la función se ejecute correctamente
pudoSumar = param.IntentarSumar(10, 10, out resultado);
```

```
if (pudoSumar)
{
    Console.WriteLine("Resultado de la suma: " + resultado);
}

//Hacemos que la función falle
pudoSumar = param.IntentarSumar(0, 10, out resultado);
if (!pudoSumar)
{
    Console.WriteLine("No se pudo sumar. Resultado de la suma: " + resultado);
}
```

Como vemos, al igual que con la palabra reservada **ref**, también es necesario escribir **out** delante de la variable que será pasada para este caso. Para probar el funcionamiento de la función, se ejecuta dos veces. En el primer caso hacemos que la operación de sumar se ejecute, por lo que obtendremos como retorno de la función el valor de **true** y en el parámetro **out** el resultado de la suma; luego se evalúa la variable booleana y se escribe el resultado contenido en la variable **resultado**.
En el segundo caso, la llamada a la función retorna **false**, y se procede a realizar la misma evaluación. Notemos que la condición hace uso del carácter de negación **!** para invertir la condición y así mover la ejecución del código a la sección verdadera de la condición. Finalmente, veremos el aviso de que la operación no se pudo realizar, y se muestra el resultado contenido en la variable **resultado** (**Figura 11**).

Figura 11. *Los parámetros de salida out son retornados con el valor resultante de la operación además de la variable booleana devuelta por la función.*

Encapsulamiento

Una cualidad de los objetos es el **encapsulamiento**. De hecho, este concepto es uno de los pilares de la POO y que permite la creación de objetos con cierta abstracción en su funcionalidad. El encapsulamiento, por lo tanto, hace referencia a la capacidad de encapsular o aislar código, dejándolo inaccesible para otros objetos externos y, al mismo tiempo, exponiendo solo aquellos puntos de acceso que nosotros, como desarrolladores creamos convenientes. En el caso de nuestra clase **Rueda**, como la hemos planteado inicialmente, nos daremos cuenta de que los miembros de esta, **diametro**, **identificador** y **marca**, son, en realidad, inaccesibles desde nuestro código. Esto se debe a que, por la forma en cómo fueron escritas, C# las considere como elementos internos de la clase y no visibles hacia los objetos o líneas de código que pudieran hacer uso de esta clase. Por lo tanto, para lograr que estos miembros sean visibles, deberemos realizarle algunos cambios a nuestro código.

```
class Rueda
{

    ...
    public int diametro;
    public int identificador;
    public string marca;
    ...

}
```

Como vemos en el código anterior, se ha agregado un **modificador de acceso** a cada uno de los miembros de nuestra clase. Este modificador llamado **public** (público) nos dice que el acceso al miembro de la clase será total, público para todos los demás objetos o líneas de código que interactúen con la clase **Rueda**. Existen otros modificadores de acceso que nos permiten poner en práctica el encapsulamiento; en la **Tabla 2**, podemos ver la lista de los modificadores disponibles.

TIPOS OUT

La utilización de **out** es común dentro del Microsoft .Net Framework. Este suele ser utilizado para probar la conversión de valores de un tipo a otro. En estos casos, es necesario obtener un valor de salida de una función si se ha podido convertir el valor y, además, un valor booleano que confirme si la conversión fue válida. Un ejemplo de esta implementación es **int.TryParce**.

MODIFICADOR DE ACCESO	DESCRIPCIÓN
public	Las variables, propiedades y funciones son accesibles desde cualquier otro objeto.
private	Las variables, propiedades y funciones solo son accesibles desde dentro del mismo objeto, no siendo visibles desde fuera de este.
internal	Solo es posible acceder a estos elementos desde la misma clase, o desde cualquier clase derivada de esta.
protected	Los miembros pueden ser accedidos solo por código que comparte el mismo ensamblado.

***Tabla 2.** Lista de identificadores de acceso para la implementación del concepto de encapsulamiento en C#.*

La clase **Rueda** podría contener funcionalidad específica para manipular su información encapsulando parte del código que el usuario no debería manipular.

```
public void Girar()
{
     //Código para hacer girar la rueda
     int velocidadDeGiro = Revoluciones();
}

private int Revoluciones()
{
     //Retorna la cantidad de giros
     //a realizar bajo determinada condición
     return 10;
}
```

Entendemos que un posible cálculo de las revoluciones de giro que el objeto **rueda** lleva a cabo solo es útil para calcular una hipotética velocidad al momento de que la rueda gire, por lo tanto, es innecesario que el código externo a la clase **Rueda** pueda interactuar con esta función. Al colocar el modificador de acceso **private** (privado) por delante de la función, esta dejará de ser accesible (**Figura 12**).

III MODIFICADOR DE ACCESO

Cuando escribimos código en C#, muchas veces omitimos el modificador de acceso delante de las declaraciones. En muchos casos, es la herramienta de desarrollo la que omite escribir este modificador. Cuando esto pase, todos los tipos declarados serán tratados como si hubieran sido definidos como **protected** o protegidos.

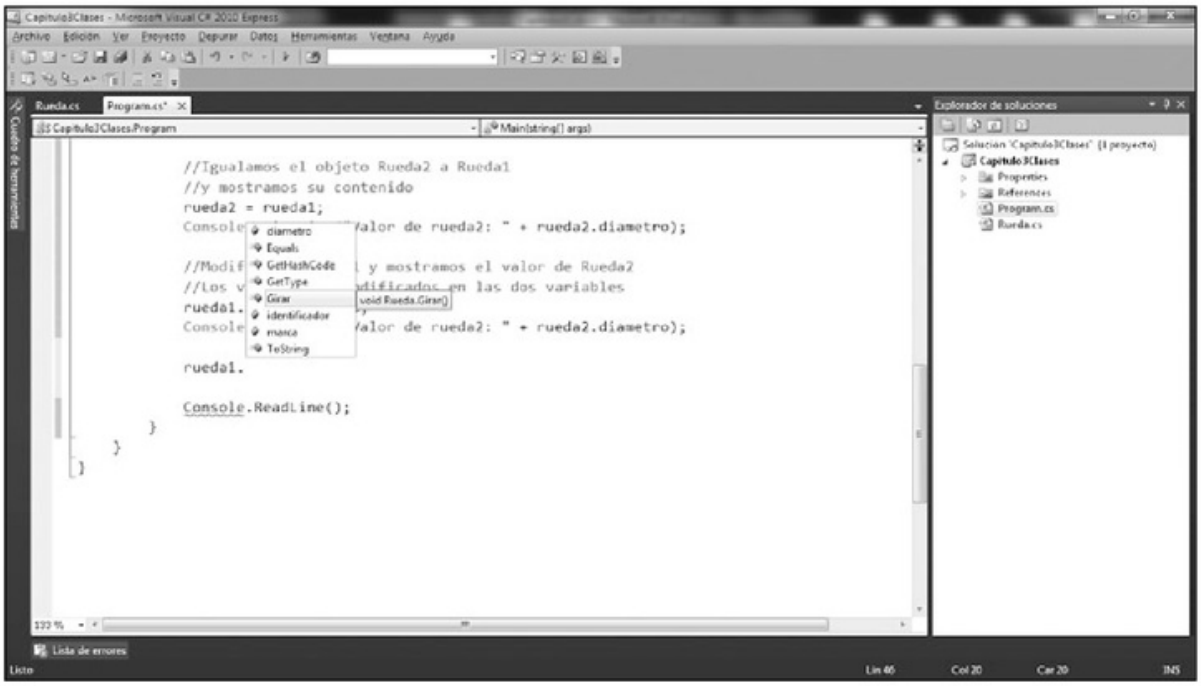


Figura 12. *La ayuda de contexto al escribir código nos muestra solo aquellos miembros de la clase que no están marcados con el modificador de acceso* ***private****.*

Herencia

En ocasiones, nos encontraremos con la necesidad de crear nuevas clases que requieran cierto comportamiento similar a otras clases. Podríamos implementar la clase **Motor** de nuestro vehículo, donde requiera, para poder trabajar correctamente, una cantidad específica de combustible.

Existen diferentes tipos de combustibles, pero todos poseen características similares, el octanaje, por ejemplo, es una de las cualidades de un combustible, aunque podamos, dentro de los combustibles encontrar diferentes tipos.

Por lo tanto, podríamos crear una clase que representara a los combustibles de forma general, con sus características comunes, y heredar estas cualidades desde cada uno de los diferentes tipos de combustibles que tengamos disponibles. Esta característica es otro de los pilares de la POO, que nos da la posibilidad de reutilizar código desde otras implementaciones sin tener que escribir una y otra vez la misma funcionalidad, además de poder crear nuevas capas de funcionalidad sobre las heredadas.

```
class Combustible
{
    protected int octanaje;

    public int Octanaje()
    {
```

```
        return octanaje;
    }
}
```

El primer paso es crear una clase que servirá como base para las demás clases. La clase **Combustible**, entonces, servirá para proveer el nivel de octanaje del combustible elegido. El siguiente paso consiste en crear diferentes combustibles que hereden estas características, veamos el ejemplo, a continuación.

```
class GNC : Combustible
{
    public GNC()
    {
        base.octanaje = 30;
    }
}

class Etanol : Combustible
{
    public Etanol()
    {
        base.octanaje = 100;
    }
}
```

En este código, encontramos dos nuevos elementos de la sintaxis de C#. Por un lado, el uso del carácter **:** (dos puntos) después del nombre de clase. Esto es utilizado para decir que tanto la clase **Etanol** como **GNC** heredan de **Combustible**. Al hacerlo, estas pueden acceder a todas las funciones, variables y atributos que contenga la clase base. Por supuesto, debemos recordar la funcionalidad de los distintos modi-

III CLASE BASE

El nombre de clase base hace referencia a la clase que está por debajo de otra. Imaginemos que, en una herencia, la clase heredada queda por debajo de la que hereda, por lo tanto, la primera es la base de construcción de la segunda. Esta es la razón por la que usamos la palabra reservada **base** para acceder esta clase que da sustento a la nueva.

ficadores de acceso ya que, dependiendo de ellos, estos elementos serán o no accesibles desde la clase que hereda a la otra.
El segundo nuevo elemento de sintaxis es la palabra reservada **base**. Mediante este es posible acceder al conjunto de propiedades, funciones o variables de la clase en la cual nos estamos basando para tomar funcionalidad. De esta forma, la clase **GNC** hará referencia a lo provisto por **Combustible**. En la **Figura 13**, vemos un diagrama de clases que muestra la relación existente entre la clase base y las que heredan de ella.

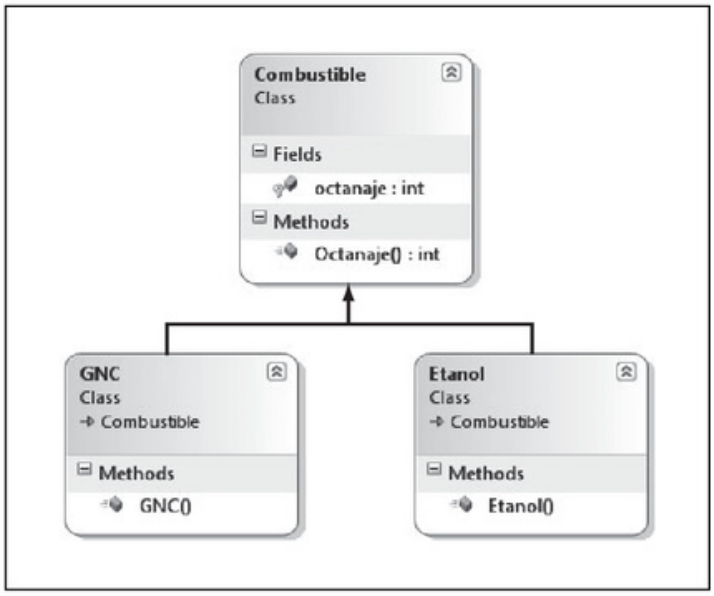


Figura 13. *El diagrama de clases muestra cómo las dos clases que representan diferentes tipos de combustibles, heredan funcionalidad desde la clase base* ***Combustible****.*

Por lo tanto, si declaramos una variable del tipo **GNC** o **Etanol**, notaremos que las propiedades de la clase base también están presentes.

```
//Declaramos una variable
//para GNC y verificamos su octanaje
GNC gnc = new GNC();
Console.WriteLine("Octanaje de GNC: " + gnc.Octanaje());
//Declaramos una variable
//para ETANOL y verificamos su octanaje
Etanol etanol = new Etanol();
Console.WriteLine("Octanaje de ETANOL: " + etanol.Octanaje());
```

A pesar de tener la misma funcionalidad, al imprimir los valores devueltos por los dos elementos, notamos que cada uno de ellos conserva su independencia, mostrando un valor de octanaje diferente para cada caso (**Figura 14**).

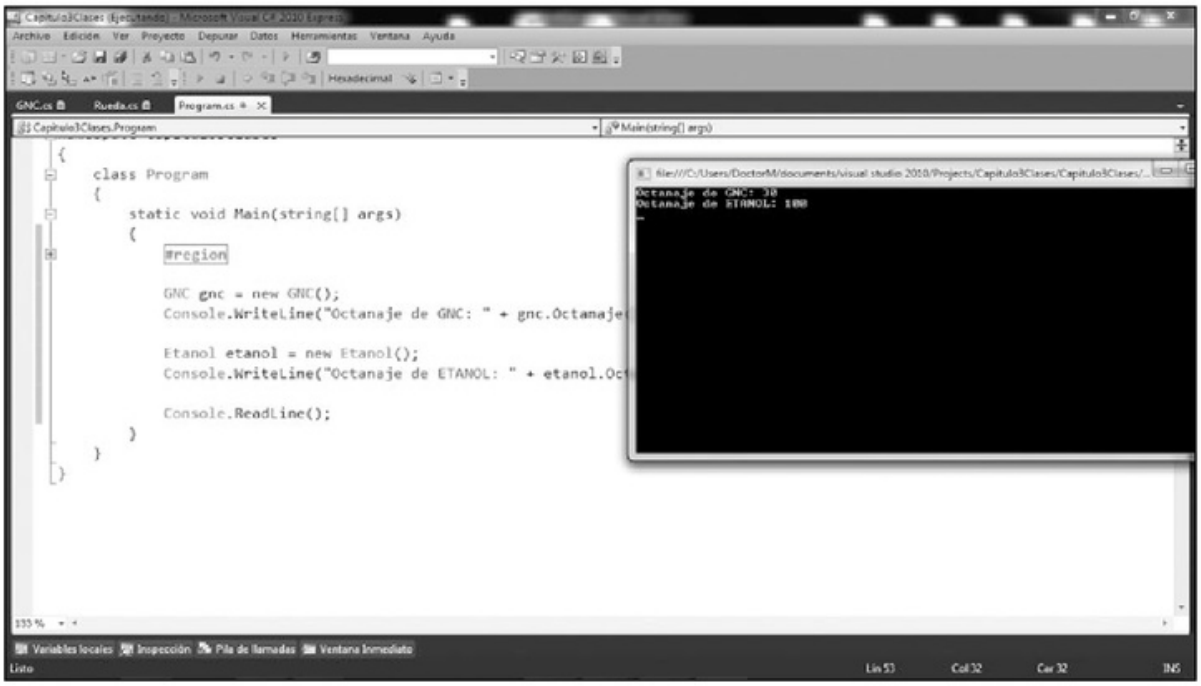


Figura 14. *Al mostrar en la consola el contenido de cada objeto, estos escriben el octanaje usando la función de la clase base, pero con los valores de cada uno de los objetos.*

Polimorfismo

El último pilar en la POO es el **polimorfismo**. Este plantea dos posibilidades. Por un lado, el poder crear múltiples funcionalidades bajo un mismo nombre y, por otro, la característica de los objetos de mutar, cambiar su forma, sobre la base del tipo creado. Si tomamos el ejemplo anterior, cuando hablábamos de herencia, podemos extenderlo para demostrar esta posibilidad de mutación de los objetos.

```
//Creamos una variable desde la clase
//base Combustible
Combustible combustible;

//Asignamos un nuevo objeto
//del tipo GNC y mostramos sus valores
combustible = new GNC();
Console.WriteLine("Octanaje GNC: " + combustible.Octanaje());

//Redefinimos el objeto
//al tipo Etanol y mostramos sus valores
combustible = new Etanol();
Console.WriteLine("Octanaje Etanol: " + combustible.Octanaje());
```

En este ejemplo de código de la página anterior, vemos cómo el primer paso es la declaración de una variable del tipo **Combustible**. Este tipo, como vimos en la herencia, es la base de los demás combustibles, por lo tanto, esta podrá ser igual, o contener, cualquier otro objeto que herede de dicha clase base.
Así, **GNC** y **Etanol** heredan de **Combustible**, por lo tanto, pueden ser manejados por esta variable. Como es la clase base la que posee la funcionalidad de mostrar el valor de octanaje, esta función estará disponible incluso si la clase es del tipo base. Al mostrar los valores de cada tipo, luego de crear una nueva instancia del combustible seleccionado, los valores mostrados siguen siendo los correctos, como vemos en la **Figura 15**.

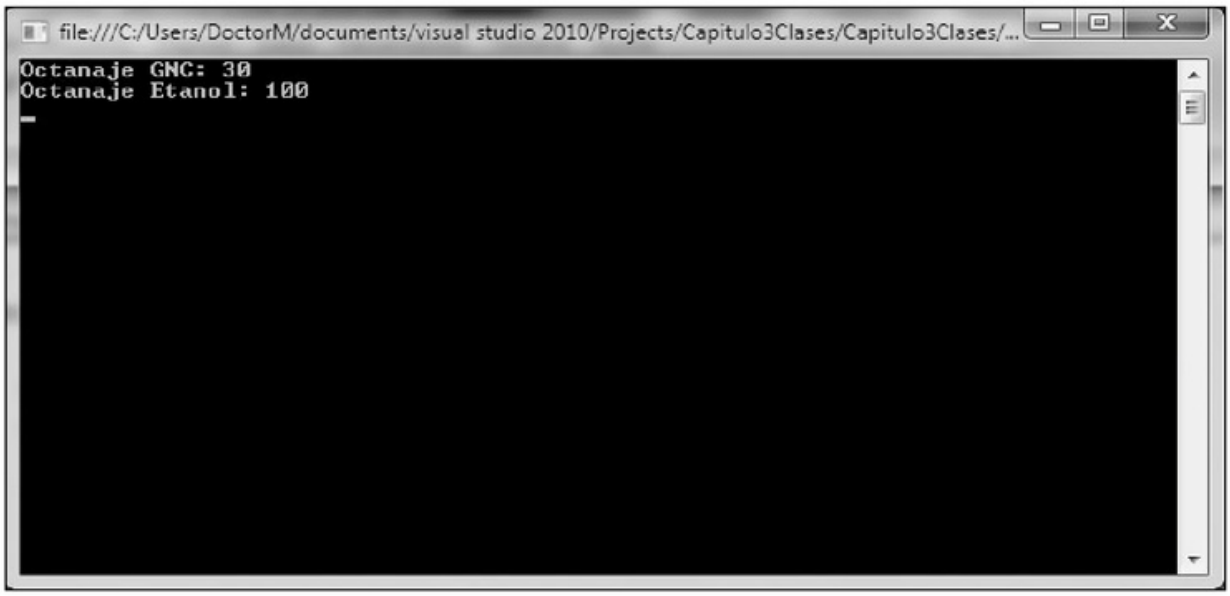


Figura 15. *Se muestran los octanajes para los dos tipos de combustibles, a pesar de que la variable es declarada como tipo base y no específico.*

Para comprender mejor el polimorfismo, agreguemos un elemento más al conjunto de clases que conforman nuestro vehículo. Ahora sumaremos una clase que represente el tanque donde se almacenará el combustible. Debido a que en el **Tanque** podemos almacenar cualquier tipo de combustible, en este ejemplo no necesitamos restringir el tipo de combustible por almacenar.

```
class Tanque
{
    private Combustible tipoCombustible;
    private int capacidadTanque;

    public void TipoCombustible(Combustible tipo)
```

```
    {
        tipoCombustible = tipo;
    }

    public void CantidadAlmacenada(int capacidad)
    {
        capacidadTanque = capacidad;
    }
}
```

La función **TipoCombustible** recibe como parámetro un **Combustible** y no un tipo específico. Debido a esto, es posible almacenar cualquier tipo de combustible que herede de la clase base **Combustible**.

```
GNC gnc = new GNC();
Etanol etanol = new Etanol();
Tanque tanque1 = new Tanque();
Tanque tanque2 = new Tanque();

tanque1.CantidadAlmacenada(100);
tanque1.TipoCombustible(gnc);

tanque2.CantidadAlmacenada(200);
tanque2.TipoCombustible(etanol);
```

Para modificar la clase **Tanque**, agregaremos una nueva función que muestre el tipo de combustible que se ha pasado como parámetro.

```
public void MostrarTipoCombustible()
{
    Console.WriteLine("Tipo de combustible cargado: " +
        tipoCombustible.GetType().ToString());
}
```

En el código de ejemplo, se le pide al objeto el tipo que contiene mediante **GetType()**, para luego imprimirlo en la consola. A pesar de que la variable **tipoCombustible** es del tipo **Combustible**, tanto para **tanque1** como para **tanque2**, los tipos pasados son **GNC** y **Etanol**; por lo tanto, al intentar imprimir su verdadero tipo de datos, obtendremos los

anteriores mencionados y no el tipo **Combustible**, desde donde heredan las otras dos clases, como vemos en el código que se encuentra a continuación.

```
//Llamamos a la función que imprime
//el tipo de combustible pasado
//por parámetro
tanque1.MostrarTipoCombustible();
tanque2.MostrarTipoCombustible();
```

Al haber pasado al objeto **tanque1** el tipo **GNC** y a **tanque2** el tipo **Etanol**, podemos ver cómo estos tipos son impresos de forma correcta (**Figura 16**).

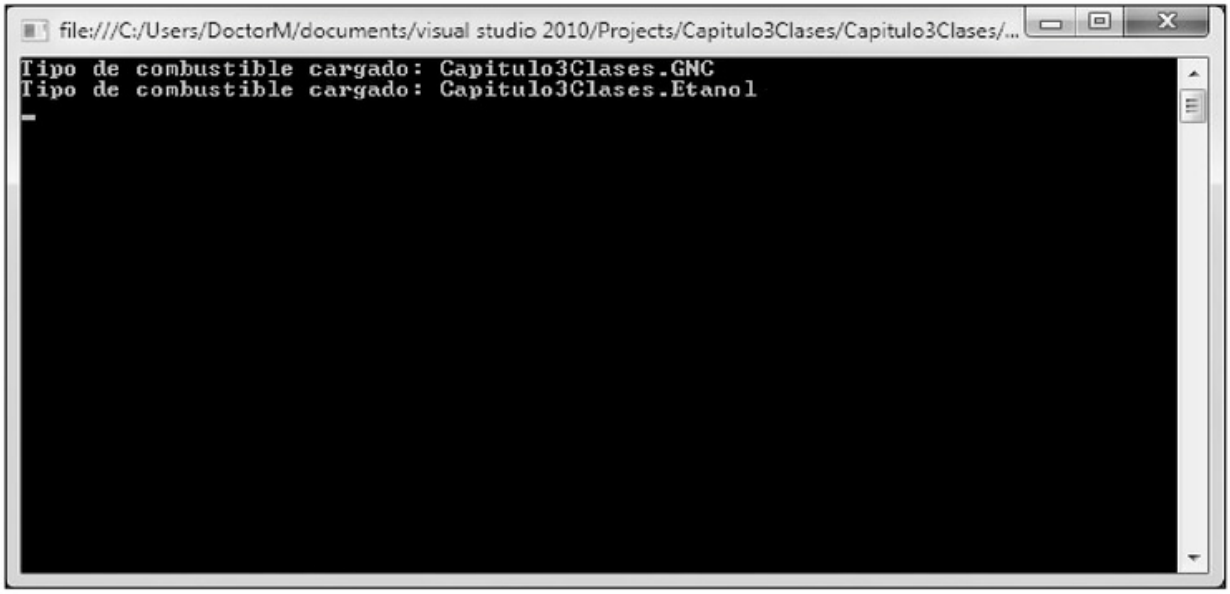


Figura 16. *Se muestran los tipos específicos pasados por parámetro a cada objeto Tanque. Además del tipo, se observa el espacio de nombre al cual pertenece.*

La segunda característica del polimorfismo es la capacidad de tener la misma función, o el mismo nombre para esta, pero con parámetros del todo diferentes, pudiendo realizar distintas funcionalidades.

```
public void ConfigurarTanque()
{ }

public void ConfigurarTanque(int capacidad)
{ }
```

```
public void ConfigurarTanque(int capacidad, Combustible tipo)
{ }
```

En el código anterior, agregamos tres nuevas funciones a la clase **Tanque**. El nombre de esta función es idéntico para los tres casos, pero difiere en la cantidad de parámetros. Sin recibir parámetros en el primer caso, solo la capacidad del tanque en el segundo, y la capacidad más el tipo de combustible en el tercero. Al momento de utilizar un objeto creado a partir de la clase **Tanque** y de intentar utilizar esta función, veremos que se nos sugieren las tres posibilidades escritas en el código (**Figura 17**).

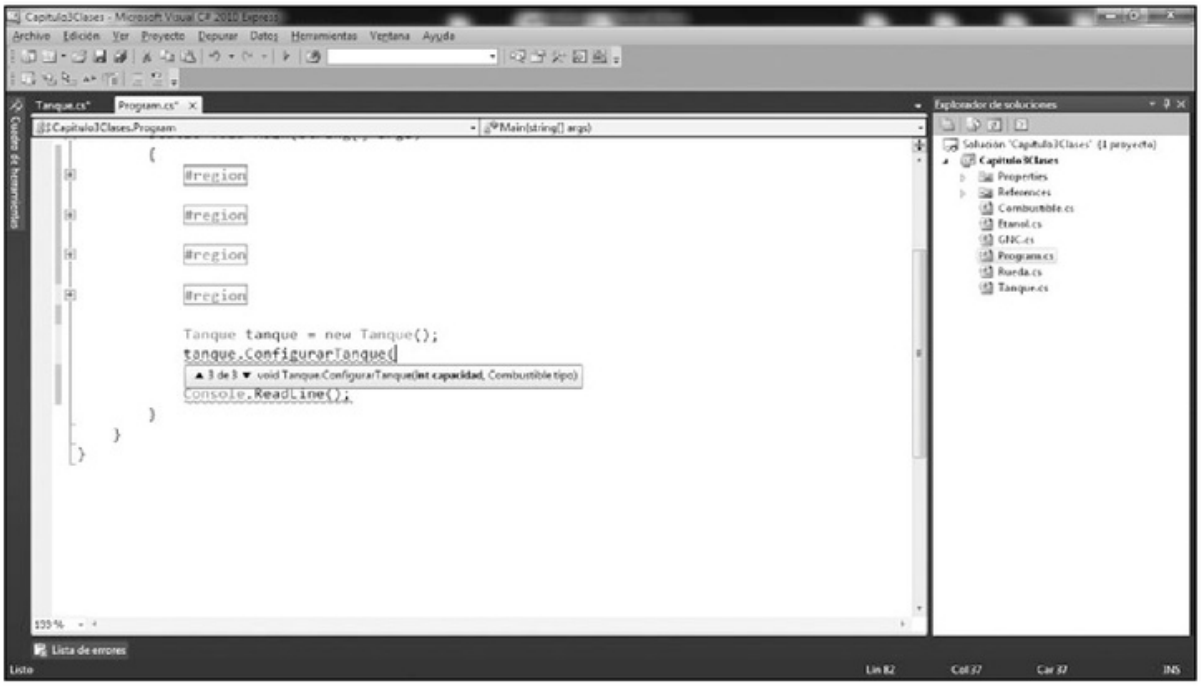


***Figura 17.** Al tener tres funciones iguales con parámetros diferentes, podremos ver en la ayuda que las tres funciones se encuentran disponibles para su uso.*

Esto resulta útil en especial en casos donde no necesitemos, en forma exclusiva, que los valores para una clase sean provistos externamente, siendo que nosotros podemos proveer algunos por defecto si estos no existiesen.

```
public void ConfigurarTanque()
{
    capacidadTanque = 100;
    tipoCombustible = new Etanol();
}

public void ConfigurarTanque(int capacidad)
```

```
{
    capacidadTanque = capacidad;
    tipoCombustible = new Etanol();
}

public void ConfigurarTanque(int capacidad, Combustible tipo)
{
    capacidadTanque = capacidad;
    tipoCombustible = tipo;
}
```

Como observamos, si el código que utiliza la clase **Tanque** al llamar a la función **ConfigurarTanque** no provee ningún parámetro, asumimos que el tanque tendrá una capacidad de **100** y el tipo de combustible será **Etanol**. En el caso de que sí provea una capacidad, entonces usamos esa variable como valor interno del objeto, pero seguimos considerando que **Etanol** es el combustible que se incluirá. Por último, si el código pasa todos los valores necesarios al objeto **Tanque**, entonces tomamos esos como válidos para el funcionamiento.

VIDA Y MUERTE DE UNA CLASE

Parte de la vida de los objetos transcurre entre su creación y su destrucción. Para cada uno de estos estados, existe una sintaxis especial que nos permitirá colocar código para realizar acciones que ayuden a su gestión.

Constructores

Si nos remitimos en este mismo capítulo al momento en el que hablábamos sobre herencia, podemos ver que las clases utilizadas allí tenían funciones que no concordaban con el patrón que hemos visto hasta ahora. Estas funciones, por un lado, tenían el mismo nombre que el nombre de la clase y, por otro, no retornaban ningún tipo, incluso no utilizaban la palabra reservada **void** para especificar que esto sería así. Este tipo de funciones poseen el nombre de **constructores**, y esto se debe a que se ejecutan en el momento en el que el objeto es construido mediante el uso de la palabra reservada **new** (nuevo). Por supuesto, esto puede resultar especialmente útil si nuestro objeto requiere configurar valores iniciales al momento de su construcción u obligar a aquel que lo utilice a pasarle determinados parámetros para garantizar la correcta construcción.

```
class Motor
{
     public Motor()
     {
              //Escribimos un mensaje
              //en el momento de la construcción
              Console.WriteLine("El motor se ha creado");
      }
}
```

La clase **Motor** posee un constructor sin parámetros. Este también es considerado el **constructor por defecto** y es utilizado cuando creamos una instancia de esta clase como hemos estado haciendo hasta el momento. La ausencia de un constructor de este tipo hace que el motor de ejecución cree uno por nosotros, aunque el código no esté presente, haciendo que este constructor por defecto exista. Un mensaje será desplegado en la consola al momento de crear el objeto **Motor** (**Figura 18**).

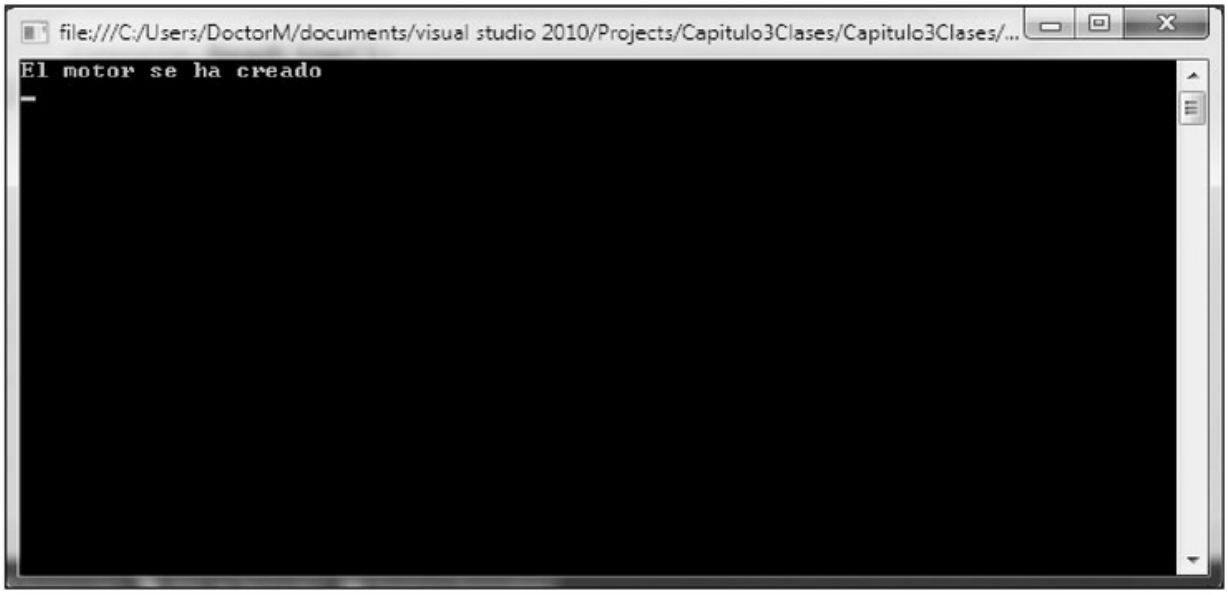


Figura 18. *En el momento de construir un objeto del tipo Motor, el constructor es disparado, y un mensaje es mostrado en la consola.*

Los constructores también están sujetos a la propiedad de polimorfismo, por lo que se podrán crear diferentes constructores con distintos parámetros.

```
public Motor()
{
```

```
        pistones = 4;
        cc3 = 1200;
    }

    public Motor(int numeroDePistones)
    {
        pistones = numeroDePistones;
        cc3 = 1200;
    }

    public Motor(int numeroDePistones, int Cc3)
    {
        pistones = numeroDePistones;
        cc3 = Cc3;
    }
```

Como ya hemos visto, al momento de interactuar con la clase **Motor** y de intentar la creación de una nueva instancia, se nos mostrará una lista de las posibles alternativas para construir el objeto, pasando ninguno o hasta dos parámetros como posibles opciones, cada una correspondiente a los distintos constructores de la clase, recomendamos revisar la **Figura 19**.

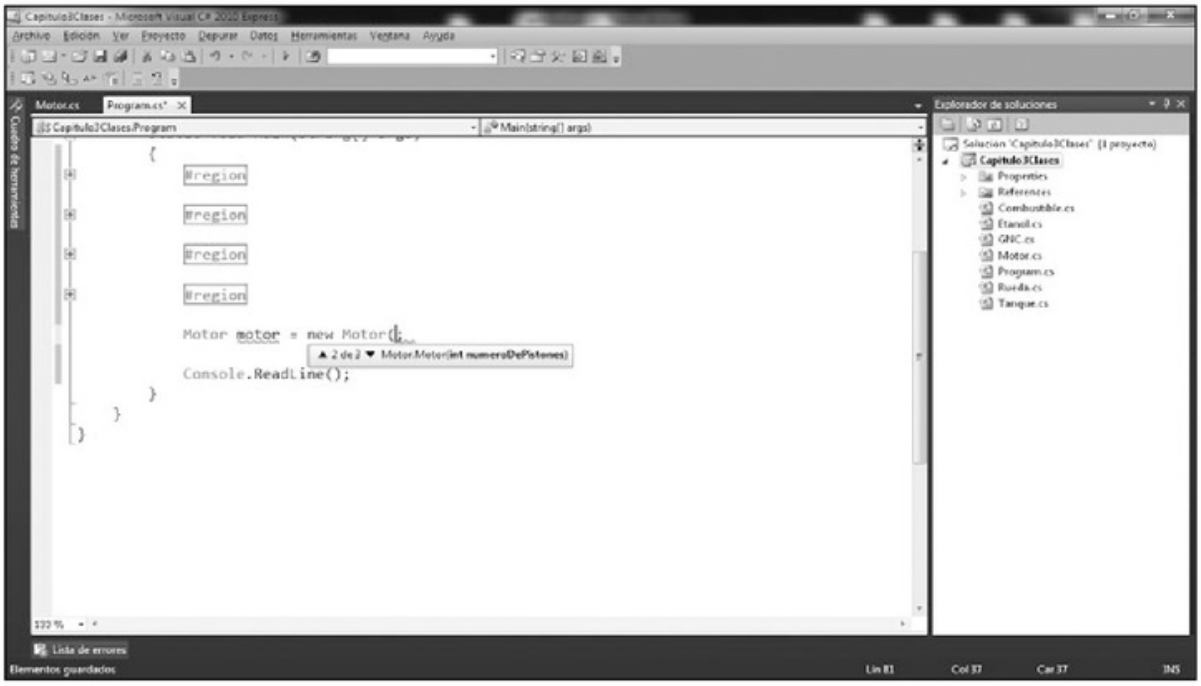


Figura 19. *Al momento de intentar construir el objeto* ***Motor*** *mediante el uso de la palabra reservada* ***new****, las tres alternativas de construcción aparecen como disponibles, pudiendo elegir una de ellas.*

Cabe recalcar que, al escribir un constructor diferente del constructor por defecto, esto es, un constructor con por lo menos un parámetro, el motor de ejecución ya no creará por nosotros el constructor por defecto, por lo que, de no estar presente, solo contaremos con los constructores con parámetros al momento de querer crear un nuevo objeto de ese tipo (**Figura 20**).

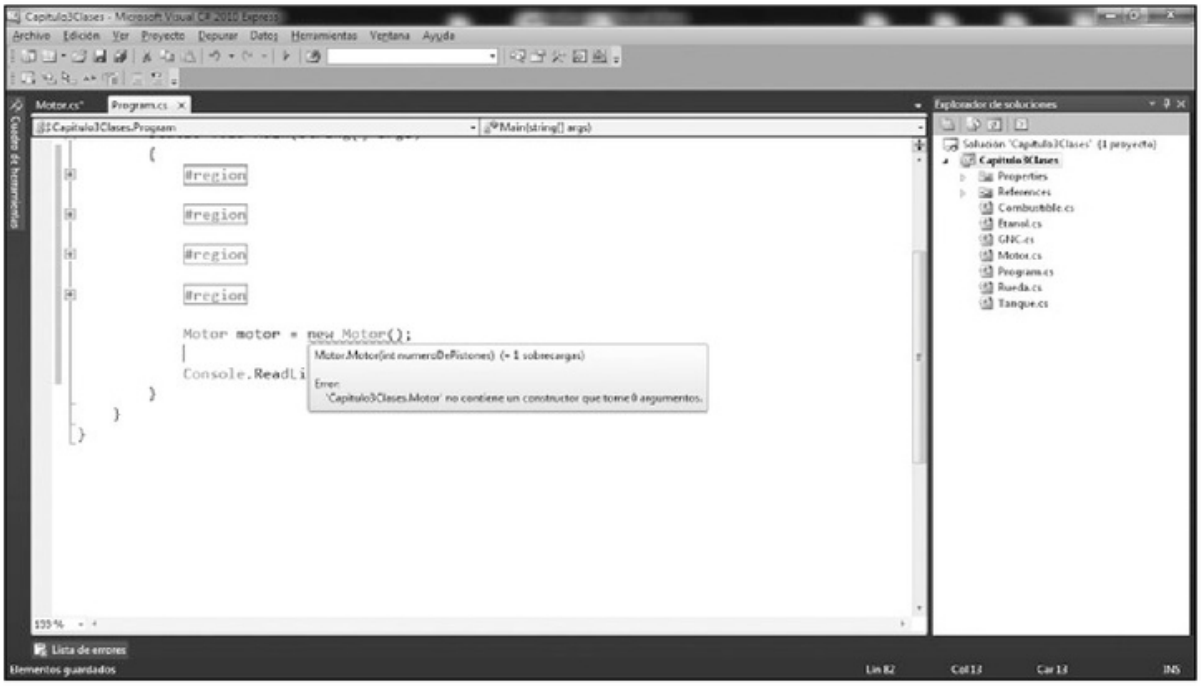


***Figura 20.** Al haber eliminado el constructor por defecto, el objeto ya no puede ser creado mediante un constructor que no sea alguno de los disponibles en el código de la clase.*

Existe un punto que es notorio en el manejo de estos constructores. Por cada constructor creado, el código es duplicado una y otra vez. Como en los ejemplos anteriores, la asignación de las variables pasadas por parámetros a las variables privadas de la clase se repite una y otra vez. Existe una forma de simplificar esto en el caso que solo necesitemos asignar los parámetros a variables internas de la clase.

```
public Motor()
    : this(4, 1200)
{ }

public Motor(int numeroDePistones)
    : this(numeroDePistones, 1200)
{ }

public Motor(int numeroDePistones, int Cc3)
```

```
{
    pistones = numeroDePistones;
    cc3 = Cc3;
}
```

Una vez creados los diferentes constructores, podemos hacer uso del atajo provisto por la sintaxis de C# para asignar los parámetros de entrada de un constructor a otro que tenga la capacidad de manejarlos. En el caso del constructor por defecto, sin parámetros, podemos predefinir los mismos parámetros por defecto que teníamos anteriormente, pero sin la igualación en el código, sino llamando al constructor que recibe dos parámetros y asigna el valor a las variables. En el segundo caso, al tener un único valor de entrada, pasamos este parámetro más un parámetro por defecto por parte de nuestro código. Por lo tanto, los dos constructores más simples, con menos parámetros, terminarán delegando la responsabilidad de asignar los valores iniciales al constructor más especializado. Para comprobar esto, veamos la siguiente modificación a la clase **Motor**.

```
public Motor()
    : this(4, 1200)
{
    ...
    Console.WriteLine("El motor se ha creado");
    Console.WriteLine("Constructor sin parámetros");
}

public Motor(int numeroDePistones)
    : this(numeroDePistones, 1200)
{
    Console.WriteLine("Constructor con un parámetro");
}
```

III CONSTRUCTOR POR DEFECTO

Cuando no declaramos ningún constructor en nuestras clases, el Microsoft.Net Framework se encargará de crear un constructor por defecto sin parámetros. Esto asegurará que el objeto puede ser construido aunque sea sin parámetros. Aunque agreguemos un constructor con un único parámetro, será necesario que creemos uno sin parámetros para tener un constructor por defecto.

```
public Motor(int numeroDePistones, int Cc3)
{
    Console.WriteLine("Constructor con dos parámetros");
    ...
}
```

Al ejecutar el código que crea un objeto de este tipo utilizando el constructor sin parámetros, vemos que son llamados tanto este constructor como el constructor especializado de dos parámetros, ejecutando el código en el mismo orden que es escrito. Primero, el **this** y luego, el contenido del constructor (**Figura 21**).

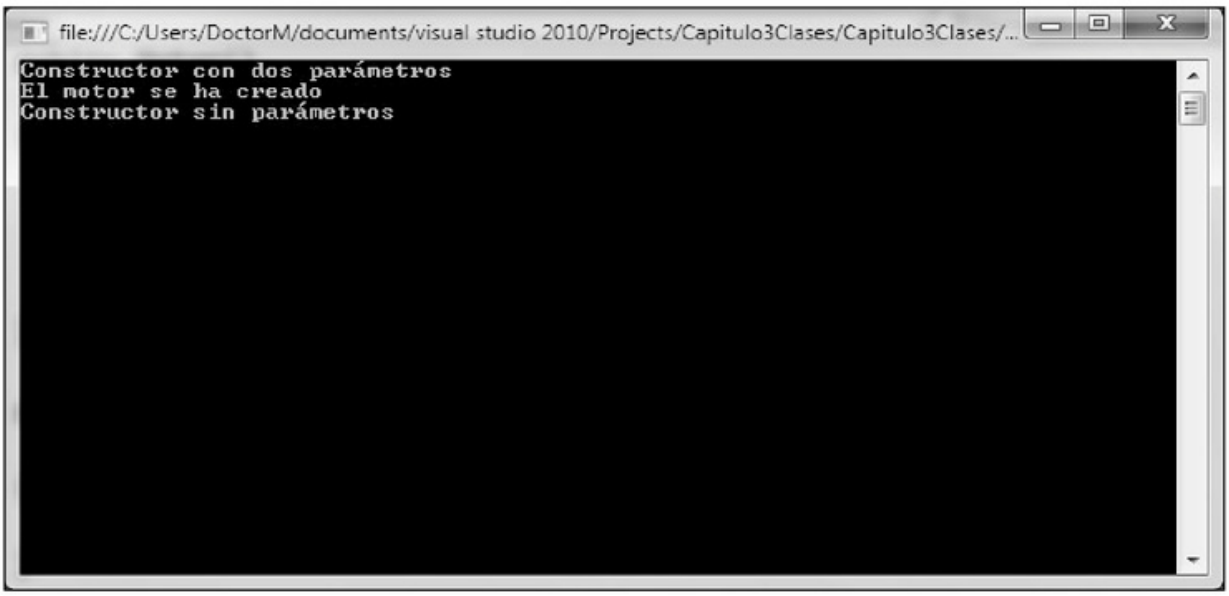


Figura 21. *Tanto el constructor por defecto como el constructor especializado con dos parámetros son ejecutados. El uso de this en el constructor pasará los parámetros al siguiente constructor que soporte estos valores.*

No debemos olvidar que los constructores, si bien son ejecutados en el momento de la creación del objeto, siguen siendo funciones, por lo que los diferentes modificadores de acceso son aplicables a estos. Por ejemplo, si marcáramos un constructor

III CONSTRUCTOR PRIVADO

Existen conjuntos de líneas de código preestablecidas para realizar diferentes funcionalidades. Estos patrones reciben el nombre de patrones de diseño. Uno de estos, llamado **Singleton**, evita que la clase pueda ser instanciada por otro código diferente de ella misma marcando como privado el constructor de la clase.

como privado mediante el uso de **private**, no habría posibilidad de que el objeto fuera construido por código ajeno a este. Quedando sujeta, su construcción, a la misma clase. Esta técnica es usada junto con el patrón de diseño **Singleton**.

Destructores

Los destructores se ejecutan cuando el objeto es destruido. Por lo tanto, cuando el objeto es eliminado de la memoria, antes de que esto suceda la porción del código alojada en el destructor será ejecutada. Los destructores son útiles en el caso que necesitemos liberar otros recursos contenidos dentro del objeto y que no hubiesen sido liberados con anterioridad. Veamos cómo declarar un destructor.

```
~Motor()
{
    Console.WriteLine("Objeto destruido");
}
```

Dentro de la misma clase, el destructor es declarado mediante el uso del carácter ~ y el nombre de la clase. Como en el caso de los constructores, estas funciones tienen una nomenclatura especial, no retornando ningún valor, colocando el símbolo ~ por delante del nombre (el cual tiene que ser idéntico al nombre de la clase) y no aceptando, en este caso, ningún parámetro.
Para provocar la ejecución del destructor, debemos esperar hasta que el objeto sea destruido; es el motor de ejecución el que se encargará de destruirlo una vez que el objeto no esté más en uso. Una forma de marcar el objeto para su destrucción es rompiendo la referencia que pudiera tener hacia la información que contiene.
Así como cuando hablábamos de punteros y el valor **null**, en este caso, podríamos forzar la destrucción del objeto asignándole a este el valor de referencia **null**.

```
Motor motor = new Motor();
...
...
//Quitamos la referencia del objeto
motor = null;
```

A pesar de esto, no lograremos ver realmente la destrucción del objeto, ya que es el motor el encargado de decidir cuándo eliminarlo en forma completa. De cualquier manera, para el ejemplo, es posible intentar forzar su ejecución mediante un pedido explícito de recolección y limpieza de variables no usadas.

```
Motor motor = new Motor();
...
...
//Quitamos la referencia del objeto
motor = null;
//Pedimos que se limpien
//los objetos marcados para eliminación
GC.Collect();
```

El resultado de esta acción hará que el objeto se limpie por completo de la memoria y que el destructor sea llamado, en nuestro caso, mediante, un mensaje que avise que el objeto fue finalmente destruido (**Figura 22**). Junto a los destructores, Microsoft .Net propone un patrón que incluye el uso de interfaces para la correcta liberación de los recursos de un objeto.

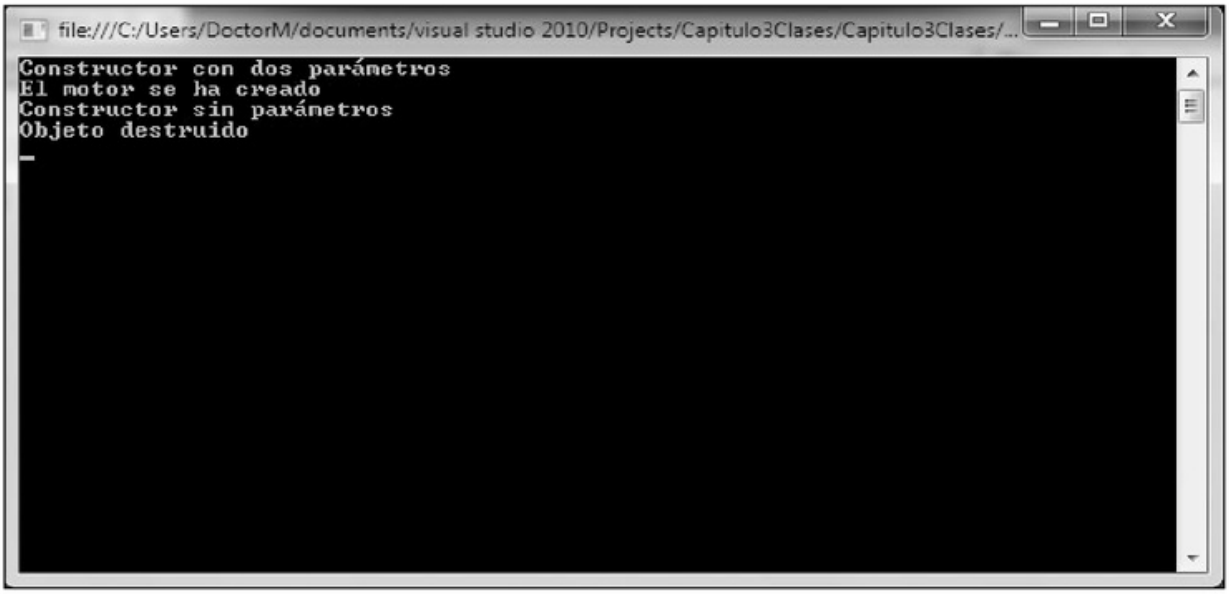


***Figura 22.** El mensaje colocado en el destructor es mostrado una vez que el objeto es efectivamente destruido y la memoria limpiada.*

III USO DE IDISPOSABLE

Cuando trabajamos con C# bajo el Microsoft .Net Framework, contamos con un patrón que ayuda a la liberación de recursos no manejados dentro de los diferentes objetos. Este patrón requiere de la implementación de la interfaz **IDisposable**, la cual, junto con los destructores, brinda funcionalidad para que el desarrollador fuerce la liberación de recursos dentro del objeto.

MÁS ALLÁ DE LAS FUNCIONES

Durante todo el capítulo hemos utilizado funciones como los puntos de entrada y ejecución de código para las clases. Esto puede resultar algo incómodo en especial cuando necesitamos enviar un único dato para ser almacenado por la clase o cuando requerimos configurarla de forma inicial. Por suerte, existen otros mecanismos para lograr este tipo de acciones y dejar a las funciones solo para la ejecución de funcionalidad específica. Además, entraremos en el campo de la lógica avanzada al implementar una cualidad de las funciones.

Propiedades

Las **propiedades** son otro mecanismo sintáctico propuesto por el lenguaje para la introducción y extracción de datos desde un objeto. En vez de utilizar funciones o exponer variables, podemos crear propiedades que permitan introducir estos valores, almacenarlos en variables internas de la clase y, luego, exponerlos o devolverlos según sea necesario. El siguiente código muestra la sintaxis para la declaración de una propiedad, aunque no incluye el cuerpo de la misma.

```
public [Tipo de dato] NombreDePropiedad
{
     get
     {
            ...
            ...
     }
     set
     {
            ...
            ...
     }
}
```

Una propiedad consta de un modificador de acceso, el tipo de dato que la propiedad manejará, tanto para entrada como salida de datos, el nombre de la propiedad y una estructura interna formada por dos partes: la sección **get** (obtener), utilizada para retornar un valor, y la sección **set** (configurar), que servirá de punto de entrada del valor desde fuera del objeto. Las propiedades no son ajenas a la clase y deben ser declaradas dentro del cuerpo de la misma. Además, se suele utilizar una variable privada auciliar para almacenar el valor de entrada y salida de la misma.

```
class Usuario
{
    private string nombre;

    public string Nombre
    {
        get
        {
            return nombre;
        }
        set
        {
            nombre = value;
        }
    }
}
```

Como podemos ver en el código, hemos declarado una propiedad llamada **Nombre**, la cual recibe y devuelve un valor de tipo **string**. Notaremos que la porción **get** retorna el valor mediante el uso de la palabra reservada **return** (devolver), la misma que hace uso de la variable **nombre** declarada en la parte superior de la clase. En la sección **get**, aparece una nueva palabra reservada, **value** (valor). Esta palabra reservada contiene, para la sección **set**, el valor enviado hacia la propiedad desde fuera del objeto. Ambas secciones, **set** y **get**, no sirven solo para almacenar o retornar un valor; en estas secciones podríamos escribir funcionalidad como, por ejemplo, para validar el valor ingresado, o mostrar un mensaje cuando el código es ejecutado. Por otro lado, el uso de propiedades posee una facultad adicional. De cualquier manera, en relación a la mayoría de los casos, las propiedades solo se usarán como medio de comunicación.
Sería válido que pensemos en que no es necesario realizar tantas líneas de código ya que podríamos lograr lo mismo simplemente colocando el modificador de acceso a **public** de una variable dentro de la clase; pero, si bien este planteamiento puede

III SIMPLIFICACIÓN DE PROPIEDADES

A medida que C# y Microsoft .Net han ido avanzando en el tiempo, desde sus primeras versiones hasta la actual la forma de escribir las propiedades se ha simplificado. En la actualidad, no necesitamos escribir todo el código interno de los bloques **get** y **set** ya que el compilador se encargará del manejo de la información que sea recibida y entregada.

resultar lógico, acarrea un problema. La exposición de variables dentro de una clase puede hacer que esta pierda coherencia mientras es ejecutada.
Imaginemos que estamos realizando una operación matemática, y el resultado es almacenado en una variable dentro de su clase con acceso público. Nuestro código realiza la operación y almacena el resultado, y, en algún punto del programa, desde fuera de nuestra clase, esta variable es modificada sin que nosotros sepamos que esto ha sucedido, lo que causaría un mal funcionamineto en la lógica interna de la clase. Luego, con el flujo del código dentro de nuestra clase, volvemos a necesitar de lo almacenado en la variable que contiene el resultado, pero como los datos fueron modificados por un agente externo a nuestro código, estos ya no son fiables ya que cualquier operación que pudiéramos hacer con esos datos podría causar anomalías en nuestro código, ya que no ha sido nuestro código el que realizó dichas modificaciones.
Por el contrario, las propiedades nos brindan un paso más allá en el concepto de encapsulación; pueden permitir que nuestras variables sean privadas, y solo dar acceso a aquellos elementos que realmente puedan ser consumidos o modificados por los desarrolladores que trabajen con nuestros objetos. Incluso, podríamos adicionar código condicional para su uso, tanto para consumir la información como para asignar nuevos valores, alertando al desarrollador si los valores introducidos no son válidos.

```
public string Nombre
{
    get
    {
     ...
    }
    set
    {
        nombre = value;
        Console.WriteLine("Variable asignada");
    }
}

...
...
//Fuera de la clase que contiene la propiedad
Usuario usuario = new Usuario();
usuario.Nombre = "Pedro";

//Mostramos el contenido de la propiedad
Console.WriteLine("Nombre de usuario: " + usuario.Nombre);
```

En el código anterior, podemos notar también que la forma en la cual se asignan y consumen los valores de una propiedad difieren de los de una función, pareciéndose más a la asignación de variables. Así, después de crear el objeto, asignamos a la propiedad **Nombre** un valor; este será capturado por la sección **set**, almacenado en la variable privada de la clase y, luego, al escribirlo en pantalla, será devuelto por la sección **get** (ver la figu**Figura 23**).

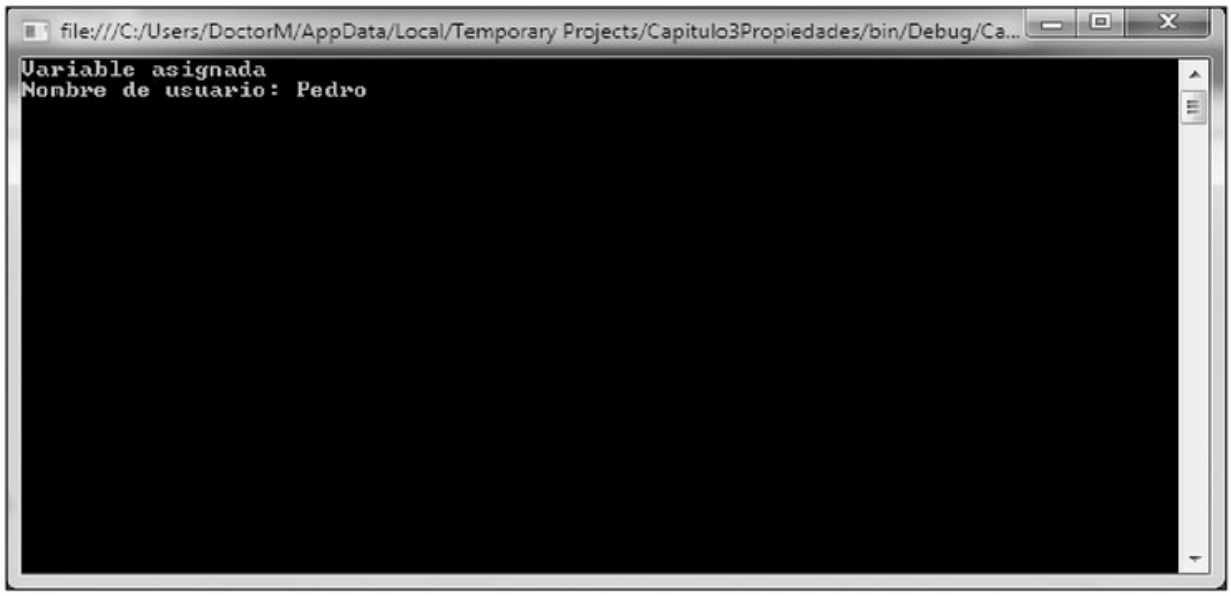


Figura 23. *Las propiedades son asignadas y leídas de forma similar a las variables. A diferencia de estas, pueden contener código de validación para prevenir modificaciones inesperadas de los datos.*

Las clases de propiedades que hemos visto hasta el momento son consideradas de lectura y escritura. Esto quiere decir que podemos tanto leer valores de ellas como asignarles valores. Pero es posible modificar las propiedades para hacerlas de solo lectura o de solo escritura. Esto nos da más versatilidad.

```
private Guid identificador;

public Usuario()
{
    identificador = Guid.NewGuid();
}

public Guid Identificador
{
    get
```

```
    {
        return identificador;
    }
}
```

Si vemos este código, veremos que al carecer de la sección **set**, únicamente podremos acceder a este como solo lectura. Esto quiere decir que no podremos asignarle valores, sino que simplemente podremos leer los datos que la clase nos proponga (**Figura 24**).

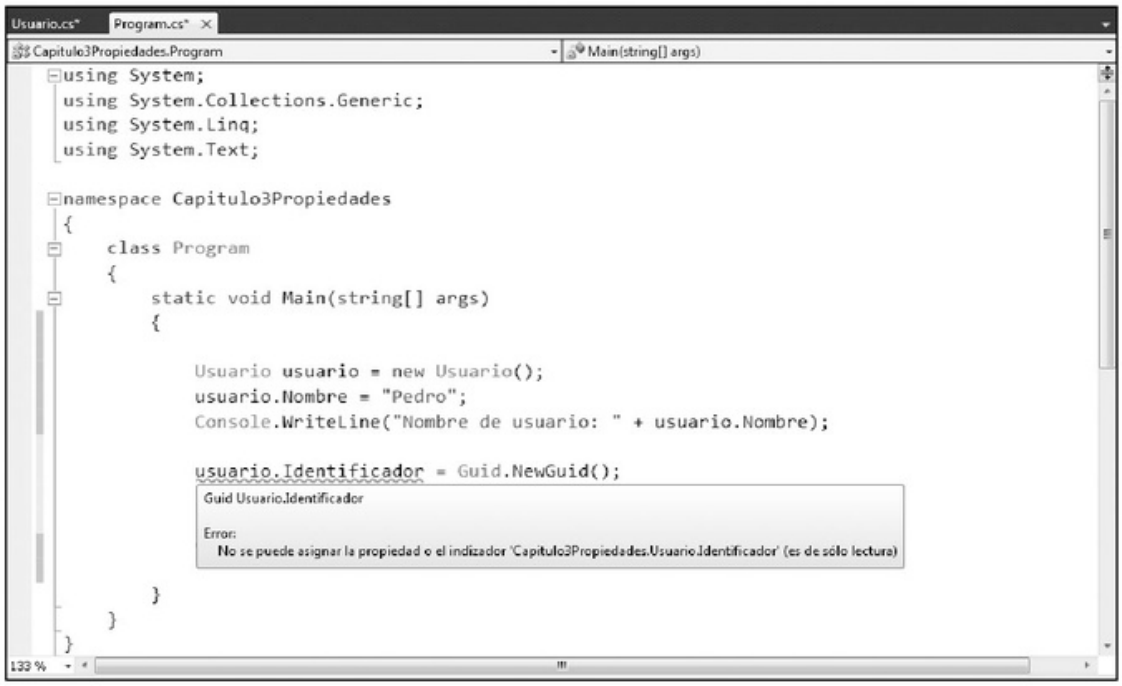


***Figura 24.** Al intentar asignar un nuevo valor a la propiedad Identificador de la clase, esta arroja un error de sintaxis ya que está declarada como de solo lectura.*

En este código, además, hacemos uso de un constructor de clase para asignar el valor inicial de la variable que luego será retornada por la propiedad. Por otra parte, al intentar leer los datos, estos serán accedidos sin problemas (**Figura 25**).

III HERENCIA

Un comportamiento curioso en el Microsoft .Net Framework en relación con la herencia de clases es el de que todos los tipos, sean propios de Microsoft .Net Framework como creados por nosotros, en forma automática heredarán del tipo **Object**. Por lo que notaremos que, al escribir nuestro código, propiedades como **ToString**, **Equals**, y otras siempre estarán presentes.

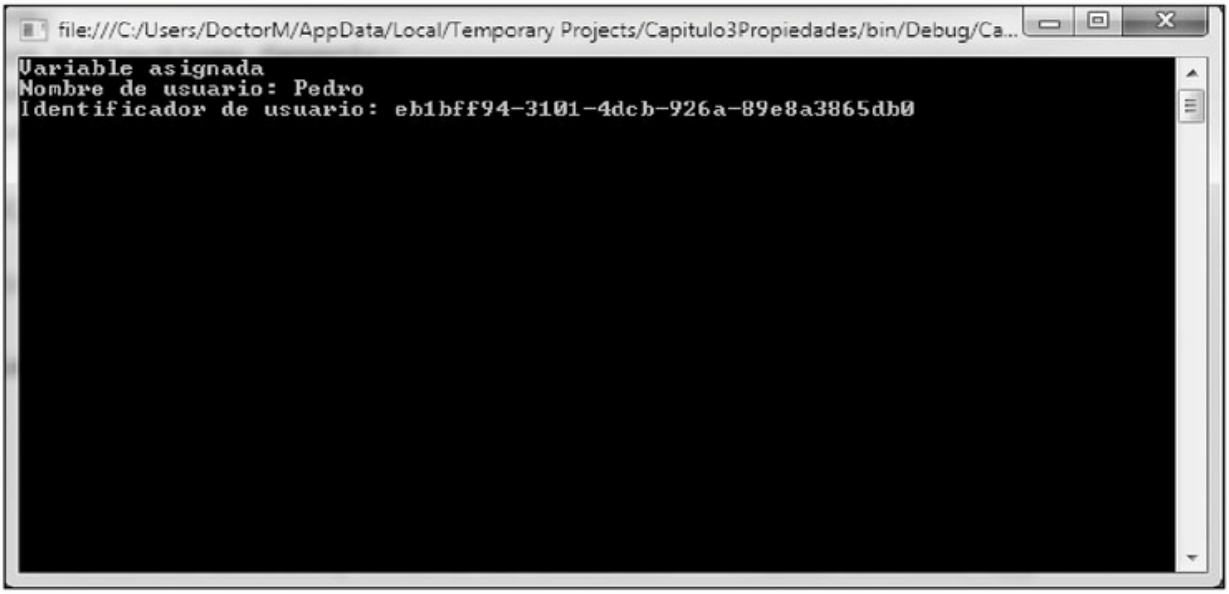


Figura 25. *La propiedad de solo lectura es accedida sin problemas cuando se intenta recolectar la información contenida en ella.*

Así como podemos hacer una propiedad como de solo lectura, es claro que podemos invertir el código para hacerla de solo escritura. Para esto, deberemos remover la sección **get** y dejar presente la sección **set** de la propiedad (**Figura 26**).
El uso de modificadores de acceso lograría un efecto similar.

```
private int elementos;

public int Elementos
{
    set
    {
        elementos = value;
    }
}
```

III MODIFICADORES DE ACCESO

Los modificadores de acceso **private**, **public**, **protected** o **internal** también pueden ser utilizados en las propiedades de una clase. Estos modificadores no solo pueden aplicar a nivel de toda la propiedad si no que pueden ser aplicados directamente sobre los bloques get y set modificando el acceso de cada bloque por separado.

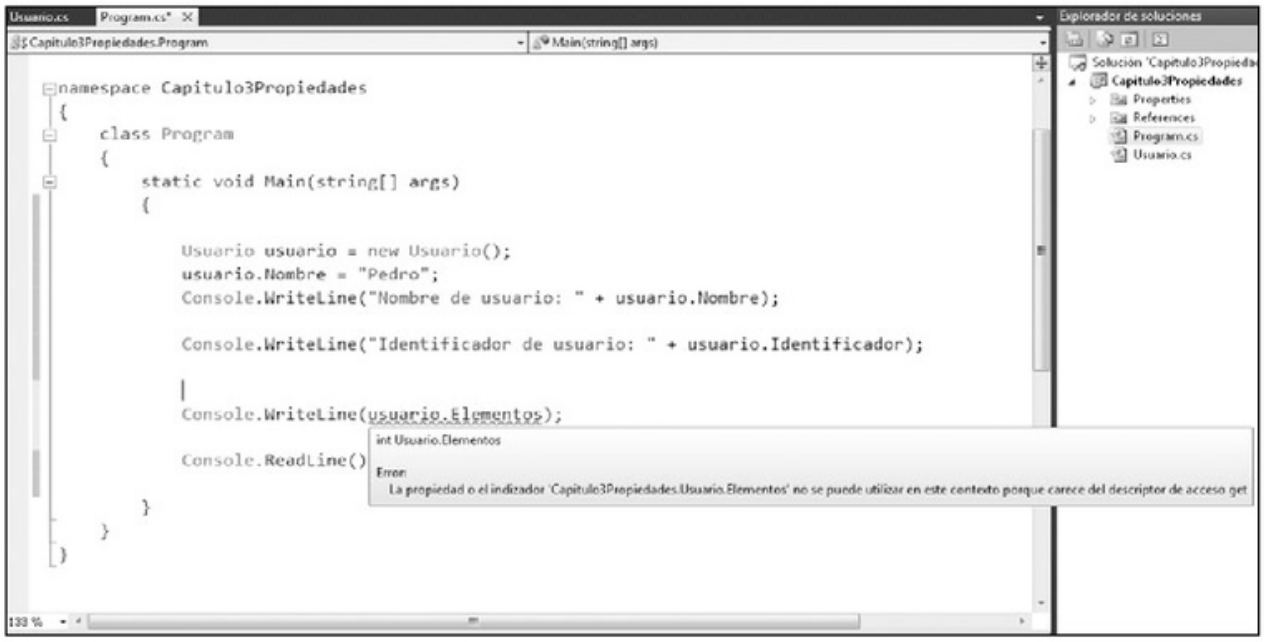


Figura 26. *La propiedad es ahora de solo escritura, y sus datos no pueden ser leídos. Para interactuar con ella, solo podremos asignarle información.*

Los modificadores de acceso también funcionan en las secciones de las propiedades. Es posible tener ambas secciones, ya sea **get** o **set**, pero con modificadores de acceso diferentes para cada uno; se podrá marcar una propiedad como de solo lectura o escritura, pero para código que resida fuera de ella, y no para el código interno o código que herede de nuestra clase.

```
class Persona
{
    private string nombre;

    public string Nombre
    {
        get
        {
            return nombre;
        }
        protected set
        {
          nombre = value;
        }
    }
}
```

```
class Hombre : Persona
{
     public Hombre()
     {
          this.Nombre = “Nombre de la Persona”;
     }
}
```

En este caso, contamos con dos clases; en la primera, considerada como clase base, existe una propiedad en la que su sección **set** es declarada como **protected** (protegido). Recordemos que este modificador de acceso hace que la propiedad, atributo o función sea privada, excepto para aquellas clases que hereden de la clase que contiene el elemento marcado **protected**. Por lo tanto, el siguiente código fallará al intentar asignar un valor a la propiedad desde fuera de la clase que heredó la propiedad pero no desde la clase **Hombre**.

```
Hombre hombre = new Hombre();
hombre.Nombre = “Otro nombre”;
```

En la **Figura 27**, vemos cómo el compilador nos arroja un error de inaccesibilidad a la propiedad debido al modificador de acceso que hemos utilizado.

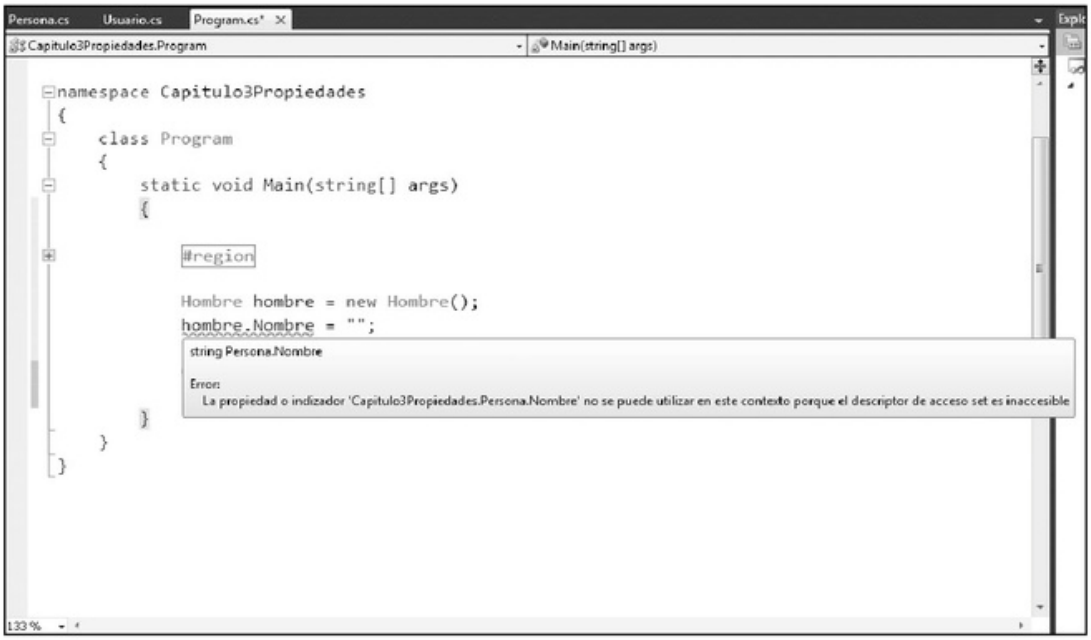


***Figura 27.** El modificador de acceso sobre la sección set de la propiedad hace que esta sección sea inaccesible desde fuera de la clase que ha heredado dicha propiedad.*

Al usar la última versión de C#, es posible simplificar la sintaxis que hemos estado utilizando para escribir las propiedades. El código se repite constantemente; más allá de la posibilidad de agregar cierta funcionalidad en las secciones **set** y **get**, siempre terminaremos retornando el contenido de una variable y asignando el valor de entrada a una variable dentro de la clase. Por lo tanto, si solo necesitamos realizar esta transacción rutinaria una y otra vez por cada propiedad en nuestras clases, podremos simplificar el código de la siguiente forma.

```
public int Valor
{
    get;
    set;
}
```

En este caso, la propiedad **Valor** será de lectura y escritura, y dejaremos que el compilador agregue los elementos faltantes a la hora de generar nuestro programa.

Recursividad

El concepto de **recursividad** se aplica a funciones que, para poder resolver un problema, se llamarán a sí mismas tantas veces como sea necesario. Este tipo de funciones son extremadamente útiles para recorrer árboles lógicos, donde una categoría contendrá una o muchas subcategorías, y cada subcategoría podría contener, a su vez, una o más subcategorías. Veamos un ejemplo de una función recursiva.

```
public int ContaryMostrar(int acumulador, int iteraciones)
{
    if (iteraciones > 0)
        acumulador += ContaryMostrar(acumulador, iteraciones - 1);

    return acumulador;
}
```

La función **ContaryMostrar** recibe dos parámetros, el primero es un acumulador. El segundo parámetro representa la cantidad de iteraciones. Toda función recursiva requiere de un punto de escape, esto quiere decir que necesita alguna condición por la cual la función debería dejar de llamarse a sí misma. Luego, para acumular el valor, simplemente se llamará, otra vez, a la misma función pasándole el siguiente valor por iterar y la cantidad de iteraciones que deberá realizar. Cuando la variable que contiene

las iteraciones llegue a cero, dejará de llamarse la función a sí misma, y los valores acumulados serán retornados uno por uno (**Figura 28**).

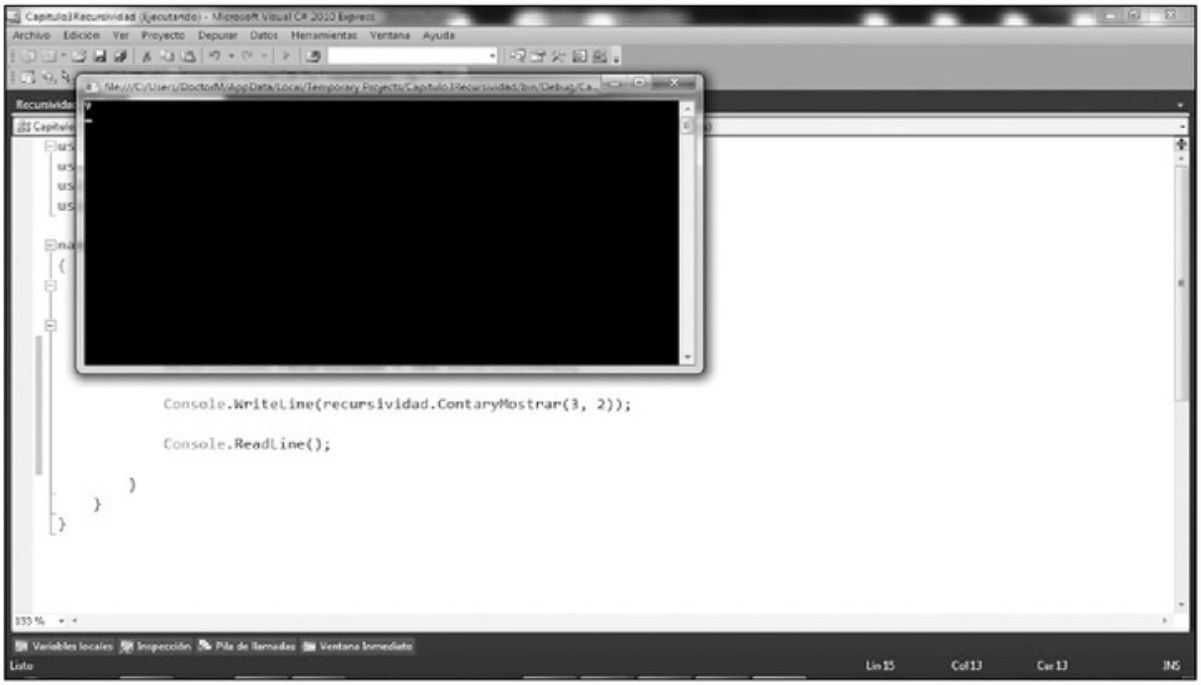


Figura 28. *La función recursiva es ejecutada tantas veces como se indica en uno de sus parámetros para retornar, por último, la suma total de cada una de las iteraciones.*

El siguiente código pretende invertir el orden de los caracteres. Este código recordará algunas de las propiedades del tipo **string**, como la posibilidad de tomar un carácter específico dentro de la cadena, así como conocer su longitud. En este caso, haciendo uso de estas dos características, se llama a la función **Invertir** tantas veces como caracteres tenga la cadena de texto, tomando como carácter por mostrar el último elemento de la lista menos una posición. Esto hará que la siguiente iteración contenga un carácter menos que la anterior iteración, hasta llegar al último carácter de ella. Al hacerlo, como si fuese un descenso en espiral, las funciones devolverán sus valores desde la última función a la más cercana, invirtiendo el texto enviado originalmente (**Figura 29**).

```
public void Invertir(string texto)
{
    if (texto.Length > 0)
    {
        Invertir(texto.Substring(1, texto.Length - 1));
        Console.Write(texto[0]);
    }
}
```

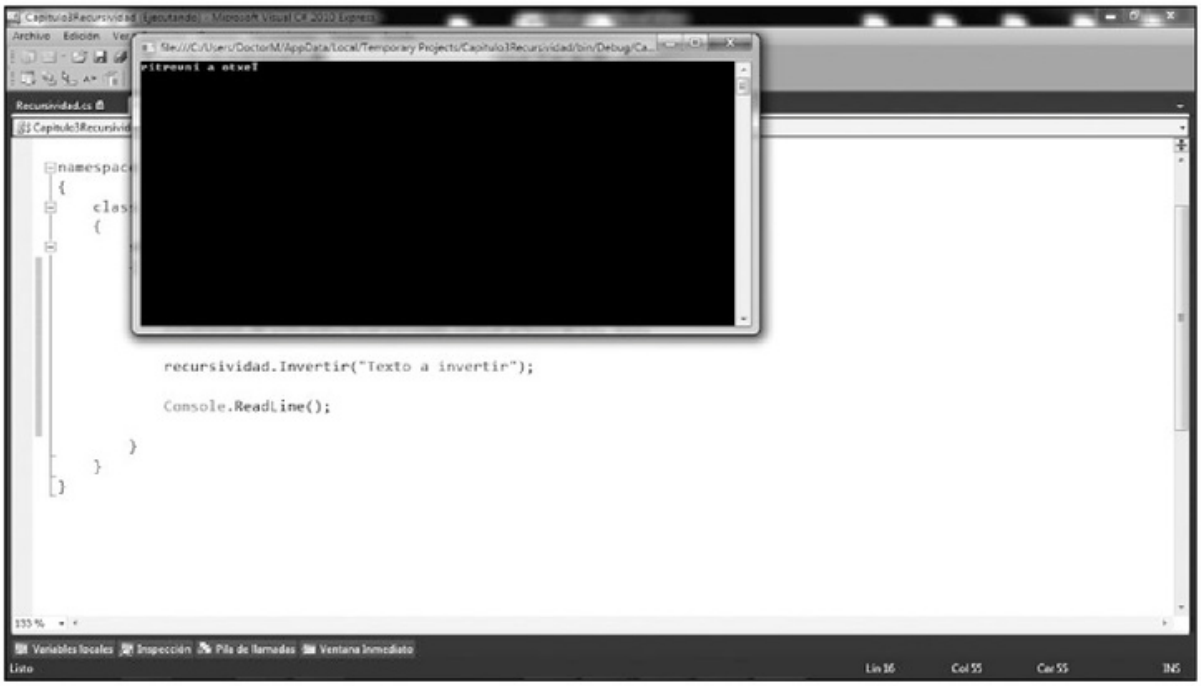


Figura 29. *Se invierte una cadena de texto utilizando una función recursiva. Como si fuese una espiral, la última función llamada será la primera en mostrar el resultado.*

A pesar de que la recursividad es útil, debemos tener cuidado al manejarla ya que puede agotar, muy rápido, los recursos del sistema al tener que mantener una gran lista de llamadas en memoria, como podemos ver en la **Figura 30**. Por otro lado, corregir errores en funciones recursivas suele ser difícil y costoso, por eso, solo deberíamos utilizarlas para casos muy específicos.

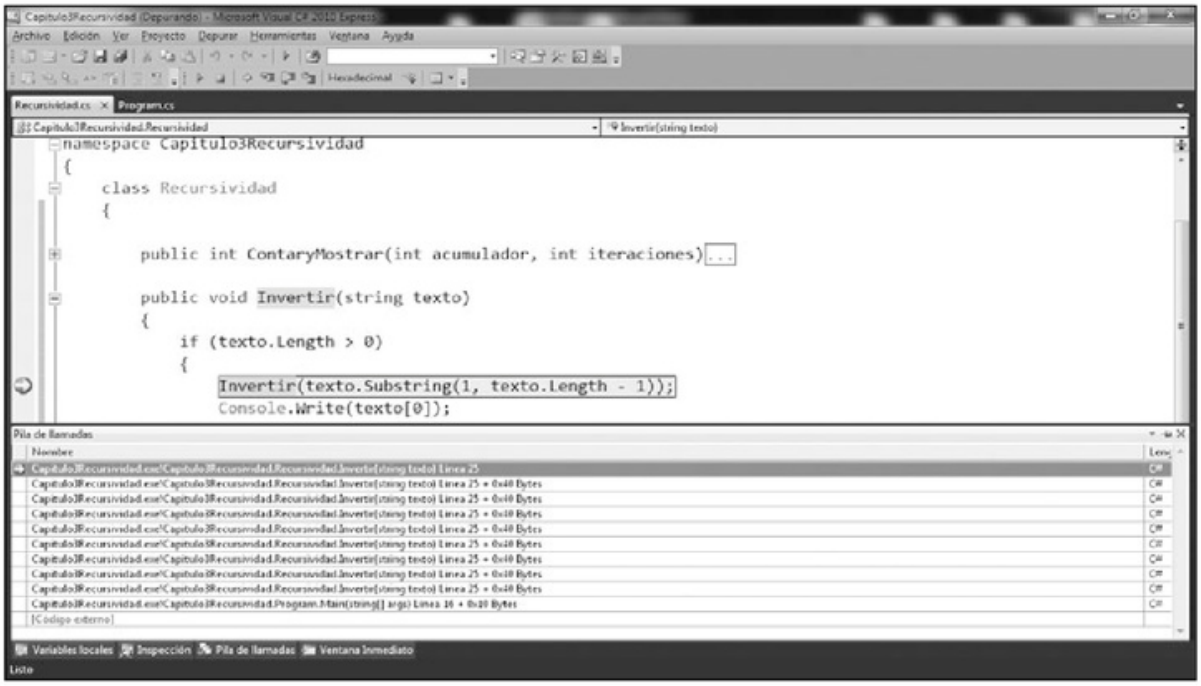


Figura 30. *Podemos ver la lista de llamadas después de algunas iteraciones con la función recursiva. Su mal uso puede causar problemas de funcionamiento.*

Enumeraciones

Es común, en el desarrollo de software, que necesitemos manejar diferentes estados para realizar determinadas acciones. Cuando hablábamos de la estructura **switch** en el **Capítulo 2**, vimos que esta era una buena alternativa para ejecutar código sobre la base de estas condiciones.

De cualquier manera, estas condiciones carecían de algo. Intentemos imaginar por un momento que nuestro código está preparado para manejar solo cuatro estados esperables; de esta forma, podríamos hacer uso de un **switch** para verificar una variable y actuar basándonos en su contenido.

Si usásemos una variable entera, esta en realidad sería capaz de almacenar más de cuatro números, por lo que ya tendríamos un problema si contuviera un valor que no fue considerado por nosotros. El siguiente problema radica en que, en algún momento, otro desarrollador de código podría estar usando nuestro código, pasando por parámetro el estado que necesita ejecutar nuestro **switch**. En este punto, el desarrollador al encontrarse con un parámetro de tipo entero, no podría saber, con certeza, cuál es el número correcto que debe pasar para que el código sea ejecutado sin errores.

```
private void EjecutarLogica(int valor)
{
    switch (valor)

    {
        case 1:
                    //Logica1
                    ...
                    break;
        case 2:
                    //Logica2
                    ...
                    break;
        case 3:
                    //Logica3
                    ...
                    break;
           case 4:
                    //Logica4
                    break;
    }
}
...
```

```
...
//Ejecutamos la función con un
//valor esperado
EjectuarLogica(1);
//Ejecutamos nuevamente con un
//valor no esperado
EjecutarLogica(5);
```

Como vemos en el código anterior, si en vez de enviar los valores dentro del rango esperado recibiéramos uno fuera de este, ninguna línea de código se ejecutaría, lo que podría traer consecuencias sobre nuestra aplicación, es decir, que el parámetro, su tipo, resulta ambiguo para aquel que usa nuestro código. Podríamos tratar de ser más explícitos usando, por ejemplo, un tipo **string**, donde el texto contenido en él representaría la funcionalidad por ejecutar, aunque llegaríamos al mismo punto, porque no tendríamos una forma de decirle al desarrollador cuáles son los nombres aceptados para la función. Es aquí donde entran en juego las **enumeraciones**. Una enumeración nos da la posibilidad de aunar un valor numérico a un texto descriptivo, y poder usar este último como identificador en los algoritmos de código.

```
enum AccionesLogicas
{
    Logica1,
    Logica2,
    Logica3,
    Logica4
}
```

Para declarar una enumeración, haremos uso de la palabra reservada **enum** más el nombre de la enumeración. En el cuerpo de esta, podremos colocar tantos nombres como creamos necesarios; cada uno de estos será uno de los posibles valores o estados que podrá contener una variable del tipo de esta enumeración.

```
AccionesLogicas enumeracion = new AccionesLogicas();
enumeracion = AccionesLogicas.Logica2;
```

Esta enumeración es usada para especificar el tipo de una variable. Al momento de asignar un valor, ya no usaremos un texto o un número, sino que podremos elegir desde la lista específica de valores de acuerdo con la enumeración utilizada (**Figura 31**).

Figura 31. *Utilizando la ayuda contextual, vemos cómo se despliega la lista de los posibles valores contenidos dentro de la enumeración.*

De esta forma, es posible modificar la función inicial para utilizar esta enumeración y conseguir que sea fácilmente comprendida por los desarrolladores, Además que restringe la posibilidad de cometer errores al pasar el valor esperado.

```
private void EjecutarLogica(AccionesLogicas enumeracion)
{
    switch (enumeracion)
    {
        case AccionesLogicas.Logica1:
                    ...
                    break;
        case AccionesLogicas.Logica2:
                    ...
                    break;
        case AccionesLogicas.Logica3:
                    ...
                    break;
        case AccionesLogicas.Logica4:
                    ...
                    break;
    }
}
...
```

```
...
//Ejecutamos la función con un
//valor esperado
EjectuarLogica(AccionesLogicas.Logica1);
//Ejecutamos nuevamente con otro
//valor esperado
EjecutarLogica(AccionesLogicas.Logica4);
```

Los nombres que utilizamos como elementos de las enumeraciones están relacionados con un valor numérico. Detrás de la cadena de caracteres, hay un valor numérico para esa cadena de texto. Entonces, el nombre desplegado es una forma de crear código más simple de leer además de lo ya expresado. Por lo tanto, es posible manipular este número. Si mostramos el contenido de la enumeración en la consola, en el primer caso aparece el texto que representa la opción seleccionada para esta enumeración. En el segundo caso, si transformamos el contenido de la enumeración a **int**, el valor numérico contenido en dicha enumeración es escrito en la consola, cambiando el nombre por el número que ésta representa.

```
AccionesLogicas enumeracion = new AccionesLogicas();
enumeracion = AccionesLogicas.Logica2;
Console.WriteLine("Nombre: " + enumeracion + ", Valor: " +
     (int)enumeracion);
```

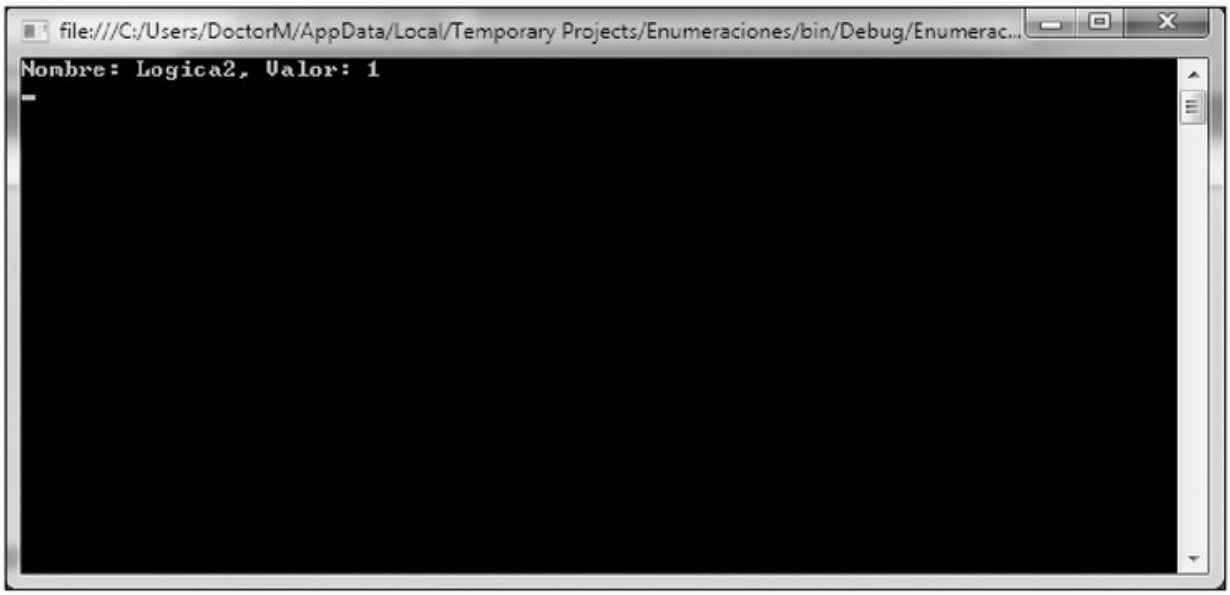


Figura 32. *Las enumeraciones contienen un valor numérico para cada uno de sus elementos. La cadena de texto solo debe ser considerada como un identificador.*

En este caso, al seleccionar el segundo elemento de la enumeración, este tendrá el valor de **1**; al igual que en los vectores, el índice inicial del primer elemento sería igual a **0**, y los siguientes mantendrán un orden ascendente.

```
enum AccionesLogicas
{
    Logica1 = 10,
    Logica2 = 45,
    Logica3 = 5,
    Logica4 = 6
}
```

Al ejecutar el mismo código, vemos que ahora el valor numérico del elemento seleccionado en la enumeración representa el número asignado en su declaración (**Figura 33**).

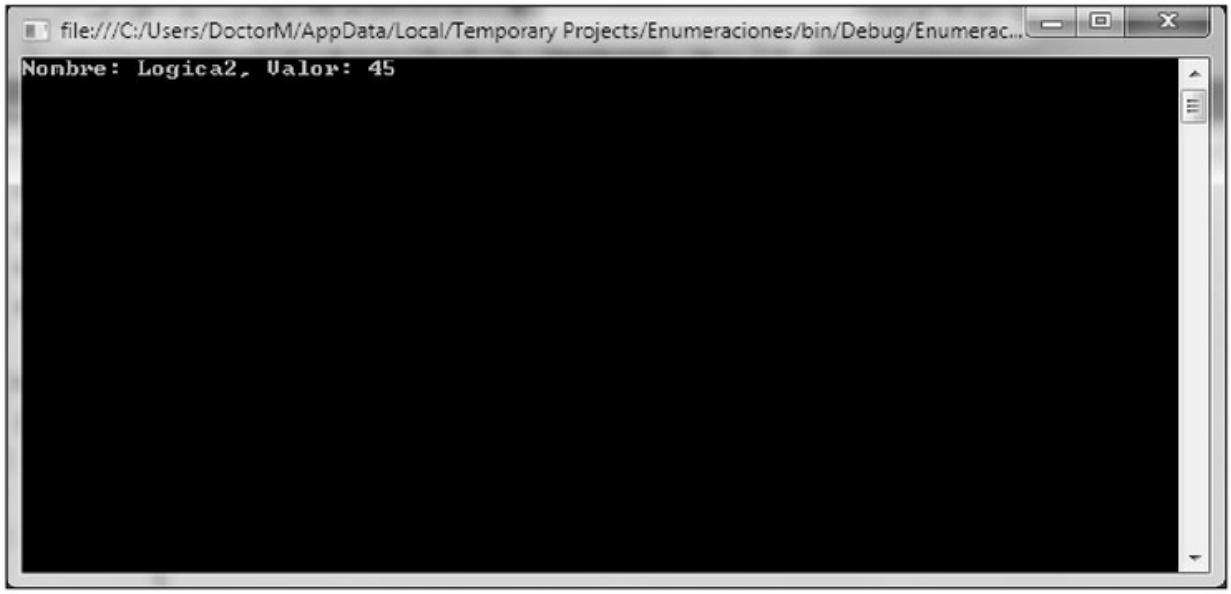


Figura 33. *El segundo elemento de la enumeración ya no sigue el orden por defecto, sino que respeta el valor asignado al momento de confeccionar la enumeración.*

Por lo tanto, será posible modificar el valor de la enumeración mediante este valor numérico asignado a cada uno de sus elementos. Como la cadena de texto es solo representativa, el número contenido en cada uno de estos valores es el importante. Usaremos este número para seleccionar el valor correspondiente.

```
enumeracion = (AccionesLogicas)6;
Console.WriteLine("Nombre: " + enumeracion + ", Valor: " + (int)enumeracion);
```

El valor numérico **6** hace referencia a la cadena de caracteres **Logica4**. Al cambiar el valor de la variable por este número, la enumeración pasará a representar, como valor seleccionado, el de **Logica4** (**Figura 34**).

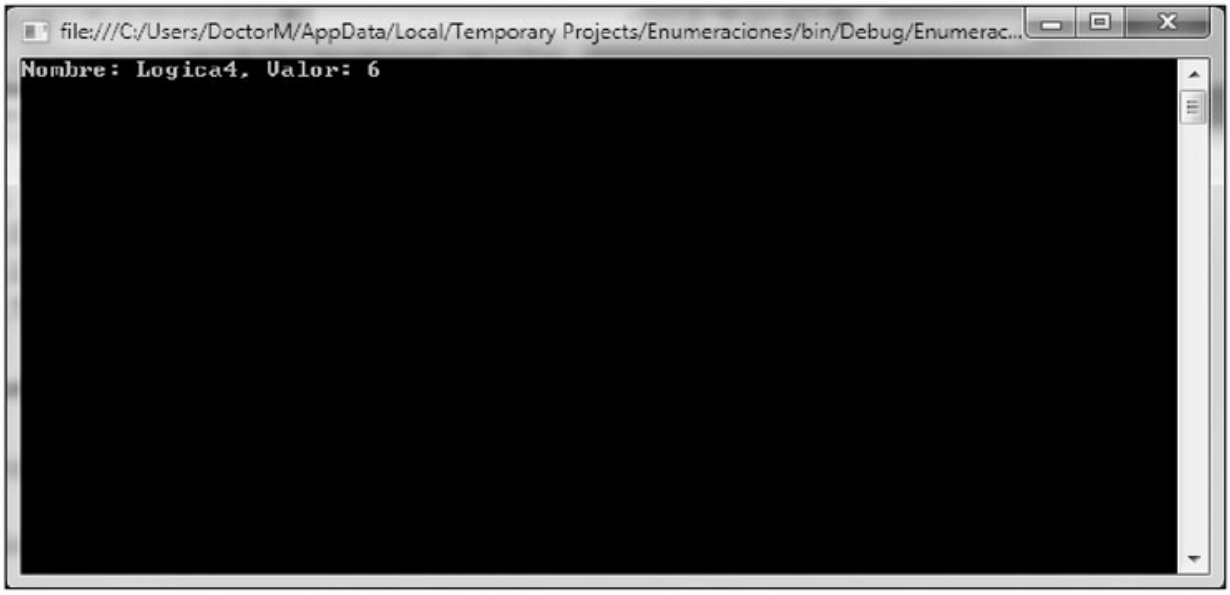


***Figura 34.** La variable que contiene la enumeración es modificada mediante el uso de un valor numérico y no por la lista de opciones disponibles dentro de la enumeración.*

MUCHO MÁS ALLÁ

Todo lo visto hasta el momento representa solo una parte de las posibilidades del lenguaje, del paradigma de programación orientado a objetos y del desarrollo de código en general. Los temas que siguen pretenden ampliar más aún los conceptos sobre estos tres ejes, y así garantizarnos que podamos tener conocimiento de todas las herramientas y atajos del desarrollo disponibles para que tomemos la mejor decisión a la hora de implementar nuestro propio código. Teniendo en cuenta que no existe la solución perfecta, será necesario conocer lo mejor posible cada uno para saber cuál usar.

Estructuras de datos

Son tipos de datos personalizados. Así como contamos con los tipos **int**, **long**, **string**, etcétera, una estructura representa un nuevo tipo definido por el desarrollador. Son especialmente útiles en casos donde los tipos básicos conocidos no brindan suficiente flexibilidad para el desarrollo, y es necesario, por lo tanto, contar con una forma de definir un tipo que sí lo haga. Para definir una, deberemos hacer uso de la palabra reservada **struct**, encapsulando el contenido de esta entre llaves.

```
struct EstructuraPersonalizada
{
     ...
     ...
}
```

Dentro de la estructura, colocaremos los diferentes tipos, variables, que representarán su cuerpo. Podríamos considerar la necesidad de una estructura en la representación de un punto de coordenada tridimensional, donde, dentro de los tipos conocidos no encontramos uno que pueda almacenar los ejes **X**, **Y** y **Z**.

```
struct Coordenadas
{
    long X;
    long Y;
    long Z;
}
```

Como vemos, la estructura **Coordenadas** contiene tres variables de tipo **long**, que serán usadas para almacenar cada uno de los puntos de una coordenada. Por lo tanto, **Coordenadas** pasará a ser un nuevo tipo conocido y se usará para declarar una variable de dicho tipo. Esto hace que las estructuras tengan un comportamiento especial ya que, así como los tipos básicos conocidos, en su gran mayoría, son considerados tipos por valor, las estructuras también serán consideradas tipos por valor, incluso si en su contenido existiese un tipo por referencia.
Todo lo declarado dentro de las estructuras es afectado por los modificadores de acceso, por lo que, para poder acceder a las variables contenidas en ellas, estas deberían ser declaradas como públicas mediante el uso del modificador de acceso **public**.

```
struct Coordenadas
{
    public long X;

    public long Y;
    public long Z;
}
...
...
```

```
Coordenadas estructura = new Coordenadas();
estructura.X = 10L;
estructura.Y = 5;
estructura.Z = 0;
```

Pero las estructuras no se limitan a contener otros tipos de datos ya que estas pueden además, incluir funcionalidad. Dentro de una estructura de datos, es posible crear nuevas funciones para la modificación de variables o la ejecución de código.

```
struct Coordenadas
{
    public void MostrarCoordenadas()
    {
        Console.WriteLine("X: " + X.ToString());
        Console.WriteLine("Y: " + Y.ToString());
        Console.WriteLine("Z: " + Z.ToString());
    }

    public long X { get; set; }

    public long Y { get; set; }


    public long Z { get; set; }
}
```

Como vemos en el código anterior, no solo hemos remplazado las variables públicas por propiedades, sino que además hemos creado funcionalidad dentro de la misma estructura para mostrar los valores de cada una de estas propiedades. El resultado, entonces, será idéntico al que conocemos hasta el momento (**Figura 35**).

III TRANSFORMAR DATOS

En C#, es posible transformar un tipo de dato a otro colocando el tipo de dato al cual se quiere transformar englobado entre paréntesis por delante del objeto o variable desde la cual se quiere obtener el valor por transformar. No siempre es posible lograr esto, ya que valores incompatibles con el tipo por transformar pueden causar un error. Esta acción es conocida como **cast**.

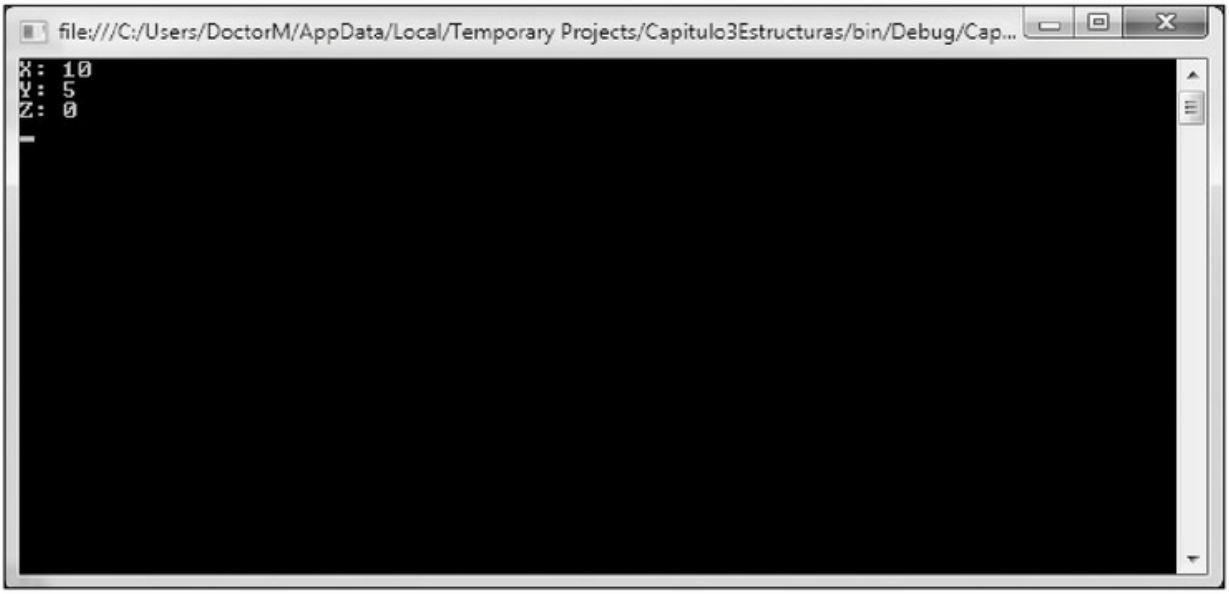


***Figura 35.** El comportamiento de una estructura es similar al de una clase. Podemos contener propiedades, atributos y funciones con código ejecutable en ellas.*

Las estructuras entonces, van más allá de la simple agrupación de tipos dentro de un único tipo de dato. Las estructuras son prácticamente iguales a una clase; la principal diferencia radica en dónde y cómo es almacenada dentro de la memoria. Por este motivo, en el momento que una estructura de datos comienza a crecer y a contener cada vez más elementos, tanto variables como funcionalidad, será conveniente transformarlas en una clase, simplemente cambiando el tipo de **struct** a **class**.

Clases abstractas

Las clases abstractas solo sirven para derivar otras clases. Pensemos en estas como la representación de las pautas que deben seguir las clases que hereden de ellas para la funcionalidad contenida. Veamos el ejemplo, a continuación.

```
abstract class ClaseAbstracta
{
    ...
}

class ClaseA : ClaseAbstracta
{
     ...
}
```

```
Class ClaseB : ClaseAbstracta
{
     ...
}
```

Para este caso, tanto **ClaseA** como **ClaseB** heredan de la clase abstracta. Al hacerlo, deberán implementar lo que, en la clase abstracta base, se dicte como obligatorio de implementar. Veamos el ejemplo, a continuación.

```
//Creamos la clase abstracta
public abstract class ClaseAbstracta
{
    public abstract int Valor1 { get; set; }

    public abstract int Valor2 { get; set; }

    public int Sumar()
    {
        return Valor1 + Valor2;
    }

}

//Implementamos una clase
//que hereda de la clase abstracta
public class ClaseA : ClaseAbstracta
{
    private int valor1;

    private int valor2;
```

III CUÁNDO USAR CLASS

A medida que entendemos el uso de las estructuras en C#, solemos usarlas de forma excesiva. Esto quiere decir que tendemos a escribir grandes cantidades de líneas dentro de estas, incluyendo funciones complejas. Cuando notemos que nuestras estructuras han dejado de ser un tipo de dato complejo, será conveniente transformarlas en clases mediante el uso de **class**.

```
        public override int Valor1
        {
            get
            {
                return valor1;
            }
            set
            {
                valor1 = value;
            }
        }

        public override int Valor2
        {
            get
            {
                return valor2;
            }
            set
            {
                valor2 = value;
            }
        }
    }
```

Al marcar la clase como abstracta, también podremos marcar elementos internos como abstractos. Estos elementos serán aquellos que todas las demás clases estén obligadas a implementar para completar su funcionalidad. Estos elementos se comportarán como contratos obligatorios de implementación, por parte de la clase que herede la clase abstracta lo que nos permitirá, entre otras cosas, poder llamar a estos elementos desde la clase base, aunque inicialmente no contenga código.

Ambas propiedades, **Valor1** y **Valor2**, son abstractas, por lo que en ellas, no existe funcionalidad alguna, y somos nosotros los que deberemos implementarla. La ventaja de esto es que podríamos tener diferentes clases que hereden del mismo tipo abstracto y, en cada una de ellas, se implemente una forma diferente de funcionalidad.

Por último, la función **sumar**, que no es abstracta, sí contiene código funcional, por lo que en este caso, podemos también aislar funcionalidad común dentro de la función **sumar**. Pensemos en código que no necesita ser replicado en cada clase que herede de nuestra clase abstracta y que con solo colocarlo en un único lugar puede ser utilizado tal cual es escrito por las demás clases (**Figura 36**).

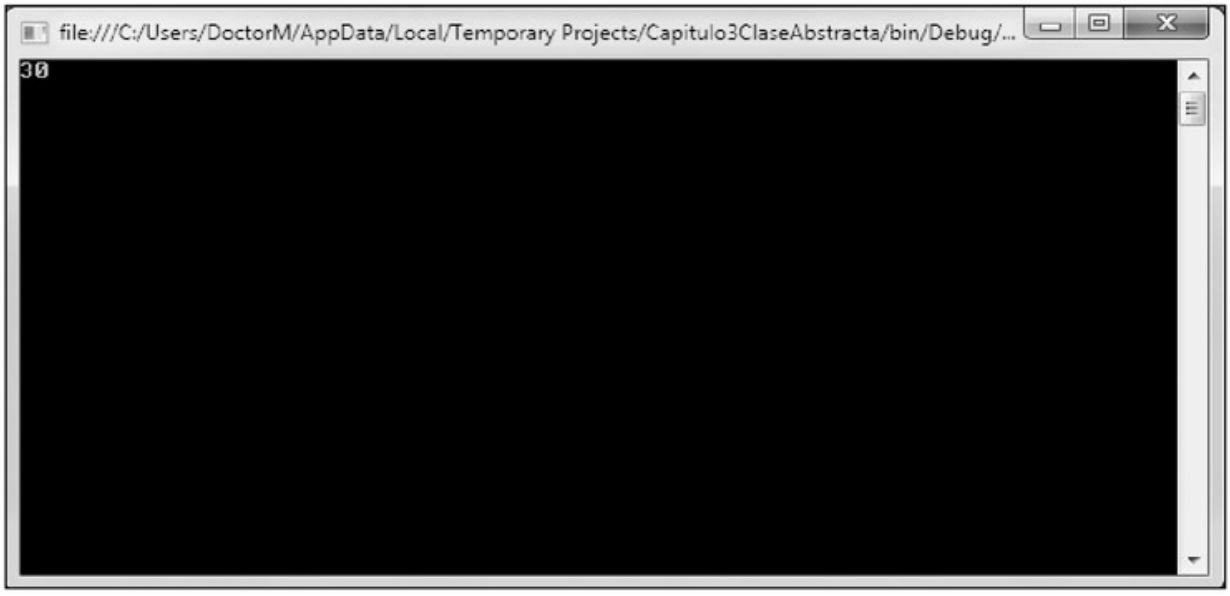


Figura 36. *Los valores de las propiedades son manejados por la clase no abstracta, mientras que el cálculo matemático es realizado por la clase abstracta.*

Otra característica de las clases abstractas es que no pueden ser instanciadas directamente. Esto quiere decir que no podemos crear un objeto de esta clase debido a que podría contener código no implementado, a diferencia del concepto de herencia que vimos al principio de este capítulo donde, a pesar de heredar de una clase su funcionalidad, la denominada clase base podía funcionar por sí sola como una instancia de esta. Esto nos otorga mayor control a la hora de crear código (**Figura 37**).

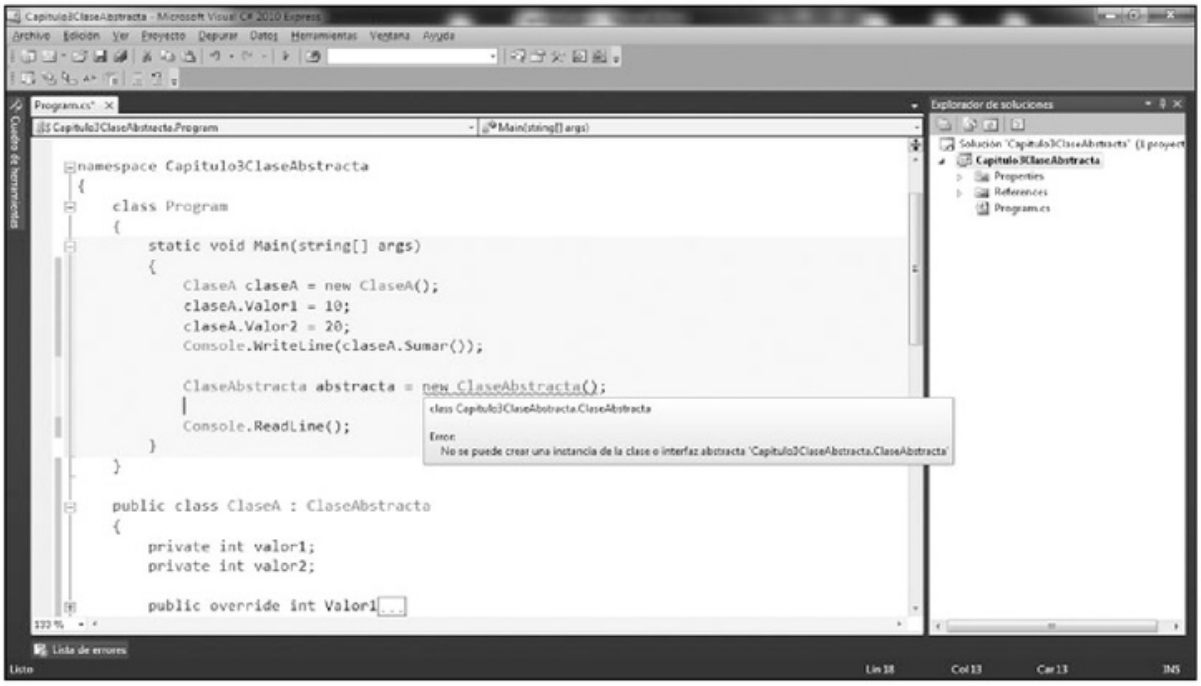


Figura 37. *Al intentar crear una nueva instanciade la clase abstracta, el compilador nos advierte que esto no es posible debido a que podría contener código no implementando.*

El uso de clases abstractas puede ser aplicado en donde necesitemos manipular diferentes instancias de objetos basados en una función que cree dichas instancias. Veamos el siguiente ejemplo donde unimos todos los postulados que hemos visto.

```
public class Operaciones
{
    public enum EnumOperaciones
    {
        Suma,
        Resta,
        Division,
        Multiplicacion
    }

    public OperacionesBase RetornarOperacion(EnumOperaciones operacion)
    {
     ...
     ...
    }
}
```

Una clase que tiene la capacidad de retornar diferentes operaciones, distintos cálculos matemáticos básicos sobre la base de una enumeración. Dentro de la función **RetornarOperación**, tendremos un **switch** que retorne una instancia de una operación. También podemos ver que esta función retorna una instancia de una clase base, que para este caso será abstracta.

```
public abstract class OperacionesBase
{
    public abstract long Calcular(long valor1, long valor2);
}
```

Esta clase abstracta solo contiene una función para realizar un cálculo, y recibe dos valores como parámetros, pero deja que las distintas clases hereden de esta la responsabilidad de implementar su funcionalidad.

```
public class Suma : OperacionesBase
{
```

```
    public override long Calcular(long valor1, long valor2)
    {
        return valor1 + valor2;
    }
}

public class Resta : OperacionesBase
{
    public override long Calcular(long valor1, long valor2)
    {
        return valor1 - valor2;
    }
}

public class Division : OperacionesBase
{
    public override long Calcular(long valor1, long valor2)
    {

        return valor1 / valor2;
    }
}

public class Multiplicacion : OperacionesBase
{
    public override long Calcular(long valor1, long valor2)
    {
        return valor1 * valor2;
    }
}
```

Por lo tanto, cada una de las clases tendrá la responsabilidad de realizar un cálculo matemático, usando todas el mismo nombre de función. Ahora que contamos la estructura completa de código, podemos implementar el contenido de la función que retornará la instancia de la operación por ejecutar.

```
public OperacionesBase RetornarOperacion(EnumOperaciones operacion)
{
    switch (operacion)
```

```
    {
        case EnumOperaciones.Suma:
            return new Suma();
            break;
        case EnumOperaciones.Resta:
            return new Resta();
            break;
        case EnumOperaciones.Division:
            return new Division();
            break;

        case EnumOperaciones.Multiplicacion:
            return new Multiplicacion();
            break;
        default:
            return new Suma();
            break;
    }
}
```

Como ya dijimos, no es posible crear una nueva instancia de una clase abstracta, pero sí es posible hacerlo por medio de una clase concreta. Aquí volvemos al concepto de polimorfismo donde, si bien se espera retornar un tipo **OperacionesBase**, se retorna una instancia de un objeto que hereda de ella. Al momento de llamar a esta función, el objeto que nos sea devuelto contendrá los mismos métodos, sea cual sea la operación que necesitemos ejecutar. Esto se debe a que todos estos objetos heredan e implementan la funcionalidad sobre la función abstracta **Calcular**. Por eso, sin importar cuál sea el objeto, es posible ejecutar esta función y que el objeto se encargue de determinar cuál será el cálculo que realizará.

```
Operaciones operaciones = new Operaciones();

//Obtenemos un objeto para dividir
OperacionesBase operacion =
    operaciones.RetornarOperacion(Operaciones.EnumOperaciones.Division);
Console.WriteLine("División: " + operacion.Calcular(10, 2));

 operacion =
```

```
operaciones.RetornarOperacion(Operaciones.EnumOperaciones.Multiplicacion);
Console.WriteLine("Multiplicación: " + operacion.Calcular(10, 2));
```

Para ambos casos, el tipo de la variable donde se almacenará el objeto instanciado es del tipo de la clase abstracta, por lo tanto, podrá almacenar cualquier objeto que haya heredado de esta. Al ejecutar la función **Calcular**, será el código de la función implementada dentro de cada objeto específico el que se ejecute, dividiendo dos números en el primer caso, y multiplicándolos, en el segundo (**Figura 38**).

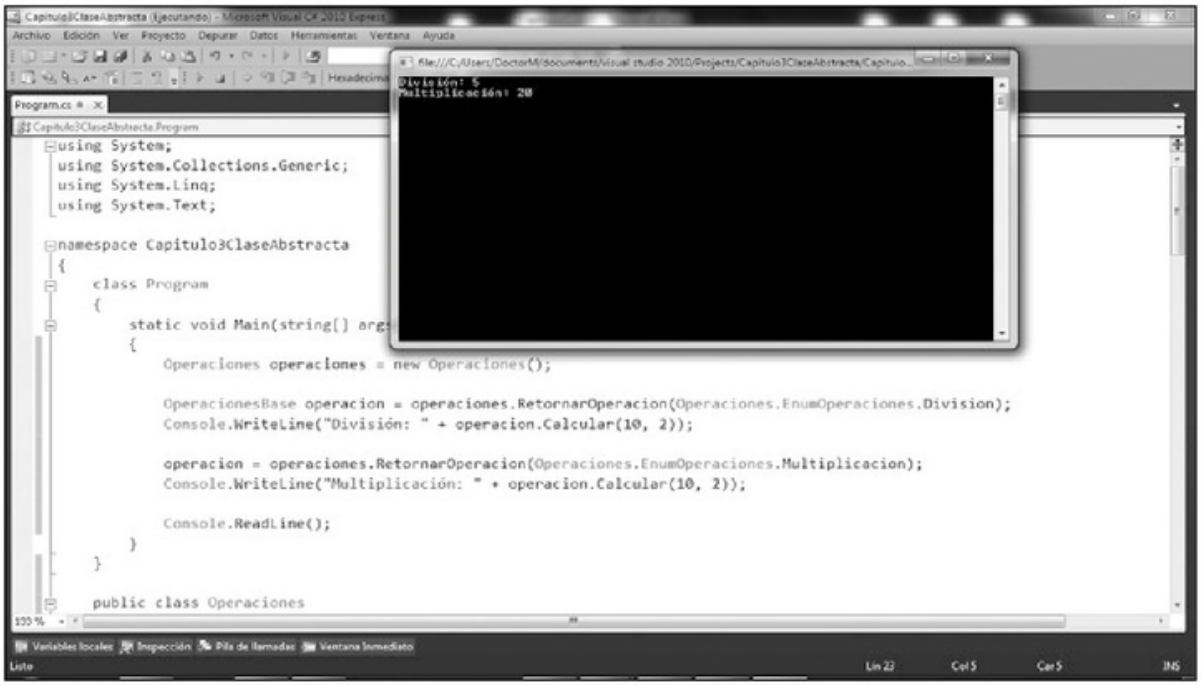


***Figura 38.** La misma variable, declarada del tipo base abstracto, puede contener cualquier objeto que herede de ella. La implementación de la operación es realizada en el código de la clase específica y no, en la clase abstracta.*

Interfaces

Las **interfaces** tienen una función similar a las clases abstractas, con una sutil diferencia, en estas no se puede crear código alguno, solo es posible declarar las propiedades y funciones que contendrán aquellas clases que implementen la interfaz. Debido a esto, las interfaces son utilizadas como contratos entre objetos a diferencia de las clases abstractas que pueden proveer cierta funcionalidad.

En este caso, cuando hablamos de contrato, nos referimos al hecho de que cierta porción de código o una aplicación externa podrían exigir la implementación de una interfaz determinada para poder aceptar un objeto. La implementación de una interfaz, por lo tanto, da una garantía de existencia de las funciones y propiedades declaradas en ella, como vemos en el código fuente a continuación.

```
public interface InterfazMatematica
{
    int Sumar(int valor1, int valor2);

    int Restar(int valor1, int valor2);
}
```

Para este caso, la interfaz **InterfazMatematica** declara dos funciones, una para **Sumar** y otra para **Restar**. Aquí, al igual que una función abstracta, no se puede escribir funcionalidad alguna, y agregar las líneas de funcionalidad es responsabilidad de la clase que implemente esta interfaz.

```
public class Calculos : InterfazMatematica
{
    public int Sumar(int valor1, int valor2)
    {
        return valor1 + valor2;
    }

    public int Restar(int valor1, int valor2)
    {
        return valor1 - valor2;
    }
}
```

La clase **Cálculos** implementa, entonces, la interfaz creada anteriormente y, por lo tanto, se hace cargo de la funcionalidad para cada una de las funciones.
Las interfaces son consideradas contratos ya que su implementación obliga a la clase a implementar sí o sí cada una de las funciones propuestas, dando esta garantía de existencia sobre el contrato de la interfaz (**Figura 39**).

NOMENCLATURA

Las interfaces suelen seguir una nomenclatura con relación a su nombre. La convención creada es que comiencen con la letra I mayúscula. Esta convención está tan difundida que todas las interfaces encontradas en el Microsoft. Net Framework comienzan con esta letra seguidas del nombre de la interfaz, así como el código comercial creado por terceros.

***Figura 39.** Al omitir una de las funciones del contrato propuesta por la interfaz, obtenemos un error al momento de intentar compilar la aplicación. Esto garantiza la validez del contrato ofrecido.*

Podríamos, por lo tanto, tener una segunda clase que implemente la interfaz creada, pero que su implementación difiera de la otra clase.

```
public class OtrosCalculos : InterfazMatematica
{
    public int Sumar(int valor1, int valor2)
    {
        return (valor1 + valor2) * 2;
    }

    public int Restar(int valor1, int valor2)
    {
```

IMPLEMENTACIÓN VS. HERENCIA

Tanto la implementación como la herencia de interfaces y clases se escriben de forma idéntica. Después del nombre de clase se utilizan los **:** para especificar el elemento del cual implementar o heredar. Pero, de las clases se hereda funcionalidad, mientras que desde las interfaces se debe implementar el contrato. Una forma de entender la diferencia entre estos dos modos.

```
            return (valor1 - valor2) / 2;
        }
    }
```

Junto a esto, aparece una nueva clase que toma la responsabilidad de ejecutar una de las funciones, pero independientemente de la clase que nosotros le enviemos.

```
public class Calculadora

{

    public void Calcular(InterfazMatematica objeto)
    {

        Console.WriteLine(objeto.Sumar(10, 20));
    }
}
```

El parámetro de la función **Calcular** es un objeto cualquiera, siempre y cuando respete el contrato de la interfaz **InterfazMatematica**. Mientras que lo haga, la ejecución del código interno de la función estará garantizada.
Por lo tanto, será posible ejecutar las funciones que traiga consigo o, por lo menos, aquellas que son parte del contrato con la interfaz.

```
Calculadora calculadora = new Calculadora();

InterfazMatematica objeto = new Calculos();

calculadora.Calcular(objeto);
```

Vemos cómo el tipo de nuestro objeto es de la interfaz, aunque al igual que las clases abstractas, las interfaces no pueden ser creadas como objetos directamente; sí es posible mediante la creación de un objeto que respete el contrato de la interfaz. Con este objeto creado, la clase **Calculadora** puede usarlo para realizar el cálculo (**Figura 40**).
En C#, así como en los diferentes lenguajes soportados por Microsoft .Net Framework, se permiten implementar, en una clase, tantas interfaces como uno quiera, cada una separada por una , (coma), a diferencia de la herencia de clases, donde solo es posible heredar de una sola clase en particular.

Figura 40. *La clase* ***Calculadora*** *utiliza el objeto con la firma de la interfaz para realizar el cálculo matemático, dejando la responsabilidad del cálculo en sí al código implementado por la clase particular.*

RESUMEN

Este capítulo nos ha servido para saltar desde la programación de hace unos 30 años, la programación estructurada vista en el Capítulo 2, al modelo de programación actual y moderno que incluye la orientación a objetos. Los conceptos de herencia, encapsulamiento y polimorfismo, y su comprensión resultan fundamentales para el pensamiento lógico requerido en la correcta escritura de código, y para evitar la redundancia tanto de líneas como de funcionalidad. Aunque en este capítulo nos hemos remitido a nombrar cada uno de ellos con ejemplos simples, en parte por el nivel de complejidad que presentan estos temas para aquel que está aprendiendo, en los capítulos siguientes usaremos con profundidad cada uno de estos conceptos, utilizados en ejemplos reales y completos.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuál es la diferencia entre los tipos por valor y los tipos por referencia?

2 ¿Por qué llamamos implementar al uso de interfaces en una clase?

3 ¿Es posible crear instancias nuevas de clases abstractas?

4 ¿Una función abstracta puede contener código?

5 Indique cuáles son las formas de manipular las enumeraciones.

6 Explique el concepto de recursividad.

7 ¿Qué pasaría si una función recursiva no poseyera una condición de escape?

8 ¿Cuántos destructores pueden ser escritos en una clase?

9 ¿Qué es el Garbage Collector?

10 ¿Existe algún límite de herencias por clase?

EJERCICIOS PRÁCTICOS

1 Haciendo uso del concepto de herencia, intente recrear la jerarquía de clases para crear un perro, partiendo desde la clase mamíferos.

2 Modifique el código presentado cuando hablábamos de recursividad, para invertir el orden de las palabras y no de todos los caracteres.

3 Cree un nuevo proyecto para la ejecución de cálculos matemáticos varios. Estos cálculos deberán ser ejecutados por lotes, por lo tanto, su código debería poder recibir un gran conjunto de objetos, y ejecutar cada uno de esos objetos y el cálculo matemático provisto por este.

4 Intente crear dos interfaces y, a varias clases, asígneles ambas. Luego, pruebe crear objetos de cada clase, pero utilizando como tipo de datos una u otra interfaz. Mediante la ayuda de contexto vea qué funciones y propiedades tiene disponibles. Analice el porqué.

5 Cree una clase con propiedades donde la sección set de cada propiedad solo sea accesible para clases que hereden de la clase que contiene estas propiedades.

Capítulo 4

Acceso a datos

Todo programa de computación está destinado a manipular datos, tanto numéricos, como textos, cálculos matemáticos, estadísticas, entre otros, por lo que el resguardo y posterior recuperación de estos resultará de vital importancia. En el presente capítulo, aprenderemos sobre estos sistemas de almacenamiento, las bases de datos, los archivos XML y los componentes provistos por Microsoft .Net para interactuar con ellos.

BASES DE DATOS

Con la creación de las aplicaciones para computadoras, desde el principio de la computación los programas han necesitado almacenar temporalmente sus datos en algún lugar que pudiera asegurar su persistencia.
Esto se ha debido a que los programas, así como las mismas computadoras, no permanecen en funcionamiento de manera perpetua, y pueden apagarse o reiniciarse, con el riesgo de perder toda la información con la cual hubiéramos estado trabajando en el momento de su ejecución.
Por lo tanto, ya sea en archivos creados especialmente para ser entendidos por el programa en ejecución como en los modelos más sofisticados con algoritmos de búsqueda optimizada, resulta indispensable contar con sistemas que permitan resguardar estos datos, para recuperarlos en cualquier momento.
En la actualidad, podemos encontrar cientos de modelos de bases de datos, pagos o gratuitos, relacionales, documentales u orientados a objetos, con soporte para replicación de datos o específicos para aplicaciones en tiempo real. En definitiva, la elección del modelo de base de datos dependerá directamente de nuestras necesidades, los planes de escalamiento de nuestras aplicaciones o la cantidad de datos que requieran manejar (**Figura 1**).

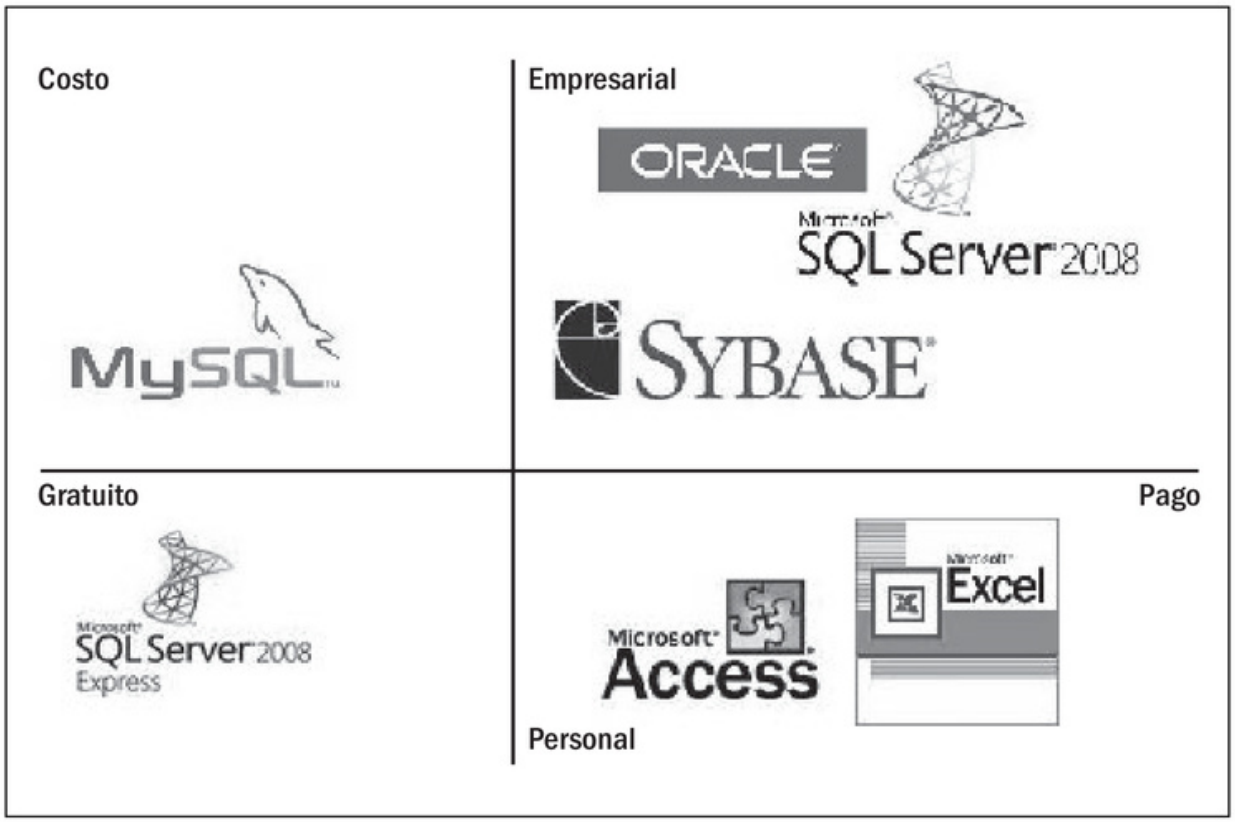


Figura 1. *Podemos agrupar los distintos motores de bases de datos por su costo, prestaciones, forma de manipulación, o almacenamiento de los datos o sistemas operativos soportados.*

Hoy por hoy, las aplicaciones empresariales, así como los desarrollos pequeños y medianos, concuerdan en la utilización de bases de datos relacionales. Estos tipos de bases de datos obtienen su nombre debido a las relaciones que pueden generarse entre sus datos. Para ejemplificar esto, pensemos en un modelo de materias de una universidad y los alumnos que asisten a ellas.

Por una parte, contaríamos con el conjunto de datos que represente a las materias disponibles y, por otro, a los alumnos de dicha universidad. Cada alumno, al asistir a una facultad, genera una relación entre él y las materias que cursa, dándonos la posibilidad de saber a cuales materias asiste este alumno del conjunto de materias disponibles y a su vez, saber cuántos y qué alumnos del conjunto de alumnos asisten a una materia en particular. En las bases de datos relacionales, ambos conjuntos de datos son almacenados en tablas, y las relaciones son realizadas a nivel de esta sobre la base de los campos que las distintas tablas pudieran contener (**Figura 2**).

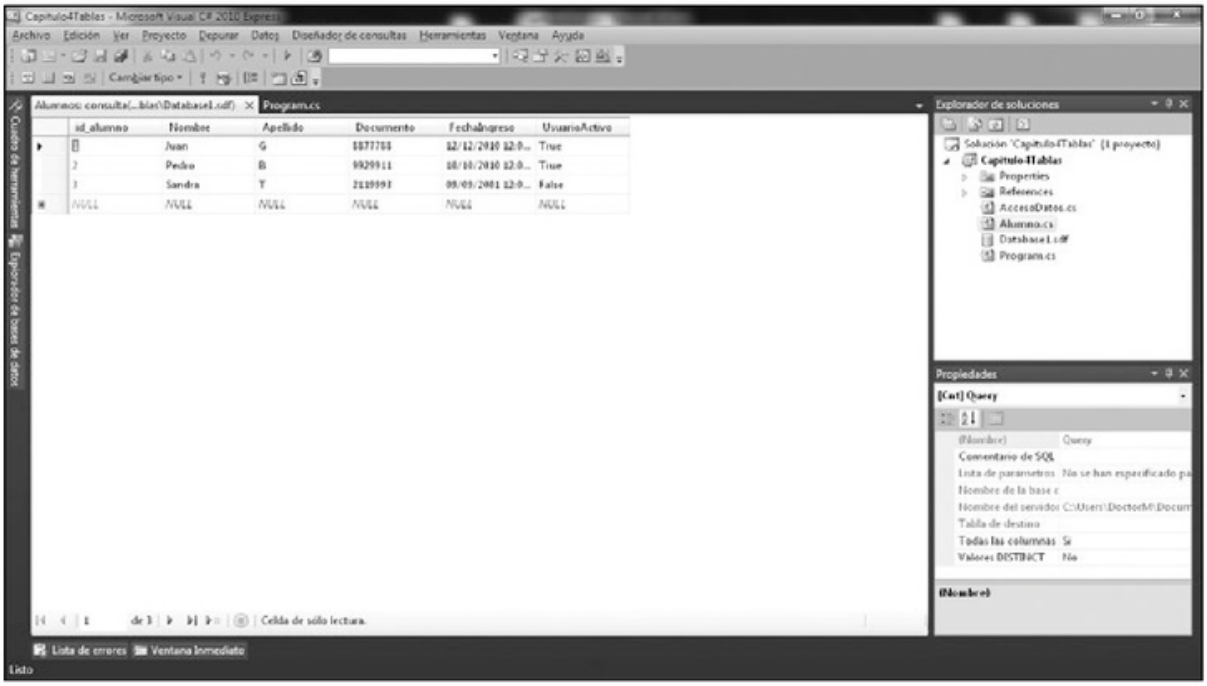


***Figura 2.** Una tabla es representada por el conjunto de campos disponibles para dicha tabla. Los registros, por lo tanto, serán cada una de las entradas que representan el conjunto de campos.*

III BASES DE DATOS RELACIONALES

Las bases de datos relacionales obtienen su nombre a partir de las relaciones que se forman entre sus tablas. Así, una tabla podría depender de la existencia de registros en otra, pero no podrá crear nuevos si no se satisface la relación como primera meta: darnos la posibilidad de rastrear distintas entidades sobre la base de un único dato.

Si bien tenemos a disposición un sinnúmero de bases de datos, nosotros utilizaremos **Microsoft SQL Server 2008 Express**, una versión pequeña y portable de **Microsoft SQL Server 2008**, y que se encuentra incluida dentro de la instalación que ya hemos realizado de **Microsoft Visual C# 2010 Express**. De cualquier manera, los conocimientos que aquí adquiriremos nos serán útiles para otros motores de bases de datos que necesitemos utilizar en el futuro.

Microsoft SQL Server 2008 Express

Microsoft SQL Server 2008 no es por sí solo una base de datos, sino un sistema para el control y mantenimiento de bases de datos. Con Microsoft SQL Server 2008 es posible que tengamos funcionando simultáneamente muchas bases de datos, cada una con sus propias políticas de acceso, usuarios, relaciones y servicios que trabajan de forma independiente o interrelacionadas.

En definitiva, este gestor puede resultar útil en entornos empresariales grandes, que requieren alta disponibilidad de acceso y almacenamiento de los datos u otras cualidades propias de grandes programas. A diferencia de este gran motor, nosotros nos remitiremos al uso de la versión Express de Microsoft SQL Server 2008.

Esta versión se remite a un archivo físico único, el cual tendrá limitaciones, tanto en la cantidad de usuarios concurrentes que acceden a la base de datos, como en el tamaño máximo de la misma base de datos. En la **Figura 3**, podemos ver una lista de las distintas versiones de Microsoft SQL Server 2008.

Alta disponibilidad

Nombre de la característica	Enterprise	Standard	Workgroup	Web	Express	Express Tools	Express Advanced
Compatibilidad con varias instancias	50	16	16	16	16	16	16
Cambios del sistema en línea	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Trasvase de registros	Sí	Sí	Sí	Sí			
Creación de reflejo de la base de datos[2]	Sí (FULL)	Sí (solo seguridad FULL)	Solo testigo	Solo testigo	Solo testigo	Solo testigo	Solo testigo
Agrupación en clústeres de conmutación por error	Máximo del sistema operativo[1]	2 nodos					
AWE dinámicas	Sí	Sí					
Conmutación por error sin configuración del cliente	Sí	Sí					
Recuperación automática de daños desde el reflejo	Sí	Sí					
Instantáneas de la base de datos	Sí						

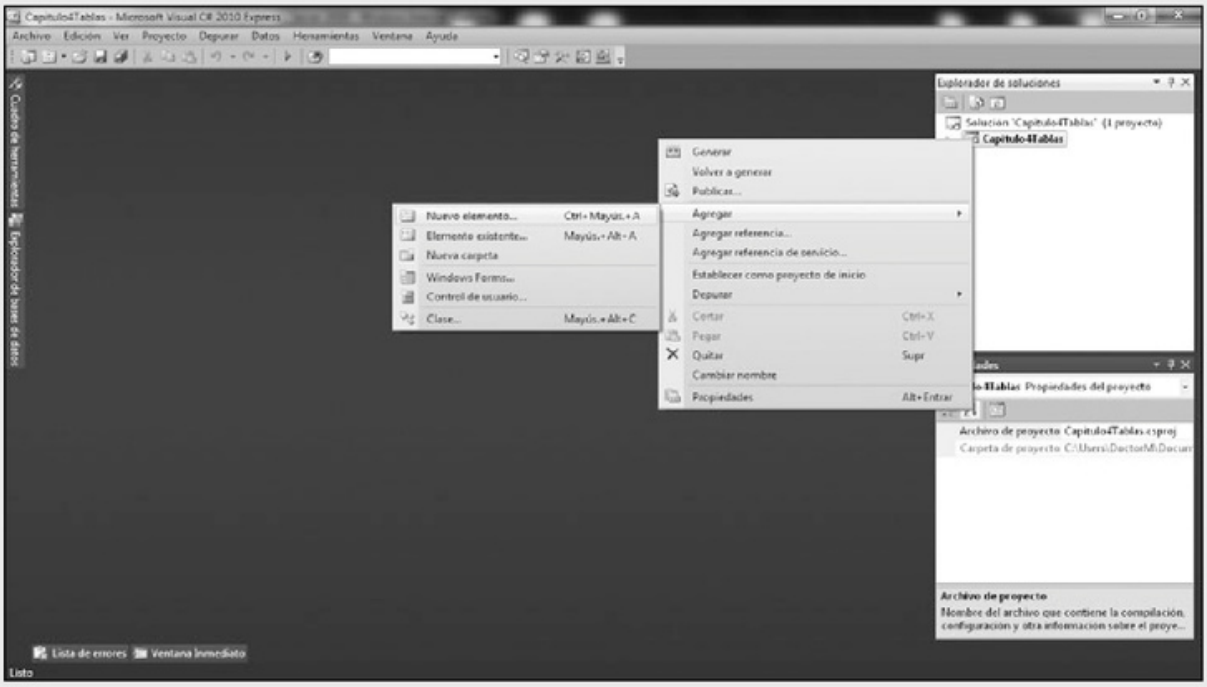
***Figura 3.** Las versiones más avanzadas de Microsoft SQL Server son ideales para manejar grandes volúmenes de datos, así como para hospedarlos en servidores de gran envergadura. Las versiones Express resultan ideales para aplicaciones pequeñas.*

Para que podamos trabajar con una base de datos de Microsoft SQL Server 2008 Express, será necesario crear el archivo de base de datos desde nuestra aplicación.

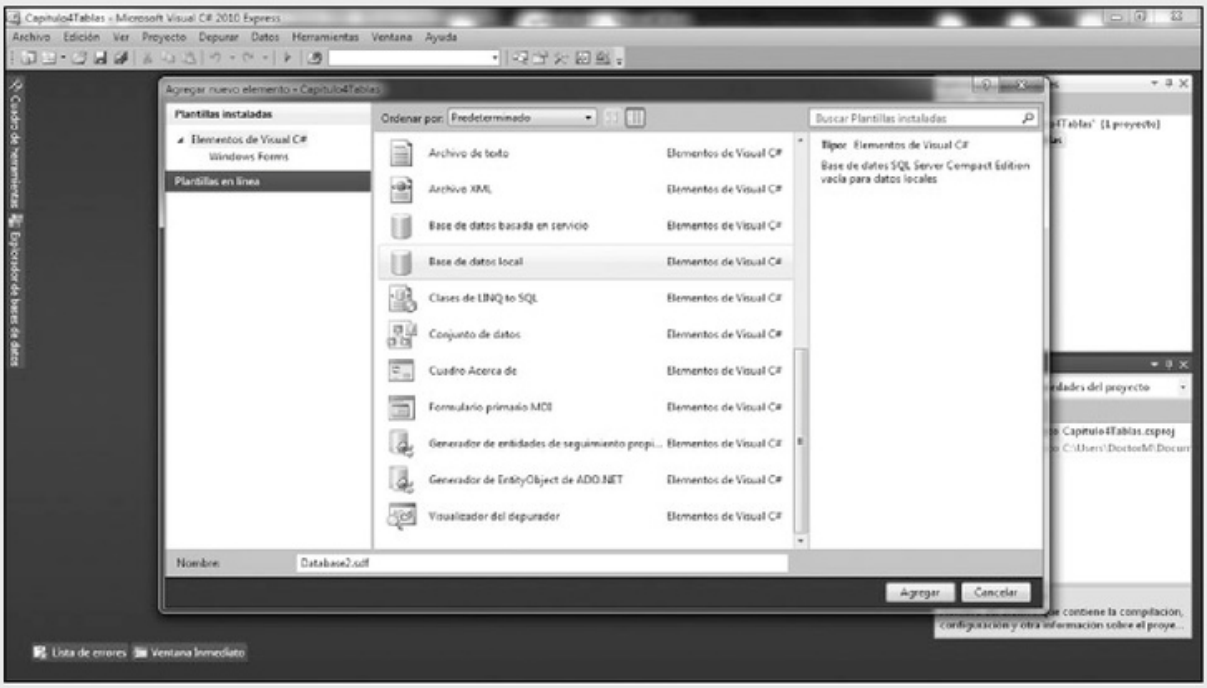
Crear una base de datos

PASO A PASO

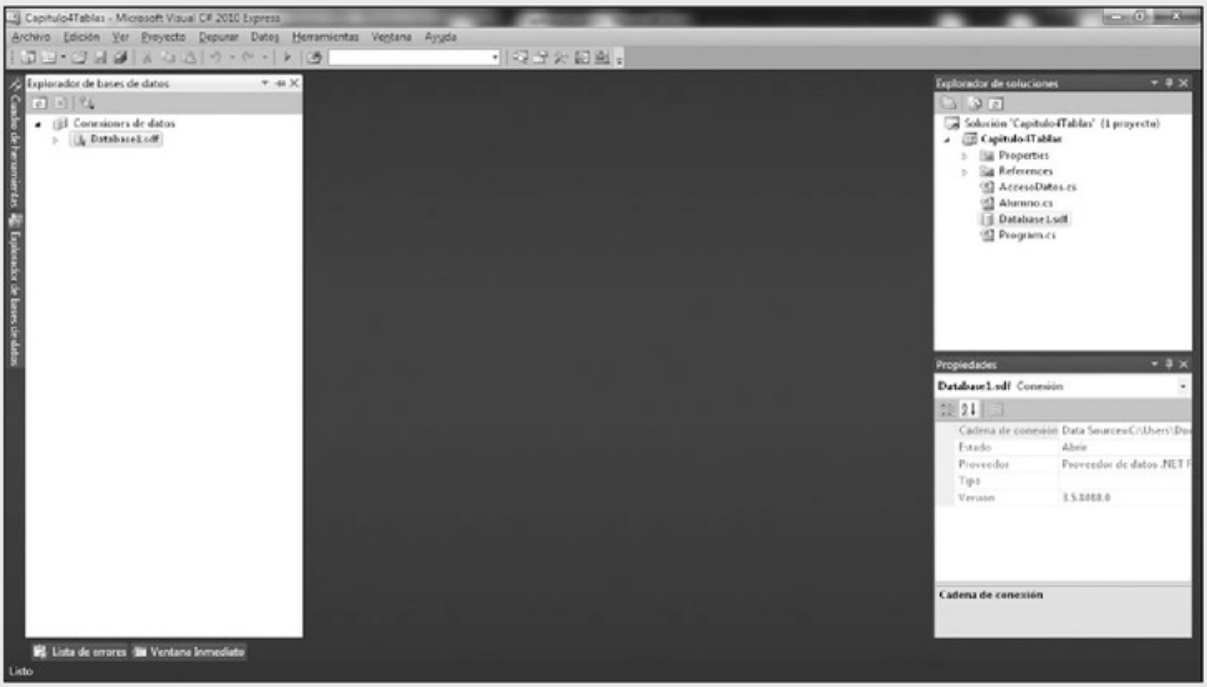
1 Desde el **Explorador de soluciones** de Visual C# 2010 Express, adicione desde el menú **Agregar**, un **Nuevo Elemento**.



2 A continuación, debe seleccionar el tipo de archivo **Base de datos local**, y escribir el nombre correspondiente de la base de datos en la parte inferior de la ventana de diálogo. Este será el nombre físico de la base de datos.



3 Con la base de datos creada, presione dos veces sobre el archivo para abrir el **Explorador de bases de datos**.



Una vez que hemos creado el archivo que servirá para contener la base de datos, podemos iniciar la construcción de la estructura de los datos. Esta estructura estará compuesta por tablas, relaciones y procedimientos almacenados.

Tablas

Las **tablas** dentro de la base de datos, como ya hemos comentado, son la representación física de los datos de nuestro sistema. Estas tablas se componen de campos, donde cada campo puede ser de un tipo específico. Así como en el código C# existen los tipos de datos, los campos de las tablas poseen los suyos, lo que nos ayudará al momento de representar nuestros datos. En la **Tabla 1**, podremos ver los diferentes tipos de datos disponibles en Microsoft SQL Server 2008 Express.

III CÓDIGO .NET EN SQL SERVER

Además de las consultas SQL que podamos realizar en un modelo de bases de datos, Microsoft SQL Server, a partir de la versión 2005, trae consigo la posibilidad de incorporar código C# para la creación de funcionalidad, procedimientos almacenados y servicios de ejecución dentro de la base de datos, que potencian aún más el poder de esta base de datos.

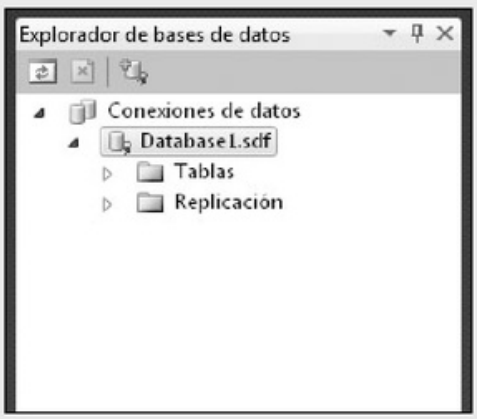
TIPO DE DATO	DESCRIPCIÓN
bigint	Tipo de dato entero con rangos de -2^63 (-9.223.372.036.854.775.808) a 2^63-1 (9.223.372.036.854.775.807).
numeric	Tipo de dato numérico que puede albergar decimales de alta precisión.
bit	Tipo de dato entero que puede almacenar los valores 1, 0 y nulo.
smallint	Tipo de dato entero con rangos de -2^15 (-32.768) a 2^15-1 (32.767).
decimal	Tipo de dato numérico que puede albergar decimales de alta precisión.
int	Tipo de dato entero con rangos de -2^31 (-2.147.483.648) a 2^31-1 (2.147.483.647).
tinyint	Tipo de dato entero con rangos de 0 a 255.
money	Tipo de dato que representa cifras monetarias con rangos de -922.337.203.685.477,5808 a 922.337.203.685.477,5807.
date	Almacena una fecha determinada.
datetime	Almacena una fecha y una hora determinadas.
time	Almacena una hora determinada.
char	Representa caracteres de tamaño fijo y puede almacenar entre 1 y 8.000 caracteres.
varchar	Representa caracteres de tamaño variable y llega a almacenar entre 1 y 8.000 caracteres.
nvarchar	Representa caracteres de tamaño variable en formato UNICODE y puede almacenar entre 1 y 4.000 caracteres.
text	Almacena caracteres hasta un total de 2^31-1 (2.147.483.647).
binary	Almacena valores binarios entre 1 y 8.000 bytes.
image	Almacena valores binarios con un máximo de 2^31-1 (2.147.483.647) bytes.

***Tabla 1.** Tipos de datos disponibles para cada campo dentro de una tabla de Microsoft SQL Server 2008 Express.*

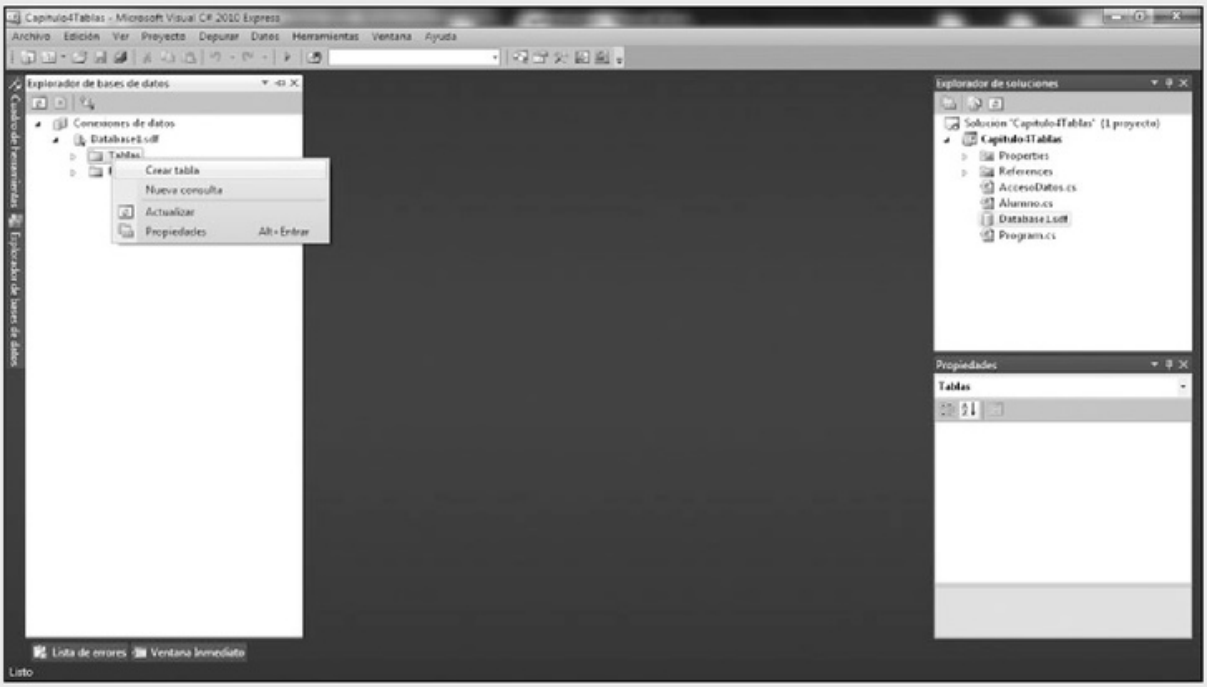
Para crear nuevas tablas dentro de la base de datos, debemos seguir las instrucciones que se encuentran en el siguiente **Pasos a paso**.

Crear nuevas tablas — PASO A PASO

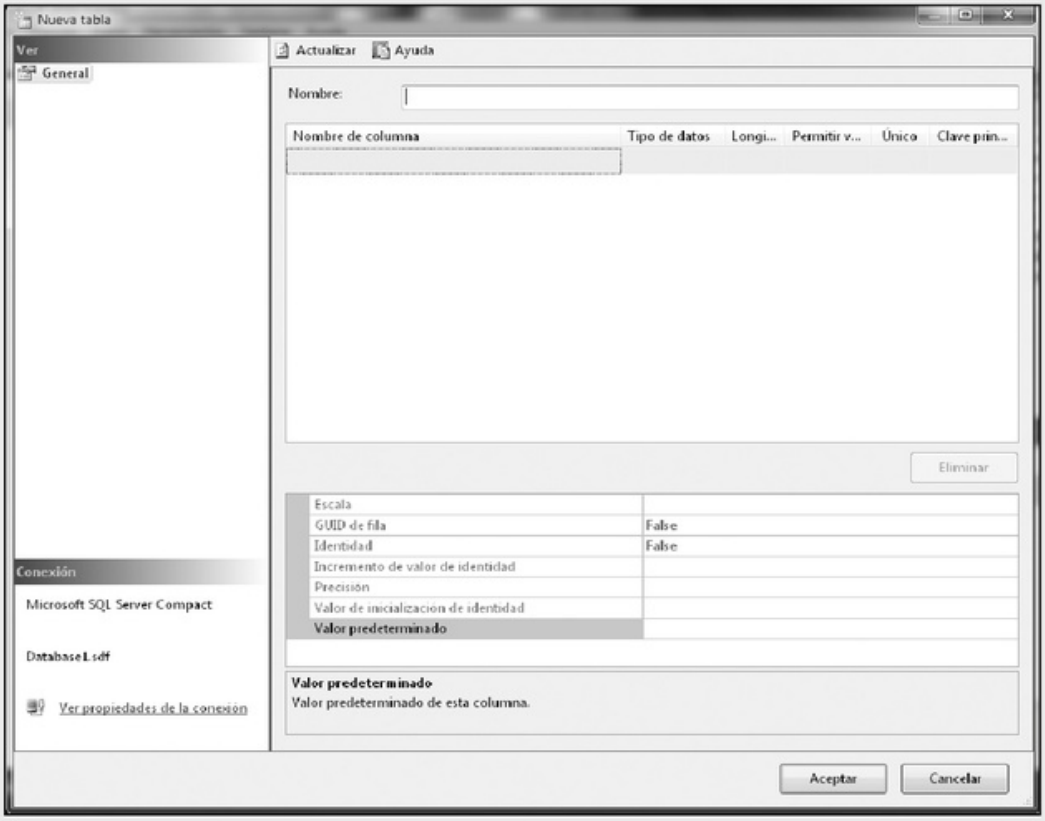
1 En el **Explorador de bases de datos**, seleccione la base de datos previamente creada y expanda su contenido.



2 Seleccione la carpeta **Tablas** y haga un clic con el botón derecho del mouse.



3 Seleccione la opción **Crear tabla**. En el asistente, escriba el nombre de la tabla, y un nombre y tipo de dato para cada columna.



Usaremos el nombre de la tabla para representar el concepto por el cual agrupar la información. Por ejemplo, **Alumnos** o **Cursos** podrían ser dos tablas que contengan tanto la lista de alumnos como la de cursos a los cuales podrán asistir dichos alumnos. Por otra parte, cada columna representará un valor para el registro de alumnos o cursos, como vemos en la **Figura 4** que se encuentra a continuación.

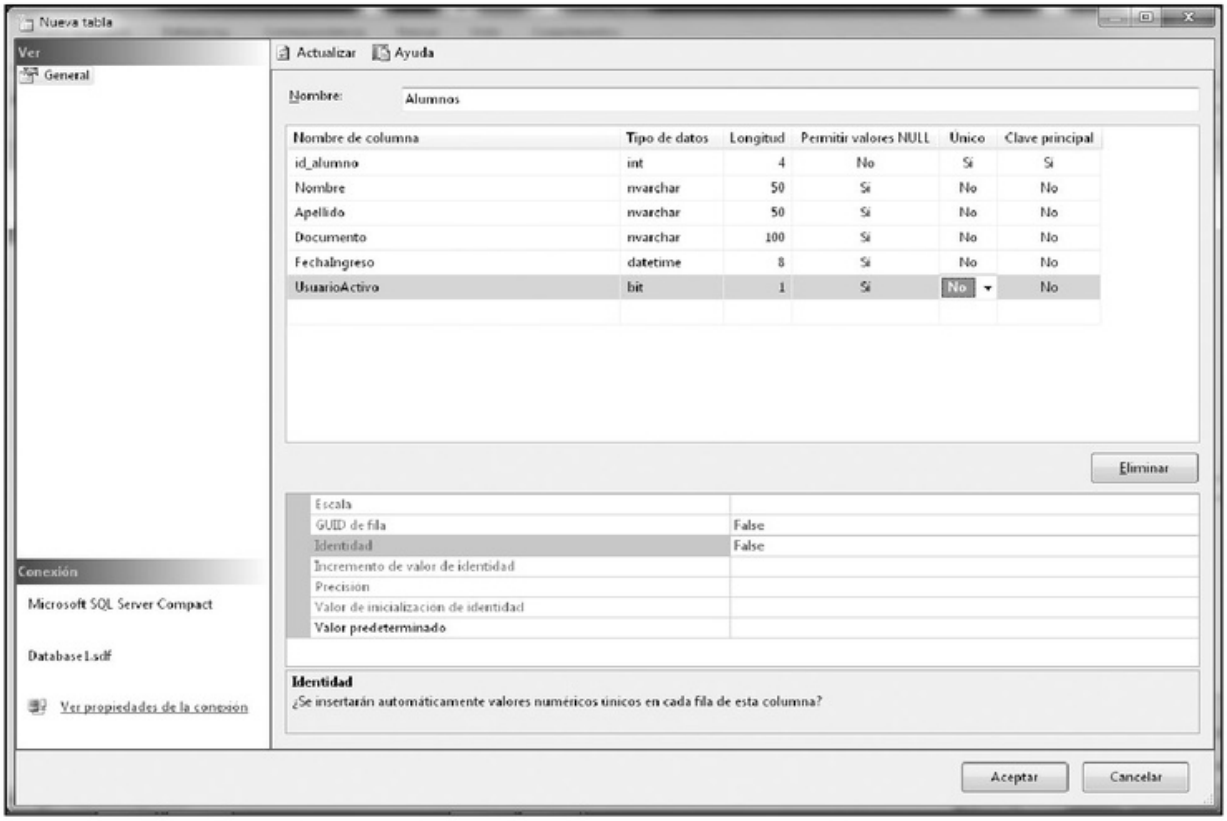


***Figura 4.** La creación de la tabla **Alumnos** incluye cada una de las columnas que representarán un registro único de alumnos. Se utiliza la primera columna como clave única de identificación del registro.*

En la **Figura 4**, podemos ver que cada columna utiliza un tipo de dato específico para representar los valores que contendrá, y así dar forma al registro de alumnos. El siguiente paso es adicionar algunos valores a esta tabla para poder interactuar con ellos desde el lenguaje provisto por el motor de bases de datos (**Figura 5**).

III UNICIDAD DE DATOS

Cuando trabajamos con bases de datos, es necesario encontrar un mecanismo para garantizar la unicidad de los datos. Normalmente se utiliza un valor numérico como primera columna del registro, el cual no se repite entre los registros, y está configurado para que sea autoincremental tras cada inserción de un nuevo registro. Este número es considerado la **llave primaria** del registro.

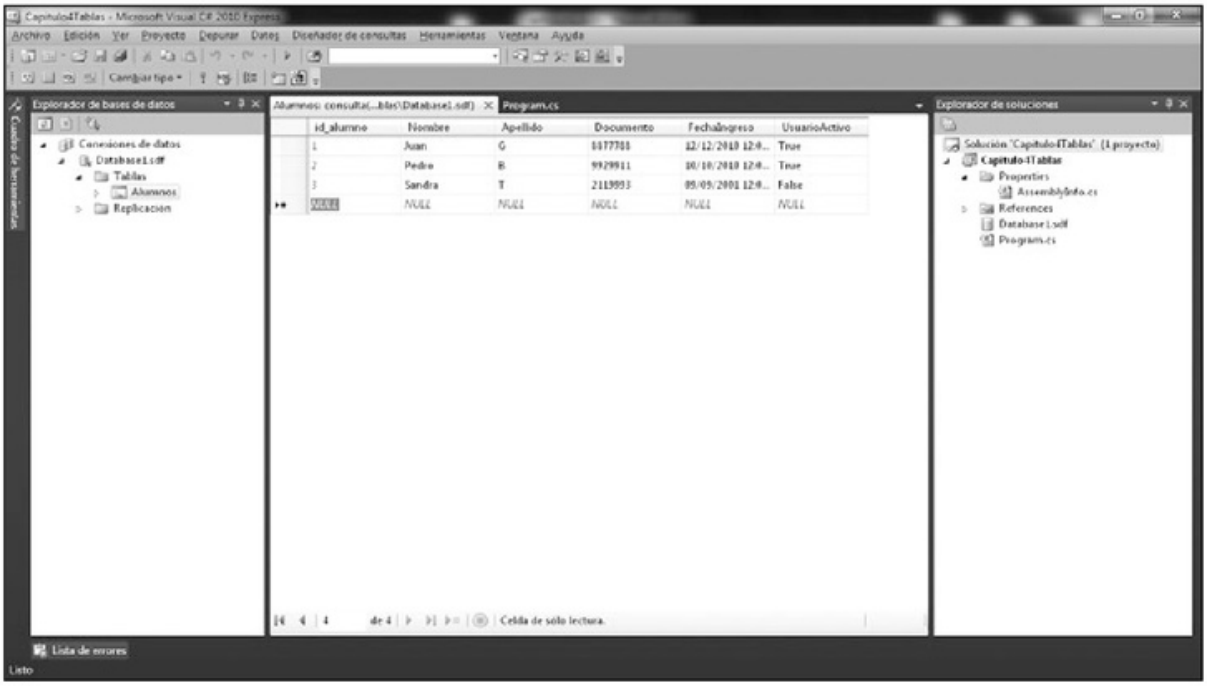


Figura 5. *Adicionaremos algunos registros desde la ventana* ***Mostrar datos de tabla****, a la que podemos acceder presionando el botón derecho del mouse sobre la tabla. Con registros dentro de la tabla es posible realizar diferentes consultas para interactuar con ellos.*

Los motores de bases de datos también cuentan con un lenguaje propio para la ejecución de código, consultar datos de una tabla o crear elementos dentro de la misma base de datos. Este código recibe el nombre de **Transact-SQL** (lenguaje estructurado para consultas transaccional) o simplemente **SQL**. Usaremos este lenguaje para crear la siguiente tabla en nuestra base de datos, así como para recuperar, guardar, borrar o modificar datos de la misma.

```
CREATE TABLE Cursos(
     [id_curso] [int] IDENTITY(1,1) NOT NULL,
     [NombreCurso] [nvarchar](50) NOT NULL,
     [FechaCurso] [datetime] NOT NULL,
     [Activo] [bit] NOT NULL)
```

III ENTITY FRAMEWORK

Las bases de datos relacionales siguen siendo las más utilizadas en los desarrollos y por lo tanto presentan un problema de compatibilidad con la orientación a objeto de nuestro código. **Entity Framework** es una herramienta que suple este problema creando intermediarios que abstraigan la base de datos y sus consultas para dejarnos acceder a los datos en forma de objetos.

Como podemos ver en el código anterior, el lenguaje propuesto por la base de datos no se presenta como de gran complejidad, y puede leerse fácilmente como instrucciones dictadas a una computadora. Podríamos entender dicho código haciendo uso del siguiente pseudocódigo, escrito en nuestro idioma.

```
Crear la tabla Cursos
     teniendo los campos
     id_curso de tipo int como identidad no permitiendo ingresar nulos
     NombreCurso de tipo nvarchar de 50 caracteres no nulos
     FechaCurso de tipo datetime no nulos
     y Activo de tipo bit (Booleano) no nulos
```

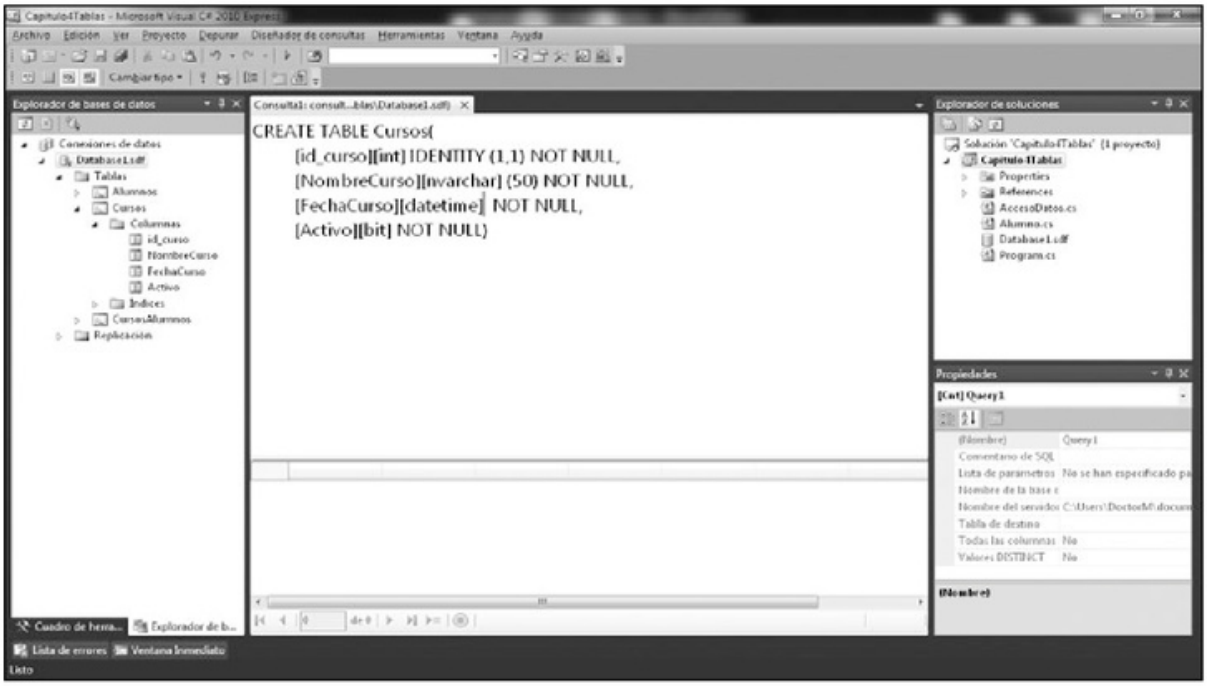


Figura 6. *Diferentes consultas pueden ser creadas y ejecutadas desde el Explorador de bases de datos. Además de dejarnos escribir la consulta SQL por ejecutar, contamos con herramientas de análisis de sintaxis como vista de resultados de la ejecución.*

Al ejecutar la consulta, podremos ver cómo la base de datos ahora incorpora la nueva tabla con los campos designados (**Figura 6**).

Consultas SQL

Las consultas SQL son poderosas y, como ya hemos visto, pueden ser utilizadas para crear, eliminar y modificar tablas, así como cualquier otro objeto dentro de la base de datos. De cualquier manera, nos enfocaremos en aprender aquellas consultas que nos servirán para manipular los datos contenidos en las tablas. En la **Tabla 2**, podemos ver la lista de sentencias más comunes para la manipulación de datos.

SENTENCIA	DESCRIPCIÓN
Select	Selecciona un grupo de datos de una o más tablas sobre la base de las condiciones dadas.
Insert	Inserta un nuevo registro dentro de una tabla.
Update	Actualiza diferentes registros sobre la base de las condiciones dadas.
Delete	Borra uno o más registros de una tabla dependiendo de las condiciones dadas.

***Tabla 2.** Las consultas SQL centran su funcionalidad en la teoría de conjuntos. Los distintos operadores aplican esta teoría para interactuar con los datos.*

Usaremos la sentencia **Select** para obtener registros de la tabla **Alumnos**.

```
SELECT id_alumno, Nombre, Apellido, Documento, FechaIngreso, UsuarioActivo
     FROM Alumnos
```

Como podemos ver, la sentencia **Select** es seguida de los nombres de las columnas de las cuales retornaremos datos. La omisión de cualquiera de estas, simplemente haría que esa columna no fuera retornada. Luego de la lista completa de columnas, debemos especificar el origen (**From**) de dichas columnas y datos, en este caso, la tabla **Alumnos**. En la **Figura 7**, vemos el resultado de la ejecución de estas líneas.

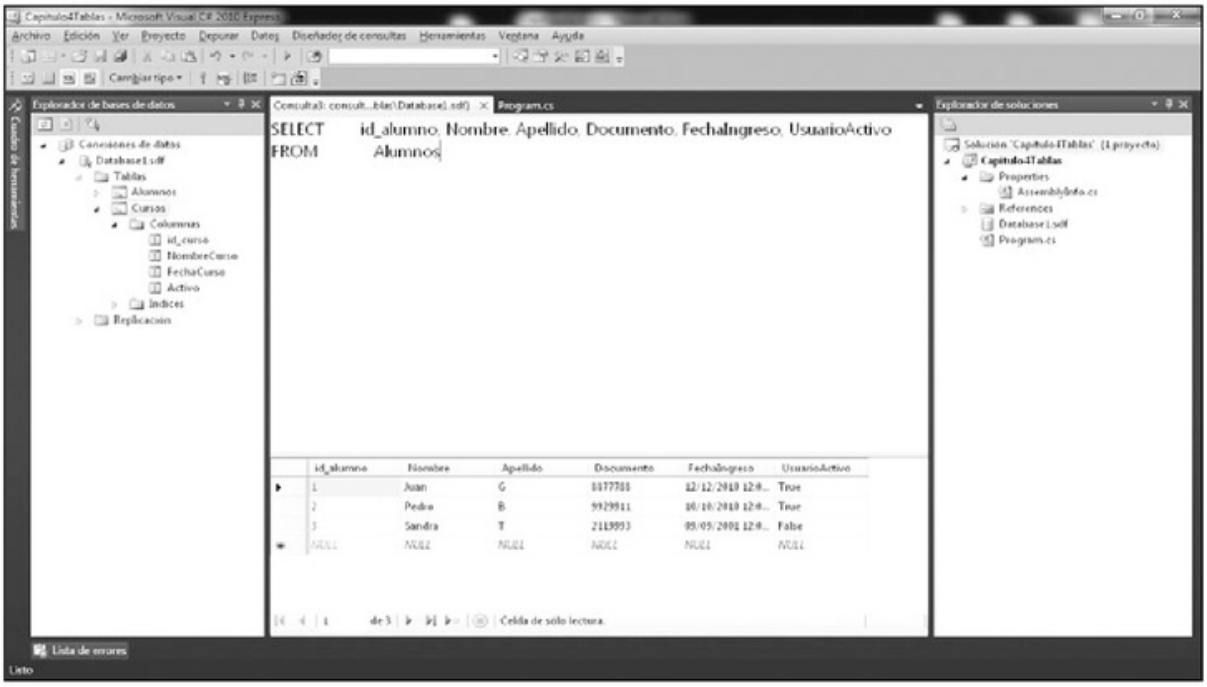


***Figura 7.** Ejecutamos la consulta presionando el icono con el signo de interrogación de color rojo. Abajo, obtenemos los resultados de la consulta realizada.*

Podemos acompañar cualquier tipo de consultas SQL con condiciones de ejecución. Esto quiere decir que es posible condicionar el resultado esperado sobre la base de una evaluación booleana dada por los datos contenidos en la tabla.

```
SELECT id_alumno, Nombre, Apellido, Documento, FechaIngreso, UsuarioActivo
     FROM Alumnos WHERE (UsuarioActivo = 1)
```

Al adicionar la cláusula condicional **Where** (donde), podemos condicionar la búsqueda de información dentro de la tabla sobre la base de la condición dada. En este caso, solo se listarán los usuarios en los que el campo **UsuarioActivo** sea igual a 1 o verdadero (**Figura 8**). Podemos colocar tantas condiciones como necesitemos siguiendo el mismo patrón que hemos utilizado en el código C#, por lo tanto, es posible utilizar los operadores **AND** u **OR** para extender la búsqueda de datos.

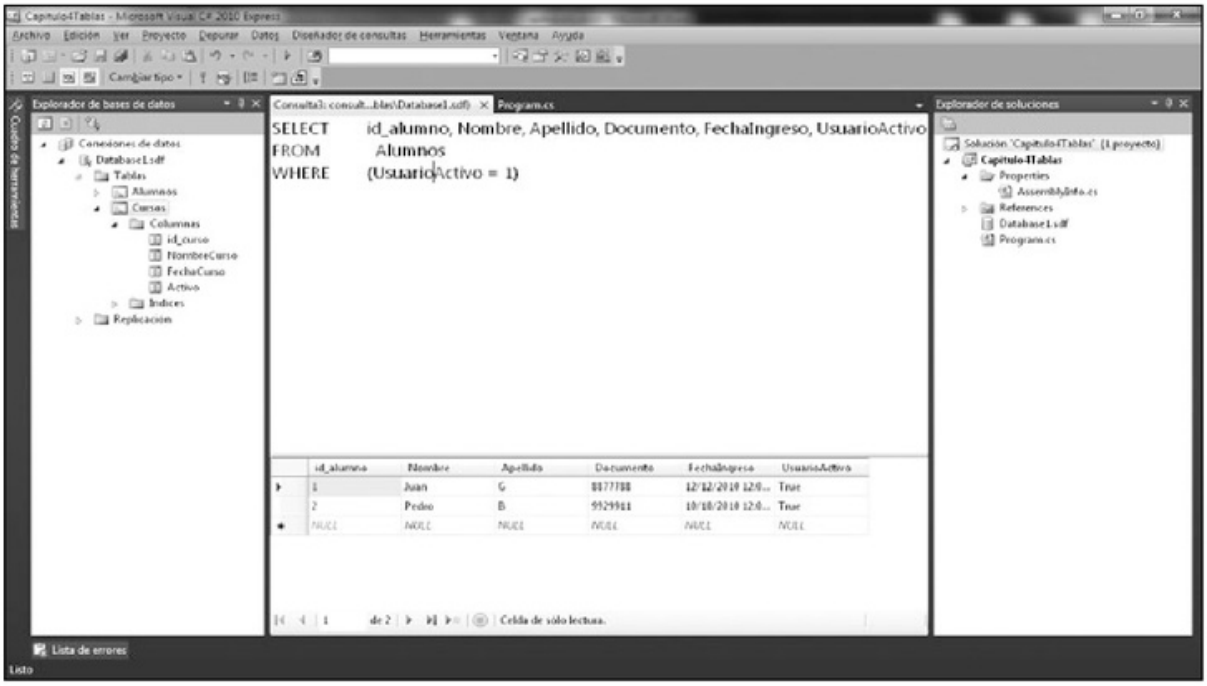


Figura 8. *El código SQL ejecutado retorna solo los registros que cumplan con la condición dada. Podemos agrupar condiciones mediante el uso de los operadores booleanos conocidos.*

La sentencia **Insert** (insertar) es la siguiente de la lista y nos servirá para agregar nuevos registros en una tabla previamente creada.

```
INSERT INTO Alumnos (Nombre, Apellido, Documento, FechaIngreso,
     UsuarioActivo) VALUES ('Maria', 'D', '8788221', '2010/12/12', 1)
```

En este caso, la instrucción de inserción se compone de la sentencia **Insert** acompañada del nombre de la tabla en la cual insertaremos datos, la lista de columnas en las cuales insertaremos los datos y los datos correspondientes para cada una de estas columnas en el orden en el cual las declaramos previamente (**Figura 9**).

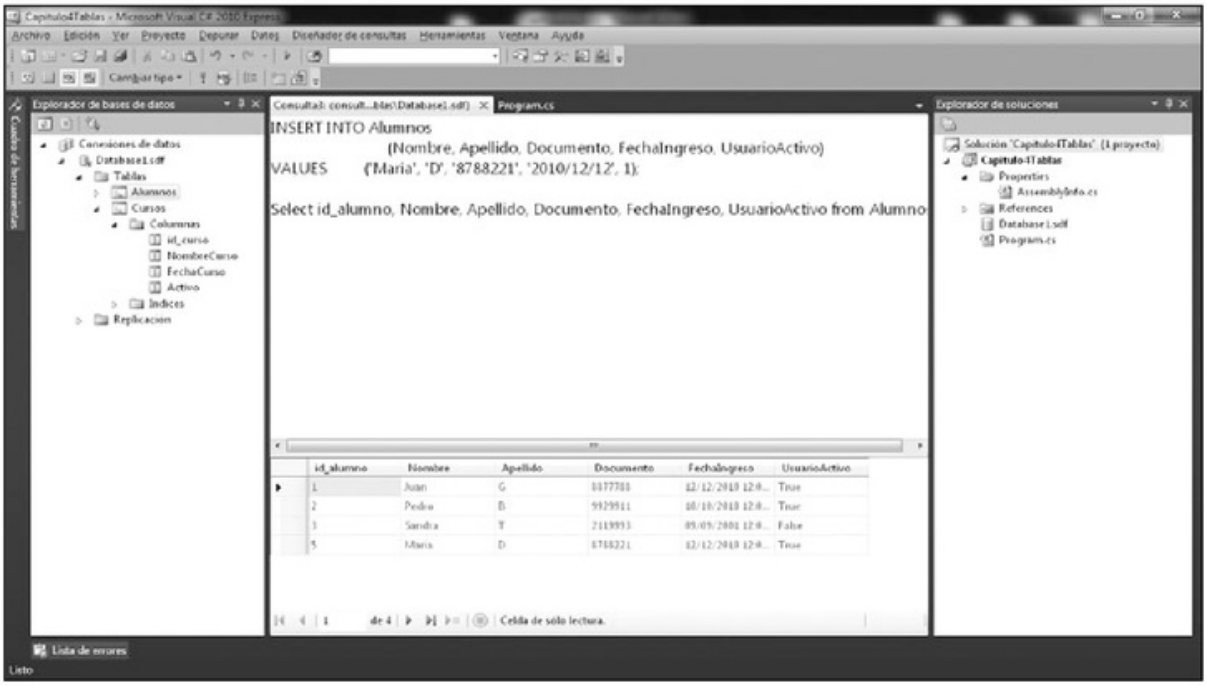


Figura 9. *La primera instrucción inserta un nuevo registro, mientras que la segunda selecciona todos los registros de la tabla para comprobar que, en efecto, aquel se haya insertado.*

Una vez que tenemos registros insertados, es posible, además de seleccionarlos, realizarle modificaciones. La sentencia para realizar este trabajo es **Update** (actualizar).

```
UPDATE Alumnos SET Nombre = 'Juana'
WHERE id_alumno = 5
```

La sentencia **Update** también puede hacer uso de condiciones, así podremos especificar a qué registro en particular necesitamos realizar la actualización. Si obviáramos esta parte de la instrucción, todos los registros de la tabla serían actualizados con el mismo valor, ya que no estaríamos restringiendo la ejecución de la consulta.
La última sentencia para la manipulación de registros es **Delete** (borrar). Esta instrucción no requiere de especificación de columna alguna, aunque deberemos res-

III CONDICIONES EN CONSULTAS SQL

La sentencia **Where** en las consultas SQL funcionan de forma similar a las sentencias utilizadas en el código C# para manipular el flujo del código. Con **Where** no solo podremos anidar condiciones mediante el uso de los operadores **AND** y **OR**, sino que también es posible el uso de los símbolos de mayor, menor, distinto y **NOT**. Esto flexibilizará la búsqueda de datos en una tabla.

tringir los registros que deben borrarse para no eliminar toda la información contenida en la tabla de la misma forma que lo haríamos con una sentencia **update**.

```
DELETE Alumnos
WHERE id_alumno = 5
```

De esta forma, el registro con el valor **5** en el campo **id_alumno** es eliminado preservando intactos los demás registros (**Figura 10**).

Figura 10. *Se elimina el registro con el valor 5 de la columna usada como identificador, y luego se seleccionan todos los valores de la tabla para verificar su eliminación.*

Existen formas más avanzadas de selección y manipulación de datos mediante consultas SQL. La unión de más de una tabla para obtener información referencial o cruzada puede sernos útil en los casos como el que hemos planteado, donde necesitemos saber qué alumnos asisten a qué cursos. Una forma rápida de crear estas

ODBC

Microsoft .Net soporta de forma nativa tres tipos de conectores a bases de datos, dando mayor énfasis sobre SQL Server. Es posible conectarse a cualquier tipo de base de datos sin necesidad de contar con un controlador específico para .Net. Desde el código podemos usar los controladores ODBC configurados en el sistema operativo para conectarnos.

consultas avanzadas es mediante el **Asistente de consultas** provisto por Visual C# 2010 Express, al que podremos ingresar presionando con el botón derecho del mouse sobre la carpeta **Tablas** y seleccionando **Nueva consulta** (**Figura 11**).

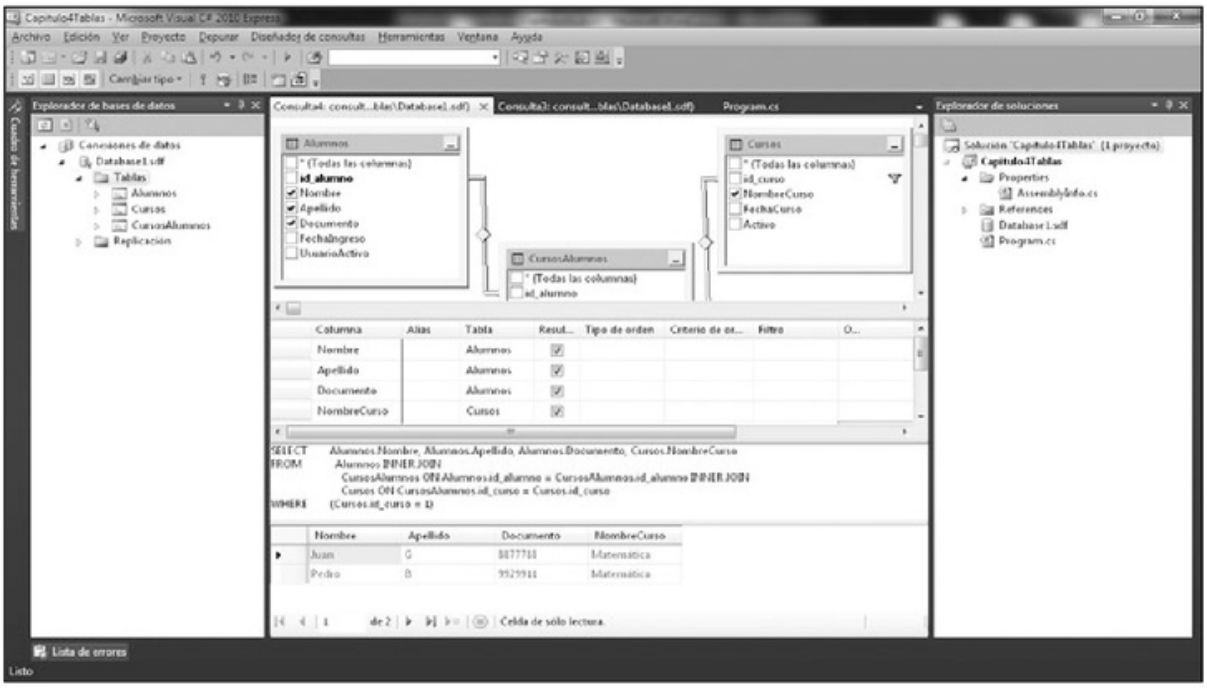


***Figura 11.** El Asistente de consultas nos permite seleccionar las tablas que necesitemos incluir en nuestras consultas, seleccionar los campos que necesitemos retornar y ejecutar la consulta para obtener los registros.*

Al construir la consulta SQL, desde el asistente obtenemos el siguiente resultado.

```
SELECT Alumnos.Nombre, Alumnos.Apellido, Alumnos.Documento,
     Cursos.NombreCurso
FROM Alumnos INNER JOIN CursosAlumnos ON Alumnos.id_alumno =
     CursosAlumnos.id_alumno INNER JOIN Cursos ON CursosAlumnos.id_curso =
     Cursos.id_curso WHERE (Cursos.id_curso = 1)
```

Como vemos, se utilizan las cláusulas **Join** (unir) para unir más de una tabla mediante un campo en común. Normalmente, este tipo de relaciones se pueden conseguir mediante el uso de una tabla intermedia, que realiza la relación entre las dos tablas por medio del identificador único de cada una de ellas como registro. Así, el registro 1 de la tabla **Alumnos** podrá tener su relación con cualquier otro registro de la tabla **Cursos**.

Procedimientos almacenados

La potencia de las bases de datos va en incremento dependiendo del profesionalismo del motor de bases de datos utilizado. Las versiones de SQL Server Express

suelen proporcionar pocas herramientas en comparación con las versiones mayores. Una de las capacidades que suele dejarse de lado es la de **procedimientos almacenados**. Los procedimientos almacenados son piezas operativas que contienen diferentes consultas SQL por ejecutar.
Dentro de un procedimiento, podremos encontrar además, manejo del flujo de ejecución de las consultas, así como el poder enviar parámetros a los mismos para ser incluidos en las consultas que el usuario pudiera realizar.
El código incluido en un procedimiento almacenado es escrito enteramente con lenguaje SQL; no se pueden usar lenguajes como C# para manipular su comportamiento interno. Veamos cómo crear un procedimiento almacenado para nuestra base de datos, pero utilizando Microsoft SQL Server 2008.

```
CREATE PROCEDURE SeleccionarAlumnos
     @id_Materia int
AS
BEGIN
     SELECT Alumnos.Nombre, Alumnos.Apellido, Alumnos.Documento,
               Cursos.NombreCurso
     FROM Alumnos INNER JOIN CursosAlumnos
              ON Alumnos.id_alumno = CursosAlumnos.id_alumno INNER JOIN
              Cursos
              ON CursosAlumnos.id_curso = Cursos.id_curso
     WHERE (Cursos.id_curso = @id_Materia)
END
```

Como vemos, definimos el nombre del procedimiento **SeleccionarAlumnos**, el que recibirá un parámetro de entrada **@id_Materia** que será utilizado en el filtro de la selección de los alumnos correspondientes a la materia determinada.
Una vez creado el procedimiento almacenado, podremos llamarlo mediante su nombre, pasándole el parámetro que este requiere para su ejecución. Los resultados del procedimiento son retornados de la misma forma que una consulta tradicional.

III CÓDIGO DE LA BASE DE DATOS

Cuando necesitamos transportar la base de datos a otra computadora, muchas veces optamos por sacar copias de respaldo de dicha base e instalarla en la siguiente computadora. Los motores de bases de datos profesionales incorporan funcionalidad de **Scripting**, lo que genera el código que construye nuestra base datos en formato texto; así resulta más fácil para su posterior recreación.

```
exec SeleccionarAlumnos 1
```

Al ejecutar la línea anterior, veremos que solo los alumnos que pertenecen al curso 1 son retornados como se ve en la **Figura 12** a continuación.

***Figura 12.** El asistente de bases de datos nos muestra los resultados de la ejecución del procedimiento almacenado. Esto es especialmente útil para probar el código antes de considerarlo completado.*

Los procedimientos almacenados son especialmente útiles para volver a usar la funcionalidad provista por una consulta y así no tener que escribirla directamente en el código C#; también presentan mayor optimización en la ejecución que una consulta simple que pudiéramos ejecutar desde nuestro código. Esto último se debe a a la forma en cómo los distintos motores de datos administran estas consultas.

Acceso a datos

Con el modelo de datos construido en su totalidad, podemos enfocarnos nuevamente en el código C# y en cómo poder acceder a los datos desde nuestras aplicaciones. Microsoft .Net trae consigo un conjunto de clases para el acceso a datos especializado en el tipo de base de datos a la cual queramos acceder. En nuestro caso, como queremos acceder a una base de datos Microsoft SQL Server, haremos uso de la especialización para este tipo de base de datos, aunque también contamos con especializaciones para bases de datos Oracle, Access, por conexión ODBC o por conexión OLEDB, lo que nos permitirá conectarnos a cualquier motor del mercado.

Conectar la base de datos

Para poder trabajar con la base de datos desde C#, necesitamos de tres elementos básicos. El primero de ellos es la conexión a la base de datos. Esta conexión es la encargada de encontrar la base de datos con la cual queremos trabajar, además es la responsable de la gestión entre nuestro código y el motor de la base de datos. La conexión nos asegura la posibilidad de tener acceso a los datos por medio de un canal de comunicaciones entre nuestra aplicación y la base de datos. Lo primero que necesitaremos para poder conectarnos es una cadena de conexión. Esta cadena de conexión es la que contiene todos aquellos datos para poder acceder a la base de datos desde la ruta de la misma base: credenciales de acceso como el nombre de usuario y contraseña, puertos, instancias, entre otros. Veamos un ejemplo de una cadena de conexión.

```
Integrated Security=SSPI;Persist Security Info=False;Initial
     Catalog=BasedeDatosSP;Data Source=.\SQLEXPRESS
```

En la anterior cadena de conexión, se especifica la dirección de la fuente de datos (**Data Source**), además de que se usará seguridad integrada de Windows (**Integrated Security**) para ingresar en ella. De esta forma, no se pedirán credenciales adicionales, como el nombre de usuario y la contraseña, sino que se usarán las del usuario que ejecute la aplicación que se esté conectando a la base de datos. Con esta información, será posible dar el siguiente paso y conectarnos a la base de datos por medio de nuestro código.

```
SqlConnection conexion = new SqlConnection();

//Asignamos la cadena de conexión al objeto Conexión
conexion.ConnectionString = @"Integrated Security=SSPI;Persist Security
     Info=False;Initial Catalog=Database1;Data Source=.\SQLEXPRESS";

//Abrimos el canal de conexión
conexion.Open();
```

III INYECCIÓN SQL

Uno de los principales agujeros de seguridad cuando trabajamos con bases de datos es el denominado **Inyección SQL**. Un usuario mal intencionado podría escribir una consulta SQL que eliminase registros de la base de datos en cualquier campo de texto de nuestra aplicación. El uso de procedimientos almacenados es un camino para solventar este problema de seguridad.

En el ejemplo anterior, creamos un nuevo objeto de la clase especializada **SqlConnection** que, como sus iniciales denominan, sirve para manipular conexiones a motores SQL Server. Mediante la propiedad **ConnectionString** (cadena de conexión), especificamos la cadena de conexión para abrir, finalmente, el canal de comunicaciones con la base de datos.

Ejecución de consultas

Una vez que hemos realizado la conexión a la base de datos, el siguiente paso es el de crear el objeto que hablará con la base de datos mediante las consultas SQL que podamos realizar. Este objeto es llamado **command** (comando) y sirve de intérprete para la ejecución de las consultas SQL y la devolución de los registros que de estas pudieran resultar. Además, los comandos poseen especializaciones para el tipo de ejecución que necesitemos realizar, como aprendimos en este capítulo cuando vimos las consultas SQL; algunas de estas no necesariamente retornarán registros desde la base de datos, por lo que tendremos diferentes propiedades para cada uno de estos casos como podemos ver en la **Tabla 3**.

FUNCIÓN	DESCRIPCIÓN
ExecuteReader	Ejecuta una consulta SQL y retorna todas las filas y columnas que esta pudiera arrojar.
ExecuteScalar	Ejecuta una consulta SQL y retorna la primera columna del primer registro encontrado.
ExecuteNonQuery	Ejecuta una consulta SQL y devuelve el número de registros afectados en la operación.

***Tabla 3.** Tres funciones primordiales para la ejecución de consultas SQL sobre bases de datos.*

Para poder ejecutar consultas SQL, los comandos requieren de una conexión activa hacia la base de datos.

```
SqlCommand comando = new SqlCommand();
comando.CommandText = "SELECT [id_alumno] ,[Nombre] ,[Apellido]
     ,[Documento] ,[FechaIngreso] ,[Activo] FROM [Alumnos] ";
comando.Connection = conexion;
```

La propiedad **CommandText** sirve para alojar la consulta SQL que queremos ejecutar. Es importante abrir la conexión antes de ejecutar cualquier sentencia SQL contenida en un comando, sino obtendríamos una excepción.

Leer datos

El último paso, antes de ejecutar la consulta mediante el objeto **comando**, es tener un lugar donde alojar los datos que vendrán desde la base de datos. Uno de los objetos más rápidos y comúnmente utilizados es el **DataReader** (lector de datos). Este

objeto tiene ciertas particularidades a la hora de su manipulación. Por un lado, es de solo lectura, esto quiere decir que no podremos modificar los datos que este retorne desde una consulta. Al mismo tiempo, la lectura de datos solo se desplaza hacia adelante, y no puede leer un registro anterior que pudiéramos haber olvidado en algún punto; finalmente, este mantiene la conexión abierta mientras dure el acceso a los datos. A pesar de estos puntos que pudieran parecer negativos, sigue siendo el más óptimo en velocidad de acceso y consumo de recursos, frente a otros modelos propuestos como el objeto **DataSet** (conjunto de datos), que puede recolectar datos, desconectarse de la base de datos y mantener estos en memoria para su uso. Aquí solo nos dedicaremos a trabajar con el primero de estos por considerarlo el más adecuado a la hora del desarrollo.

```
SqlDataReader lector;

//Ejecutamos la consulta y
//almacenamos el resultado en
//el objeto SqlDataReader
lector = comando.ExecuteReader();

//Recorremos el lector
//mientras se puedan leer datos
while (lector.Read())
{
    Console.WriteLine("Nombre: " + lector[1] + ", Apellido: " +
     lector["Apellido"]);
}
```

Para poder recolectar los datos contenidos por el **DataReader** será necesario evaluar el resultado de la función **Read** (leer) que este incluye. Esta función retornará **verdadero** si se pudo leer el siguiente dato de la lista de registros retornados por la consulta, por lo tanto, es necesario ejecutar la función incluso si es la primera

III CIERRE DE CONEXIONES

Siempre que usemos conexiones a bases de datos desde código C#, es necesario cerrarlas después de utilizarlas. Así como es necesario abrir el canal de comunicaciones para realizar consultas y manipular datos, cerrar las conexiones a la base es una de las buenas prácticas de código que garantizan la estabilidad de nuestro sistema y del motor de base de datos.

vez que se interactúa con el lector. Esto se debe a que, al inicio, el lector se encuentra en una posición anterior al primer registro encontrado, y se necesita desplazarse al primer registro mediante la primera ejecución de **Read**. Por último, podremos acceder a cada una de las columnas mientras navegamos por los registros mediante el índice numérico de estas en relación al orden en el que se encuentran en la consulta, o por el nombre de la columna retornada (**Figura 13**).

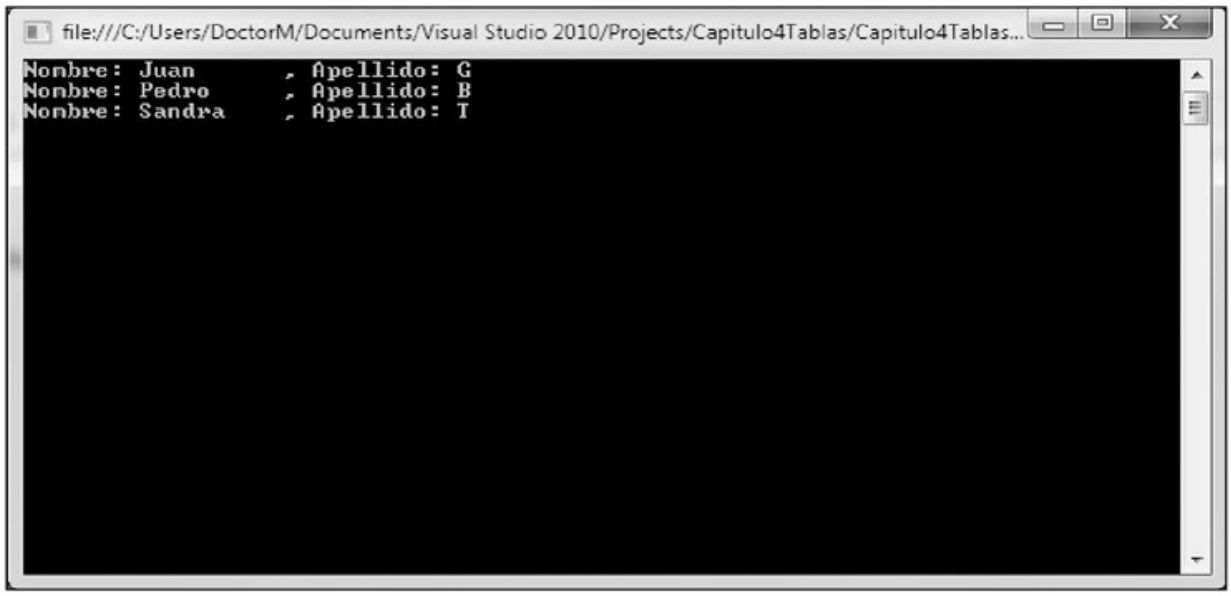


Figura 13. *Mediante un bucle, se recorren todos los registros devueltos por la consulta SQL y administrados por el lector de datos. Cada registro es accesible por medio de su índice de posición o el nombre de la columna dentro de la consulta.*

Tanto **ExecuteScalar** como **ExecuteNonQuery** dentro del objeto **comando** son utilizados para casos específicos como la inserción, actualización y borrado de datos. Aunque también es posible utilizar el primero con consultas de selección. Para entender esta idea, supongamos que necesitamos saber cuántos registros existen en una determinada tabla, o cuántos registros devolverá una determinada consulta; para esto, podemos usar la primera función junto a una instrucción de selección y conteo.
Al ejecutar **ExcecuteScalar** este retornará los valores de la consulta, pero solo podremos acceder a aquel que se encuentre en la primer fila y primer columna de los resultados.

ENTITY FRAMEWORK

Entity Framework puede ayudarnos a abstraer la lógica detrás de las consultas a las bases de datos dejándonos como resultado la manipulación de los distintos objetos que componen nuestro dominio. Para saber más sobre **Entity Framework** ingrese a la siguiente dirección: **http://msdn.microsoft.com/es-es/library/bb399572.aspx**.

```
comando.CommandText = "SELECT COUNT(id_alumno) as Total FROM Alumnos";
int totalRegistros = (int)comando.ExecuteScalar();
Console.WriteLine("Registros encontrados: " + totalRegistros.ToString());
```

ExecuteScalar retorna un valor de tipo **object**. Esto se debe a que el resultado de su ejecución podría ser cualquier tipo de dato soportado por la base de datos, y, por tal motivo, es necesario que nosotros transformemos dicho valor al tipo conocido desde C# para poder manejarlo (**Figura 14**).

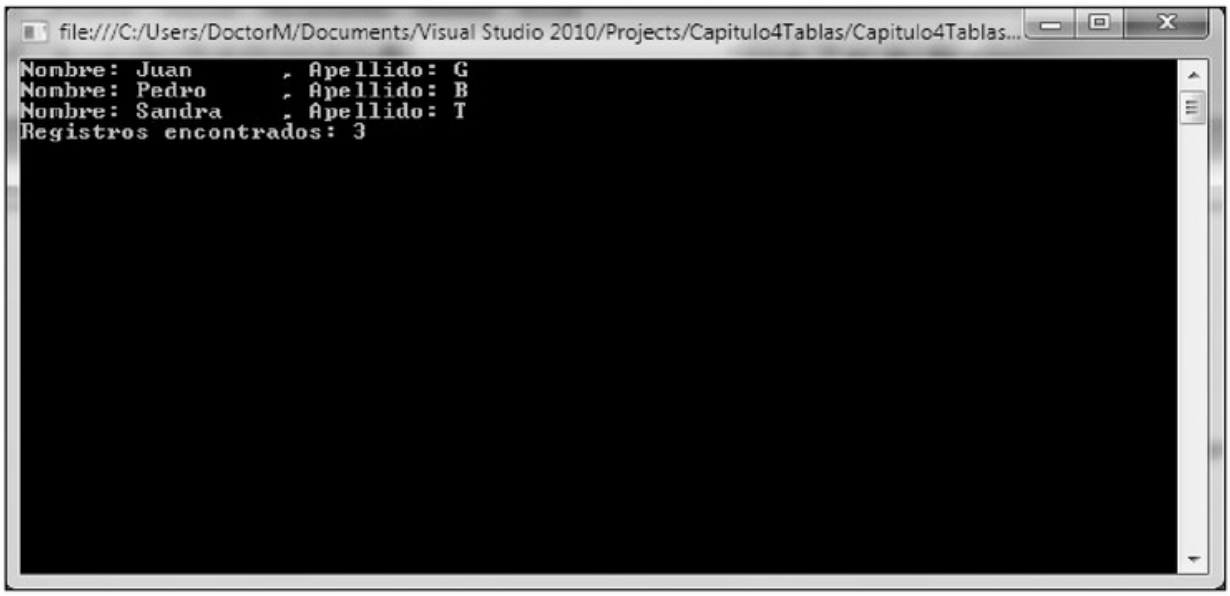


Figura 14. *Ejecutamos una consulta que seleccionará el resultado de contar todos los registros sobre la base de su llave única de identificación. Luego es necesario transformar el resultado a un tipo conocido en C#.*

Finalmente, al utilizar **ExecuteNonQuery**, podremos saber cuántos registros fueron afectados por la operación realizada. Esto puede resultar de utilidad si necesitamos saber si una consulta se ejecutó de forma correcta o no.

```
comando.CommandText = "UPDATE Alumnos SET Activo = 1";
int registrosAfectados = comando.ExecuteNonQuery();
Console.WriteLine("Registros afectados: " + registrosAfectados);
```

Realizamos una actualización de la tabla **Alumnos**. Notemos que no utilizamos el condicionante **Where**, por lo que todos los registros se verán afectados por esta operación. Si tenemos tres registros en la tabla **Alumnos**, cambiarían todos su estado a activos ya que no hemos utilizado ninguna condición al ejecutar la consulta SQL. Esto puede ser un problema en consultas de borrado.

Especializaciones para acceso a datos

Al principio del apartado, dijimos que Microsoft .Net trae especializaciones para los diferentes motores de bases de datos, y, si bien nosotros hemos utilizado aquellos con la raíz SQL (**SqlConnection**, **SqlCommand** y **SqlDataReader**), tendremos disponibles otros con una raíz diferente, pero con funcionalidades similares.

MOTOR DE BASE DE DATOS	OBJETOS	DESCRIPCIÓN
Oracle	**OracleConnection** **OracleCommand** **OracleDataReader**	Utilizado para acceder a bases de datos Oracle. Las librerías no están incluidas directamente en Microsoft .Net y deben ser descargadas desde la página oficial de Oracle.
Todas (OleDB)	**OleDbConnection** **OleDbCommand** **OleDbDataReader**	El controlador OleDB requiere de controladores especificados por el proveedor del motor de bases de datos. OleDB es un modelo de conectividad provisto por Microsoft.
Todas (ODBC)	**OdbcConnection** **OdbcCommand** **OdbcDataReader**	Los tipos ODBC requieren de un controlador de conexión especificado por el proveedor de la base de datos. ODBC hace referencia a Open Database Connectivity (Conectividad de base de datos abierta).

***Tabla 4.** Diferentes modelos de acceso a datos dependiendo de la base de datos a la cual estemos accediendo.*

Notaremos que las diferentes especializaciones para el acceso a bases de datos se comportan de forma similar. Esto quiere decir que tanto la conexión y los comandos como los lectores de datos se comportarán de igual forma independientemente de la base de datos a la cual estemos conectándonos y del objeto que utilicemos para conectarnos a ella. Las diferencias, por el contrario, residen en la cadena de conexión y en la forma de escribir las consultas SQL. Esto se debe a que algunos motores de bases de datos podrían requerir determinados parámetros en la cadena de conexión, así como no soportar determinadas sintaxis o sentencias en la ejecución de código SQL.
Veamos la siguiente cadena de conexión a una base de datos Microsoft Access utilizando el proveedor OleDb, en el código fuente a continuación.

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=[Ruta de la base de
     datos]\baseDeDatos.mdb;Jet OLEDB:Database Password=Contraseña;
```

Notamos la inclusión del atributo **Provider** que especifica el conector que utilizaremos para acceder a la base de datos, y su versión. El siguiente caso también nos sirve para conectarnos a una base de datos Microsoft Access, pero utilizando ODBC.

```
Driver={Microsoft Access Driver (*.mdb)};Dbq=[Ruta de la base de
     datos]\baseDeDatos.mdb;Uid=Usuario;Pwd=Contraseña;
```

Se sustituye el proveedor por **Driver**, el cual hace referencia al uso de Microsoft Access como base de datos. Por lo tanto, si necesitáramos conectarnos a otro motor, solo deberíamos buscar o conocer la cadena de conexión correspondiente.

Independencia en el acceso a datos

Contamos con diferentes gestores para el acceso a datos. Esto nos lleva a tener diferentes tipos y objetos en nuestro código que no son compatibles entre sí, aunque hemos recalcado que todos los objetos poseen un comportamiento similar o, por lo menos, podremos utilizarlos de igual manera independientemente del motor de base de datos y objetos presentes en el código C#. ¿Qué pasaría si necesitáramos crear código para acceder a datos sin importar el motor y los objetos utilizados? Deberemos ahondar en la implementación de cada objeto. De esta forma, descubriremos que todos estos implementan interfaces similares, por lo tanto, poseen un contrato en común, por lo que podríamos utilizar esto para crear la respuesta a la pregunta planteada.

```
public IDataReader EjecutarReader()
{
    SqlConnection conexion = new SqlConnection();
    conexion.ConnectionString = @"Integrated Security=SSPI;Persist Security

     Info=False;Initial Catalog= Database1;Data Source=.\SQLEXPRESS";

    SqlCommand comando = new SqlCommand();
    comando.CommandText = "SELECT [id_alumno] ,[Nombre] ,[Apellido]
     ,[Documento] ,[FechaIngreso] ,[Activo] FROM [Alumnos] ";
    comando.Connection = conexion;

    conexion.Open();

    IDataReader lector = comando.ExecuteReader();

    return lector;
}
```

La función de ejecución, si bien utiliza una conexión y un comando para SQL Server, retorna como valor un tipo **IDataReader**. Por lo tanto, si necesitáramos modificar este código haciendo que la función se conectara con una base de datos Oracle, el código que utiliza esta función seguiría trabajando.

```
AccesoDatos accesoDatos = new AccesoDatos();

IDataReader lector = accesoDatos.EjecutarReader();

while (lector.Read())
{
    Console.WriteLine("Nombre: " + lector[1]);
}
```

El mismo patrón es aplicable a los objetos utilizados para la conexión y la ejecución de comandos. Veamos el siguiente ejemplo.

```
public enum TipoConexion
{
    SQLServer,
    Oracle,
    OleDb,
    ODBC
}

public IDbConnection ObtenerConexion(TipoConexion tipoConexion)
{
    IDbConnection conector;

    switch (tipoConexion)
    {
        case TipoConexion.SQLServer:
            conector = new SqlConnection();
            conector.ConnectionString = ...;
            break;
```

III ANSI SQL

Cada motor de base de datos posee un subconjunto de sentencias para la ejecución de consultas que solo es entendido por este. Programar consultas para dicho motor hará que estas no funcionen en otro motor. El patrón de creación de consultas basado en **ANSI SQL**, detalla las sentencias mínimas que todo lenguaje SQL deberá incluir, garantizando su ejecución en cualquier motor.

```
            case TipoConexion.Oracle:
                conector = new OracleConnection();
                conector.ConnectionString = ...;
                break;
            case TipoConexion.OleDb:
                conector = new OleDbConnection();
                conector.ConnectionString = ...;
                break;
            case TipoConexion.ODBC:
                conector = new OdbcConnection();
                conector.ConnectionString = ...;
                break;
        }

        return conector;
    }
```

Este código retornará una conexión activa sin importar el motor de base de datos que usemos, por lo tanto, podríamos modificar el código que accede a los datos para hacer uso de esta funcionalidad de la siguiente forma.

```
public IDataReader EjecutarReader()
{
     //Obtenemos un conector para SQL Server
    IDbConnection conexion = ObtenerConexion(TipoConexion.SQLServer);

    ...

    conexion.Open();

    IDataReader lector = comando.ExecuteReader();

    return lector;
}
```

De esta misma forma, podríamos también obtener un objeto **comando** genérico y así desacoplar nuestro código de cualquier modelo de base de datos específico, haciendo que este funcione con cualquier motor de base de datos sin tener que realizar modificación alguna o, por lo menos, con una mínima.

Otras fuentes de datos

Las bases de datos no son las únicas fuentes de datos disponibles para nuestras aplicaciones. En la actualidad se hace uso extensivo de **XML** (*Extensible Markup Language,* o lenguaje de etiquetas extensible) para almacenar y transportar información, ya sea dentro de la misma aplicación como entre aplicaciones. Un archivo o un conjunto de datos XML es tratado como simple texto que posee cierto patrón para describir los valores contenidos en ellos.

```
<?xml version="1.0" encoding="utf-16"?>
<Alumno xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Id_Alumno>1</Id_Alumno>
  <Nombre>Pedro</Nombre>
  <Apellido>G</Apellido>
  <Activo>true</Activo>
  <FechaInicio>2010-11-22T12:40:38.3271484-03:00</FechaInicio>
</Alumno>
```

El XML anterior representa un objeto del tipo **Alumno**, donde sus propiedades son representadas por los nodos que comienzan con el símbolo **<** y finalizan con el símbolo **>**, por lo tanto, la descripción de cada dato corresponde a las propiedades de una clase con los mismos elementos.

```
public class Alumno
{
    public int Id_Alumno { get; set; }

    public string Nombre { get; set; }

    public string Apellido { get; set; }

    public bool Activo { get; set; }

    public DateTime FechaInicio { get; set; }
}
```

Para conseguir el XML anterior, primero se ha creado un objeto de la clase **Alumno**, cargado con datos, y finalmente se ha **serializado**, nombre que se le da al proceso de transformar el objeto en memoria en una cadena de texto XML.

```
Alumno alumno = new Alumno();
alumno.Activo = true;
alumno.Apellido = "G";
alumno.Nombre = "Pedro";
alumno.Id_Alumno = 1;
alumno.FechaInicio = DateTime.Now;
```

Por último, haciendo uso de los servicios de serialización provistos por Microsoft .Net, transformamos este objeto en XML.

```
XmlSerializer serializador = new XmlSerializer(alumno.GetType());

StringBuilder stringBuilder = new StringBuilder();
StringWriter escritor = new StringWriter(stringBuilder);

serializador.Serialize(escritor, alumno);
```

Dentro de la variable **stringBuilder**, tendremos el resultado del proceso de serialización. El objetivo de tener este XML es poder almacenarlo, ya sea en el disco de la computadora o en algún otro dispositivo, para su posterior utilización, por lo que una vez serializado, será posible deserializarlo y transformarlo otra vez en un objeto.

... RESUMEN

Durante este capítulo, hemos puesto el énfasis en la manipulación de datos desde las aplicaciones, en especial con el uso de bases de datos, las que nos servirán para resguardar la información de nuestras aplicaciones y recuperarla para su uso posterior. También hemos aprendido sobre la realización de consultas SQL para dialogar con la base de datos y así interactuar con los datos almacenados en ella. Por último, hemos transformado objetos almacenados en la memoria en texto en formato XML mediante el proceso de serialización, y los recuperamos en nuevos objetos por medio del proceso llamado deserialización.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuál es la diferencia entre un DataReader y un DataSet?

2 ¿Podemos desconectarnos de la base de datos mientras leemos datos desde un DataReader?

3 ¿Cómo se llama la clase para conectarse a bases de datos Oracle?

4 ¿Qué controlador usaríamos para acceder a una base de datos de Access?

5 ¿Existen las bases de datos orientadas a objetos?

6 ¿Qué es una base de datos relacional?

7 ¿Qué tipo de dato usaríamos en una tabla de Microsoft SQL Server para almacenar un texto de más de 8.000 caracteres?

8 ¿Qué necesitamos utilizar para crear una función que pueda conectarse a cualquier tipo de base de datos?

9 ¿Qué es XML?

10 ¿Es posible serializar vectores?

EJERCICIOS PRÁCTICOS

1 Intente guardar en un archivo el XML resultante de la serialización.

2 Para deserializar el XML, en vez de tomarlo desde una variable de texto, léalo desde el archivo de texto creado en el paso anterior.

3 En este capítulo, hemos creado una función para conectarnos a cualquier base de datos mediante el uso de interfaces. Intente hacer lo mismo con el comando de ejecución de consultas.

4 Realice una actualización de los registros de una tabla de la base de datos sobre la base de más de un criterio condicional.

5 Cree una tabla que contenga la lista de profesores y cree las relaciones correspondientes a profesores pertenecientes a un curso. Luego, consulte todos los alumnos que asisten al mismo curso que dicte un profesor determinado.

Capítulo 5

Poo en Microsoft .Net

Los conceptos de la programación orientada a objetos son similares en todos los lenguajes de programación que se asocian a este paradigma. Más allá de que se sigan las reglas principales de la orientación a objetos, cada lenguaje, basado en su sintaxis y restricciones, hace uso de diferentes mecanismos para la implementación del paradigma. Aprenderemos aquellas aplicaciones del paradigma propias del lenguaje C#.

SERVICIO DE ATENCIÓN AL LECTOR: usershop@redusers.com

ESPACIOS DE NOMBRE

En capítulos anteriores, hemos aprendido sobre el concepto de clases. Estos mapas o moldes son utilizados para crear distintos objetos funcionales dentro de nuestra aplicación. Estas clases suelen estar asociadas a funcionalidades específicas, como pudimos comprobar con las clases de acceso a datos, teniendo una especialización para bases de datos de Microsoft SQL Server, otras para Oracle y otras para OLEDB u ODBC.
Al mismo tiempo, todas estas especializaciones hacían referencia o heredaban de clases base con funcionalidad más genérica. Con todo esto, podemos darnos cuenta de que la gran cantidad de información retenida en las distintas clases, así como la cantidad de clases existentes, genera gran complejidad administrativa y de conocimiento por parte de los desarrolladores. Encontrar la clase que necesitemos para crear un objeto de ella podría tomar mucho tiempo al tener miles disponibles todas juntas.
Para solucionar este problema, Microsoft .Net hace uso de los **espacios de nombre** para agrupar lógicamente las distintas clases y su funcionalidad sobre la base de un nombre común y de fácil entendimiento. Dentro de estos espacios de nombre, podremos colocar nuestras clases, creando una estructura lógica basada en las funcionalidades contenidas en cada una de nuestras clases.

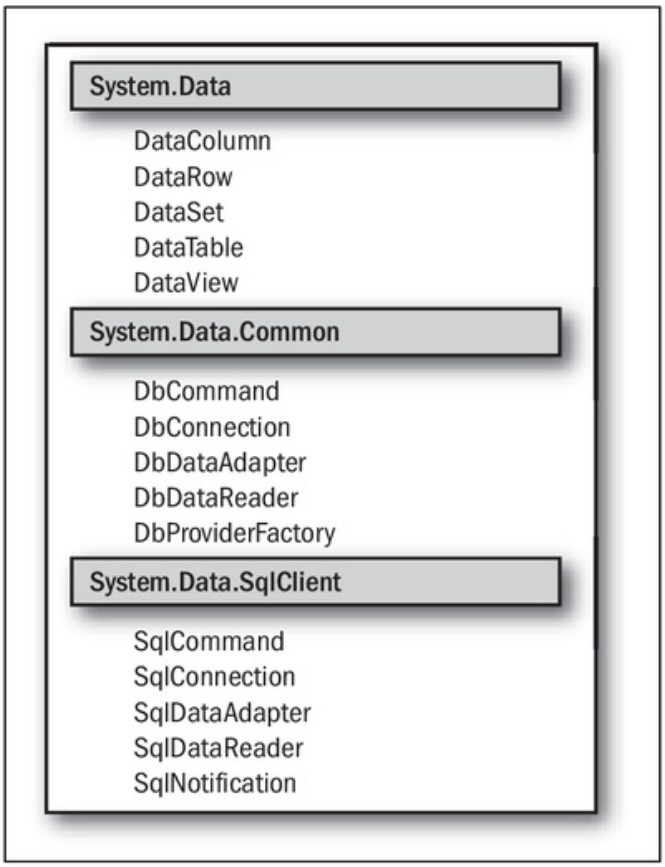


Figura 1. *Las clases de acceso a datos se agrupan en tres espacios de nombre principales. Cada uno de estos contiene todas las especializaciones, tipos y funcionalidades para interactuar con los diferentes motores de bases de datos.*

Para declarar un espacio de nombre, debemos seguir una convención sintáctica específica que vemos en el código a continuación.

```
//Declaramos un espacio de nombre
namespace MiEspacioDeNombre
{

     //Declaración de clases y funcionalidad
     ...
     ...
}
```

Como vemos en el código anterior, un espacio de nombre comienza con la palabra reservada **namespace** seguida del nombre que este espacio de nombre contendrá. Toda la funcionalidad correspondiente a este espacio de nombre deberá estar contenida por él, por lo tanto, las distintas clases, así como las enumeraciones o cualquier otro elemento de código será escrito dentro de las llaves de apertura y cierre que acompañan al espacio de nombre definido.
Una vez que la clase es asociada a un espacio de nombre específico dentro de nuestra aplicación, esto es, que la clase en cuestión se encuentre dentro de la declaración del espacio de nombre, tendremos que anteponer al nombre de clase el identificador del espacio de nombre correspondiente seguido del carácter **.** (punto).

```
//Creamos un nuevo espacio de nombre
namespace EspacioUsuarios
{
     public class Usuario
     {
            ...
            ...
```

III ESPACIO DE NOMBRE POR DEFECTO

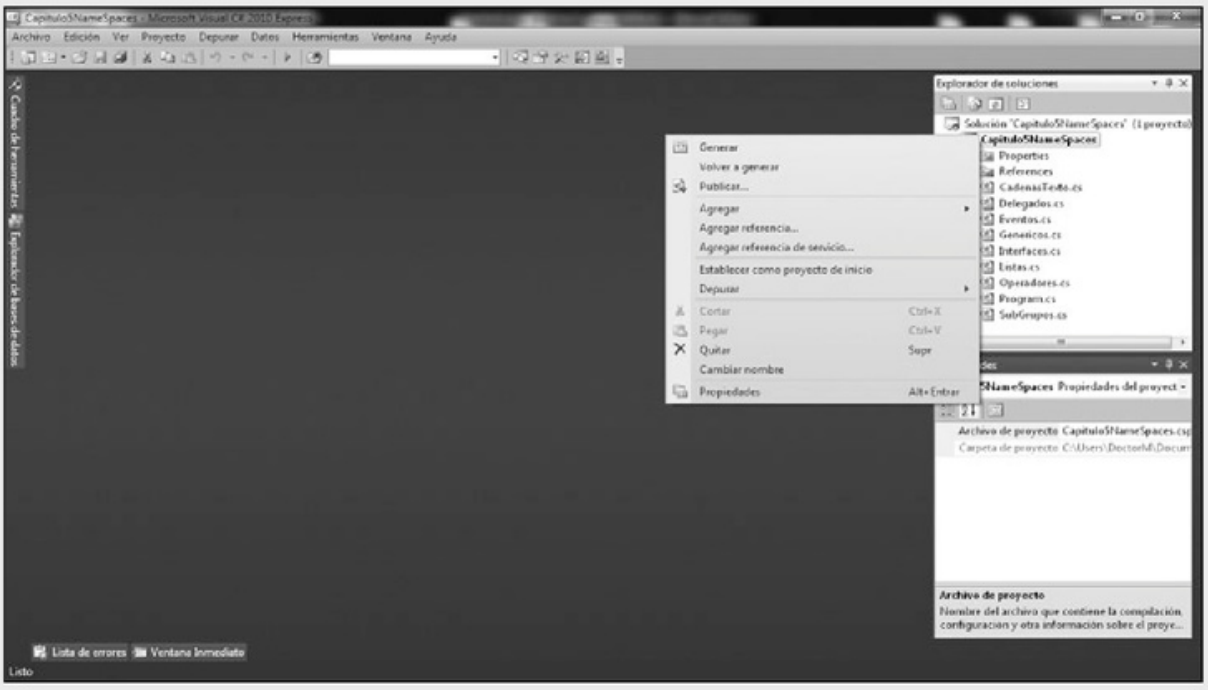
Todas las aplicaciones que creamos en Microsoft .Net poseen un espacio de nombre por defecto. Por lo general, este espacio de nombre es el mismo nombre que le hemos dado a nuestra aplicación al momento de crearla. El espacio de nombre es vital cuando compartimos nuestro código con otros programas, por lo que deberemos elegirlo cuidadosamente.

```
    }
}

...
...

//Creamos un objeto Usuario contenido en el
//espacio de nombres EspacioUsuarios
EspacioUsuarios.Usuario usuario = new EspacioUsuarios.Usuario();
```

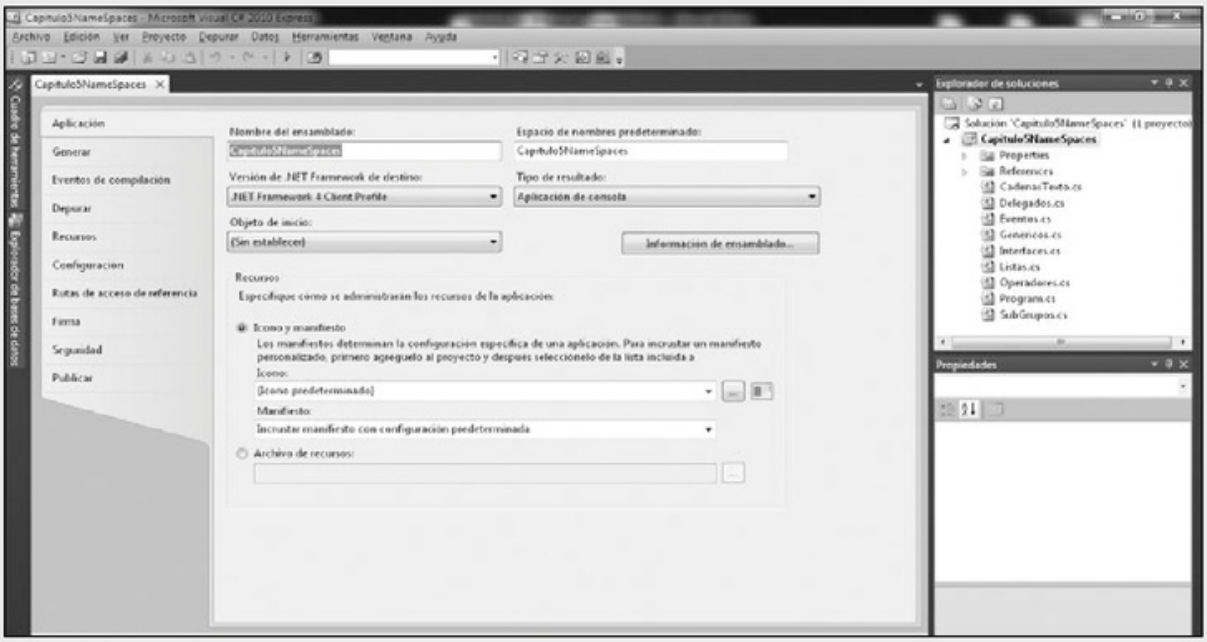
Toda clase que escribamos debe encontrarse dentro de un espacio de nombre, incluso si esta no fuera escrita de forma explícita dentro de un espacio de nombre, será incluida en el espacio de nombre por defecto creado con la aplicación o programa que estemos construyendo. Es posible que para determinados desarrollos, necesitemos modificar el espacio de nombre creado por defecto al momento de crear nuestro proyecto. Si queremos modificar el espacio de nombre por defecto de nuestra aplicación, deberemos seguir las instrucciones del **Paso a paso**.

Espacio de nombre por defecto — PASO A PASO

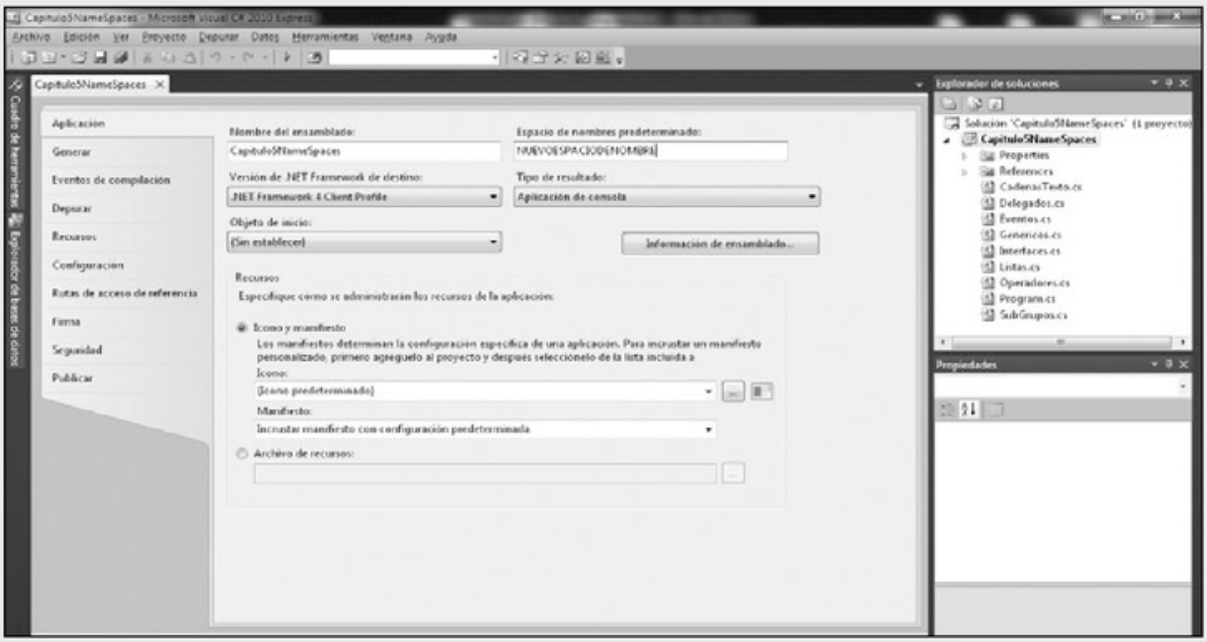
1 Desde el **Explorador de soluciones** de Visual C# 2010 Express, presione con el botón derecho del mouse sobre el **proyecto** y se abrirá el menú contextual que utilizamos para adicionar nuevos elementos al nuevo proyecto.



2 Seleccione la opción **Propiedades** del menú desplegable.



3 Modifique el valor del **Espacio de nombres predeterminado** por defecto con un nombre cualquiera que usted elija.



No es obligatorio, para referenciar algún elemento dentro de un espacio de nombre, hacer uso del espacio de nombre seguido del objeto por utilizar. En la parte superior de la clase, podemos utilizar la palabra reservada **using** seguida del espacio de nombre contenedor del elemento que se utilizará.

```
public class MiClase
{
     public void Funcion()
     {
            //Hacemos uso del espacio de nombre System
            //para acceder a la clase Console
            System.Console.WriteLine("...");
     }
}
```

El código anterior, que hace uso del espacio de nombre junto a la clase **Console** puede ser remplazado para no necesitar especificar el espacio de nombre **System**.

```
//Declaramos el uso
//del espacio de nombre System
using System;

public class MiClase
{
     public void Funcion()
     {
            //Ya no es necesario el uso de System
            //para acceder a la clase Console
            Console.WriteLine("...");
     }
}
```

Esta técnica puede ahorrarnos líneas de código, pero también traernos problemas si encontramos dos clases con el mismo nombre. Si utilizamos la palabra reservada **using** para referenciar ambos espacios de nombre, el compilador no sabrá a cuál de las dos clases estamos haciendo referencia.

```
namespace Espacio1
{
    public class ClaseA
    { }
}
```

```
namespace Espacio2
{
    public class ClaseA
    { }
}

...
...

using Espacio1;
using Espacio2;

//El compilador no sabe cuál de las dos
//clases queremos usar
ClaseA objeto = new ClaseA();
```

El compilador no puede decidir por nosotros a qué clase estamos haciendo referencia ya que existen dos clases llamadas **ClaseA**.

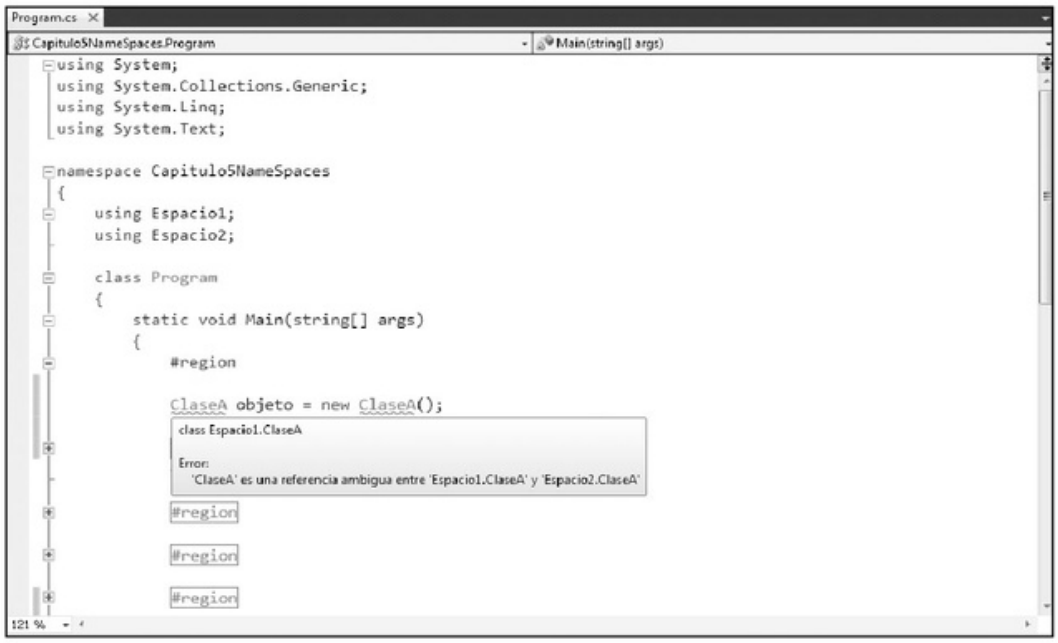


***Figura 2.** El compilador nos advierte que no puede determinar cuál de las dos clases deberá utilizar para instanciar un nuevo objeto.*

Para solucionar este problema, deberíamos colocar el espacio de nombre delante de la clase o podríamos determinarlos agrupándolos de una forma más avanzada. Los espacios de nombre no necesitan reducirse a un único nombre, ya que podemos separar estos en capas o subgrupos mediante el uso del **.** como separador de grupo.

```
namespace Proyecto.Grupo1
{
    class Funcionalidad
    { }
}

namespace Proyecto.Grupo2
{
    class Funcionalidad
    { }
}

namespace Proyecto
{
    class Funcionalidad
    { }
}
```

En el ejemplo anterior, tenemos tres clases llamadas **Funcionalidad**, pero cada una se encuentra en un nivel diferente dentro del espacio de nombre **Proyecto**. Una de estas clases se encuentra en el nivel superior del grupo del espacio de nombre **Proyecto**, mientras las otras dos se encuentran en un subnivel **Grupo1** y **Grupo2**, respectivamente.

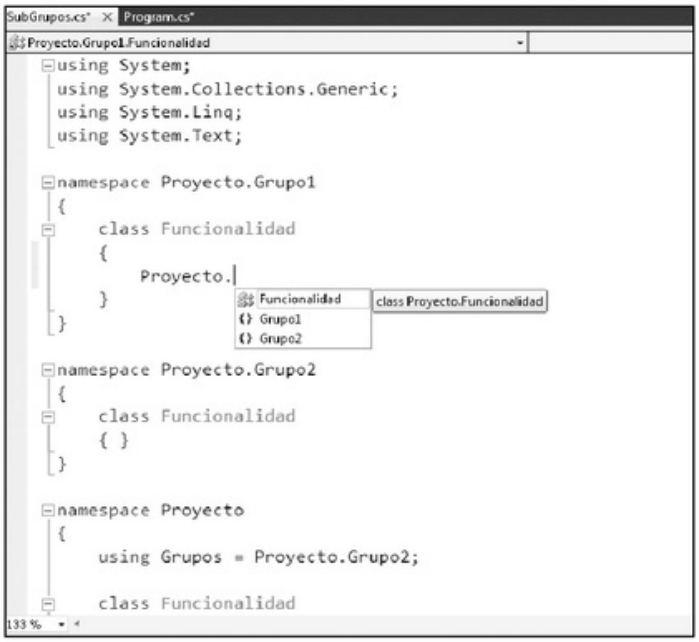


***Figura 3.** El asistente de escritura nos indica, al colocar el espacio de nombre **Proyecto**, que este cuenta con una clase **Funcionalidad** dentro de él y dos espacios de nombres internos.*

Cuando tenemos espacios de nombres muy largos y, al mismo tiempo, necesitamos escribirlos de forma completa en nuestro código por conflictos con los nombres de las clases, es posible asignarles un **alias** o nombre más corto para usar éste en reemplazo del espacio del nombre más largo, haciendo nuestro código más fácil de leer.

```
//El espacio de nombre Proyecto.Grupo2
//será referenciado solo como Grupos
using Grupos = Proyecto.Grupo2;

class Funcionalidad
{
     //Usamos el nuevo nombre Grupos
     //para acceder a la clase
     Grupos.Funcionalidad funcionalidad = new Grupos.Funcionalidad();
}
```

Usando el mismo código que visualizamos en las líneas de código anteriores, hemos designado que el espacio de nombre **Proyecto.Grupo2** será conocido dentro de nuestra clase con el nombre de **Grupos**.

Cadenas de texto eficientes

En el **Capítulo 2**, cuando vimos los diferentes tipos de datos, nombramos el tipo **string**. Este tipo poseía ciertas particularidades en su comportamiento, como su inmutabilidad, lo que causaba un consumo excesivo de memoria si nuestro programa hacía un mal uso de él. Para contrarrestar estos problemas, Microsoft .Net nos provee la clase **StringBuilder**, que se encarga de manipular con eficiencia las cadenas de texto.

```
//Declaración de un objeto StringBuilder
StringBuilder stringBuilder = new StringBuilder();
```

III USING POR ARCHIVO

El uso de **using** para incorporar espacios de nombre dentro de nuestras clases solo se remite al archivo con el cual estamos trabajando. En C#, también podemos tener múltiples clases en un mismo archivo. Si colocamos las etiquetas **using** dentro de la clase, estas solo afectarán a la clase en cuestión y no a todo el archivo.

A diferencia del tipo **string**, el **StringBuilder** es un objeto de mayor complejidad. Al momento de su creación, este reserva un espacio determinado de la memoria para alojar las cadenas de texto con las cuales necesite interactuar.
La memoria reservada será independiente de si asignamos valores al momento de su creación, así, una vez creado el objeto, este podría ocupar la memoria para almacenar 100 caracteres, aunque actualmente no almacenara ninguno; a medida que fuéramos adicionando nuevos, el objeto **StringBuilder** iría ocupando el espacio hasta llenarlo. Esta funcionalidad es la que hace que este objeto sea más eficiente al momento de trabajar con cadenas de texto.

```
DateTime horaInicio = DateTime.Now;
string cadena = "";
for (int i = 0; i < Int16.MaxValue; i++)
{
     //Sumamos la cadena de texto
    cadena += i;
}
DateTime horaFin = DateTime.Now;
Console.WriteLine("Tiempo transcurrido: " + (horaFin -
     horaInicio).TotalSeconds.ToString());

StringBuilder stringBuilder = new StringBuilder();
horaInicio = DateTime.Now;
for (int i = 0; i < Int16.MaxValue; i++)
{
     //Sumamos la cadena de texto
    stringBuilder.Append(i);
}
horaFin = DateTime.Now;
Console.WriteLine("(StringBuilder) Tiempo transcurrido: " + (horaFin -
     horaInicio).TotalSeconds.ToString());
```

USO DE STRINGBUILDER

El objeto **StringBuilder** tiene sobradas ventajas en comparación con el uso del tipo **string**. Lamentablemente, no veremos en el código de los programas un uso intensivo de este objeto. Esto muchas veces se debe al acostumbramiento por parte del desarrollador a usar **string** como variable para manejar cadenas de texto. Este es el momento de usar **StringBuilder** en vez de **string**.

Realizamos un bucle que sumará en forma reiterada esta variable **string** y **StringBuilder** con el valor de la variable **i**. Al finalizar los bucles, se comparan los tiempos de inicio y finalización de estos, comprobando que el segundo, donde se trabaja con **StringBuilder**, se ejecuta significativamente más rápido.

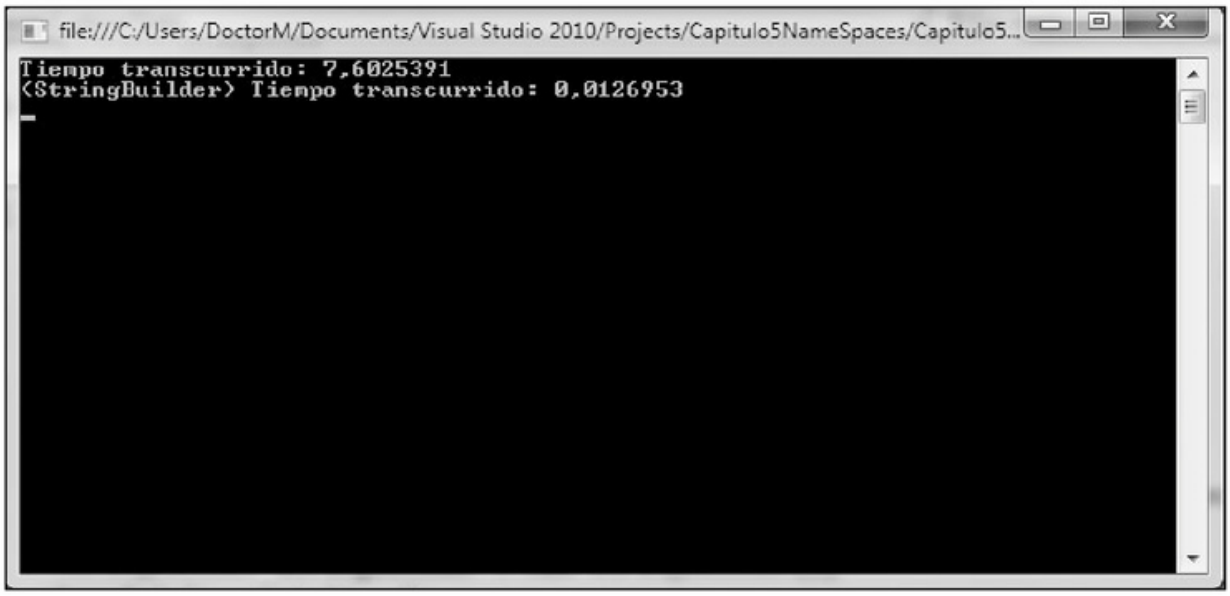


***Figura 4.** La ejecución del bucle mediante el uso de string toma 7 segundos para completar su ejecución mientras que el StringBuilder, solo una fracción de segundo.*

La clase **StringBuilder** posee diferentes funciones para interactuar con las cadenas de texto; en la **Tabla 1** podemos ver la lista completa.

FUNCIÓN	DESCRIPCIÓN
Append	Agrega una nueva cadena de texto a una ya existente contenida dentro del objeto StringBuilder.
AppendFormat	Agrega una nueva cadena de texto basada en un formato dado.
Insert	Inserta una cadena de texto basada en un índice dado.
Remove	Elimina un número específico de caracteres sobre la base de un índice de inicio y un rango dado.
Replace	Reemplaza un texto contenido en el objeto StringBuilder por otro dado.
AppendLine	Adiciona una nueva línea en blanco dentro de la cadena de texto.

***Tabla 1.** Lista completa de funciones disponibles en el objeto StringBuilder.*

III MÚLTIPLES ARCHIVOS

Cuando desarrollamos código en C# es posible tener diferentes espacios de nombre dentro de esta aplicación. Esto se debe a que, por cada archivo de clase, podemos definir un espacio de nombre que contenga las clases y el código que estemos creando dentro de ese archivo. No estamos impedidos de usar un único espacio de nombre para toda la aplicación.

Cuando asignamos distintos valores al objeto **StringBuilder**, si estos sobrepasan el espacio reservado inicialmente, el nuevo espacio es tomado para poder abarcar los nuevos valores, pero no solo se ocupa espacio para contener este valor creado, sino que se reserva un poco más del necesario por si lo requiriésemos para una futura asignación de más texto a esta variable. En el siguiente ejemplo, podemos ver este comportamiento.

```
StringBuilder stringBuilder = new StringBuilder();

//Vemos la capacidad actual del StringBuilder
Console.WriteLine("Capacidad: " + stringBuilder.Capacity);

//Agregamos más caracteres y contamos la capacidad
stringBuilder.Append("Agregamos más caracteres");
Console.WriteLine("Capacidad: " + stringBuilder.Capacity);
```

Al momento de crear el objeto, este contiene una capacidad de **16** caracteres. Luego, agregamos 24 caracteres más, 8 más de los que podía almacenar inicialmente. Al realizar esta acción, el objeto duplica su capacidad, para poder contener los 24 caracteres asignados, dejando 8 más disponibles para un uso posterior.

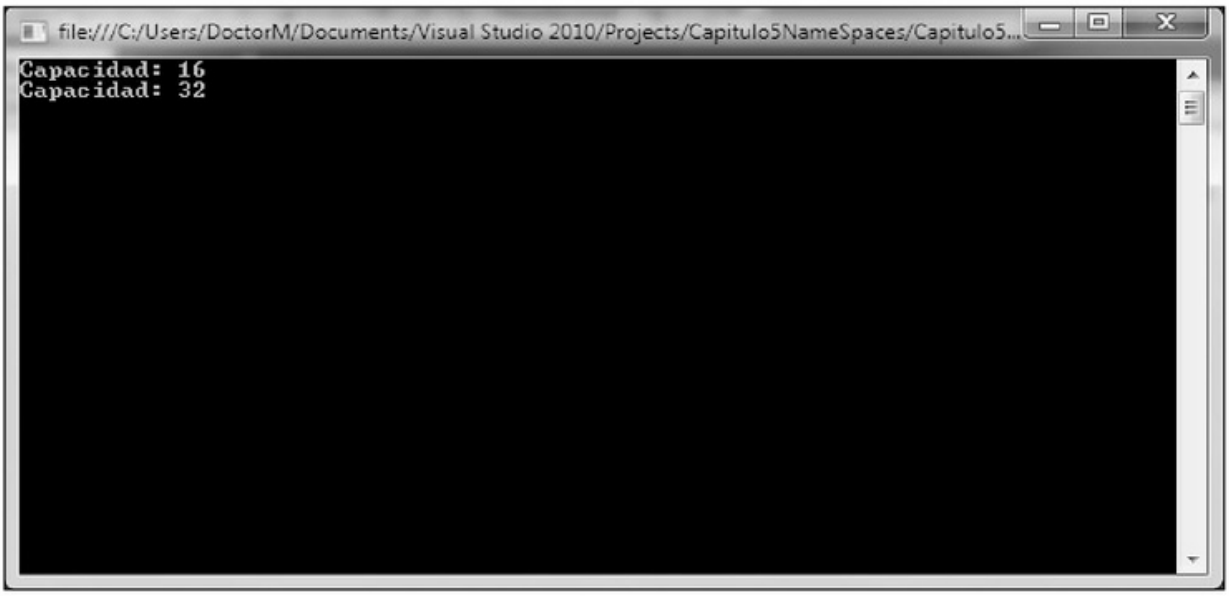


Figura 5. *Toda asignación de datos que supere el espacio designado en un principio hace que el objeto duplique su capacidad para contener los valores y, al mismo tiempo, para reservar espacio para su uso posterior.*

Las distintas funciones provistas por este objeto son similares a las utilizadas por los tipos **string**, salvo **AppendFormat**, que nos permitirá adicionar texto.

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.AppendFormat("{0:D}", DateTime.Now);
Console.WriteLine(stringBuilder);
```

En el ejemplo anterior, se toma la fecha actual y se la transforma a un formato de fecha larga mediante el uso del carácter **D**. Las llaves y el subíndice **0** representan la posición del vector de parámetros que le pasemos a la función **AppendFormat**. Esto quiere decir que podríamos agregar más parámetros separados por el carácter , (coma) y añadiendo un nuevo grupo de llaves con el subíndice correspondiente.

```
stringBuilder.AppendFormat("Fecha larga: {0:D} - Fecha corta: {1:d}",
     DateTime.Now, DateTime.Now);
```

Si agregamos un nuevo parámetro y un subíndice, podemos ver cómo el primer parámetro reemplaza el subíndice **0** y el segundo, el subíndice **1**.

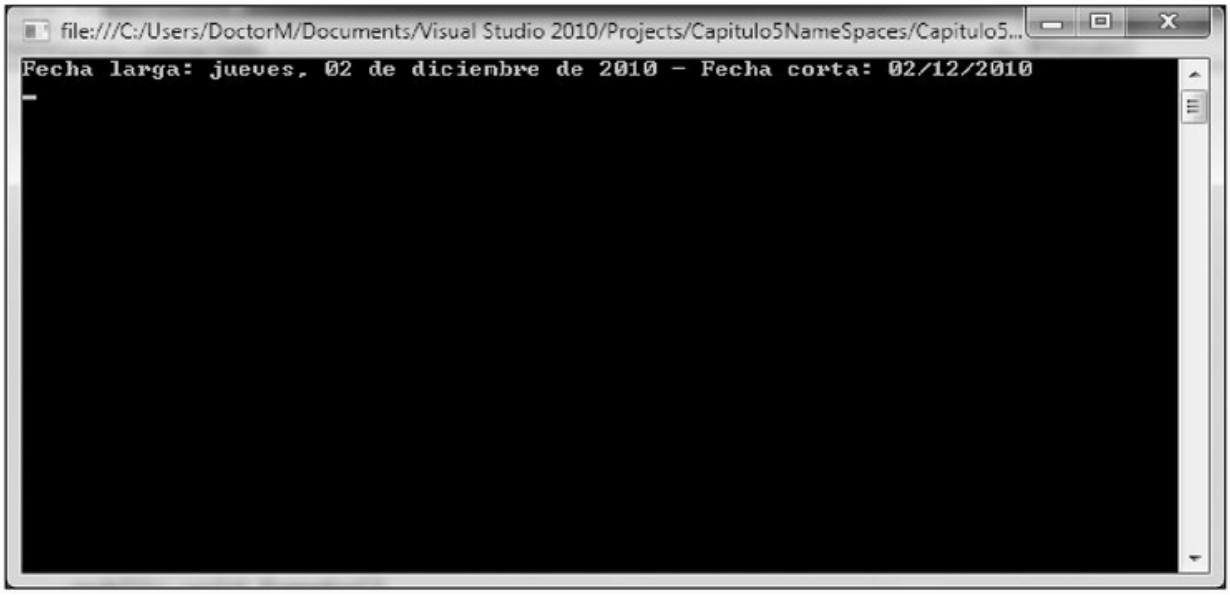


Figura 6. *Las dos fechas representan el mismo valor, pero por el uso del formateador D o d, la primera se muestra en formato largo y la segunda, en formato corto.*

Para obtener los datos contenidos en el objeto **StringBuilder**, podemos utilizar **ToString()**, o hacer uso de la variable como si se tratase de una del tipo **string**.

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("Texto de prueba");
```

```
//Escribimos el texto en la consola
Console.WriteLine(stringBuilder);
```

En el código, utilizamos el objeto como si fuese una variable de tipo **string** y podemos escribir el contenido de esta en la consola haciendo referencia a ella.

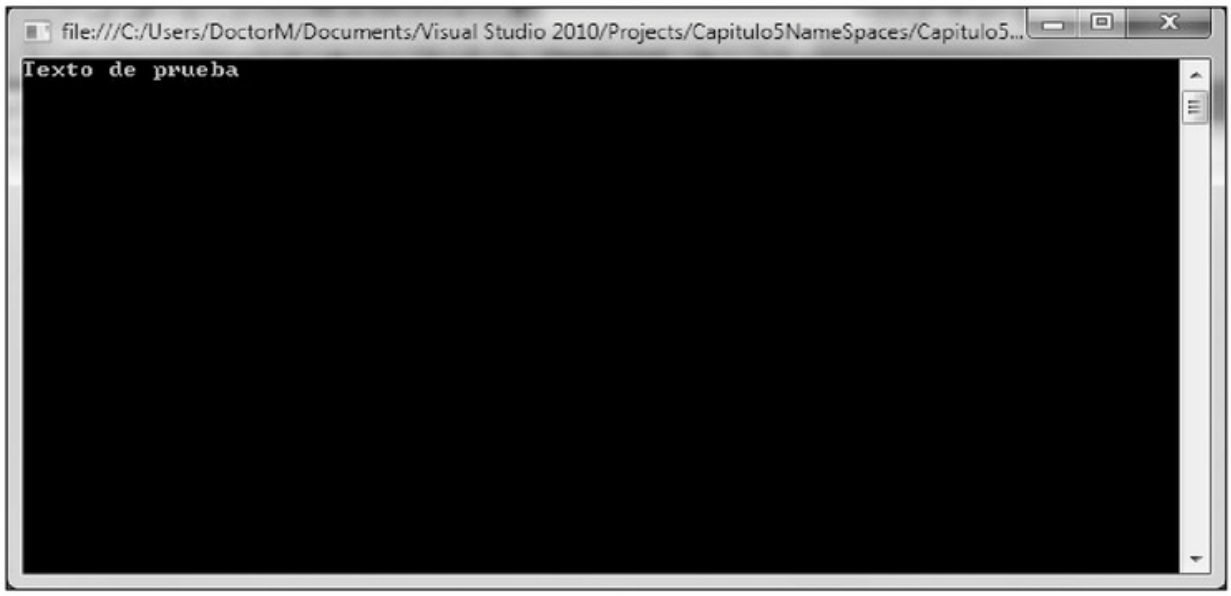


***Figura 7.** Los objetos StringBuilder pueden ser usados como variables de tipo string para obtener su contenido. También podemos utilizar la función ToString() del objeto, con la misma finalidad.*

Interfaces de utilidad

Dentro de la librería de clases que nos ofrece Microsoft .Net Framework, podemos encontrar un conjunto de **interfaces útiles**, una serie de contratos para la realización de tareas específicas. Al implementar algunas de estas interfaces, podremos garantizar la destrucción y remoción de nuestros objetos de la memoria, comparar propiedades de dos objetos o saber si nuestro objeto es igual a otro.

Interfaz IDisposable

En ocasiones, para su funcionamiento, nuestros objetos podrían utilizar internamente otros objetos que, por su tiempo de vida o la forma en cómo son ejecutados, no pueden destruirse por sí solos y liberar la memoria, por lo que necesitamos hacerlo de forma manual, o en todo caso, brindarle al desarrollador una manera estándar y conocida, que garantice esta acción. La interfaz **IDisposable** otorga esta forma estándar y nos obliga a implementar, por contrato con dicha interfaz, la función **Dispose** dentro de nuestras clases. Además, podremos obtener beneficios al combinar esta interfaz con la sintaxis **using**.

```
//Implementa IDisposable
public class Eliminador : IDisposable
{
     ...
     ...
     //Implementación de la interfaz IDisposable
    public void Dispose()
    {
            ...
    }
}
```

Dentro de la función **Dispose**, deberemos colocar todo el código requerido para la liberación de recursos. La implementación de **IDisposable** posee un patrón común de implementación para garantizar que el motor de ejecución de Microsoft .Net no intente eliminar nuestro objeto de forma equivocada. Esto lo podemos visualizar en el código de línea que se encuentra a continuación.

```
private bool eliminado = false;
private SqlConnection conexion = new SqlConnection();

public void Dispose()
{
    Dispose(true);
     //Prevenimos que el Garbage Collector
     //vuelva a llamar el finalizador de clase
    GC.SuppressFinalize(this);
}

private void Dispose(bool eliminando)
{
    if (!this.eliminado)
    {
        if (eliminando)
        {
              //Eliminamos los recursos
            Console.WriteLine("Recursos eliminados");
            conexion.Dispose();
        }
```

```
            eliminado = true;
        }
    }
```

Todos los recursos por eliminar son colocados dentro de la condición de la función privada **Dispose**. Solo se accede a esta condición cuando la función **Dispose** pública es llamada por el desarrollador. En este ejemplo, la conexión a la base de datos es destruida junto con nuestra clase mediante el uso de esta función **Dispose** provista por el contrato **IDisposable** dentro de la clase **SqlConnection**.

```
Eliminador eliminador = new Eliminador();
eliminador.Dispose();
```

Al ejecutar la función **Dispose** en nuestro objeto, observamos cómo el objeto **SqlConnection** interno es destruido, como vemos en la imagen siguiente.

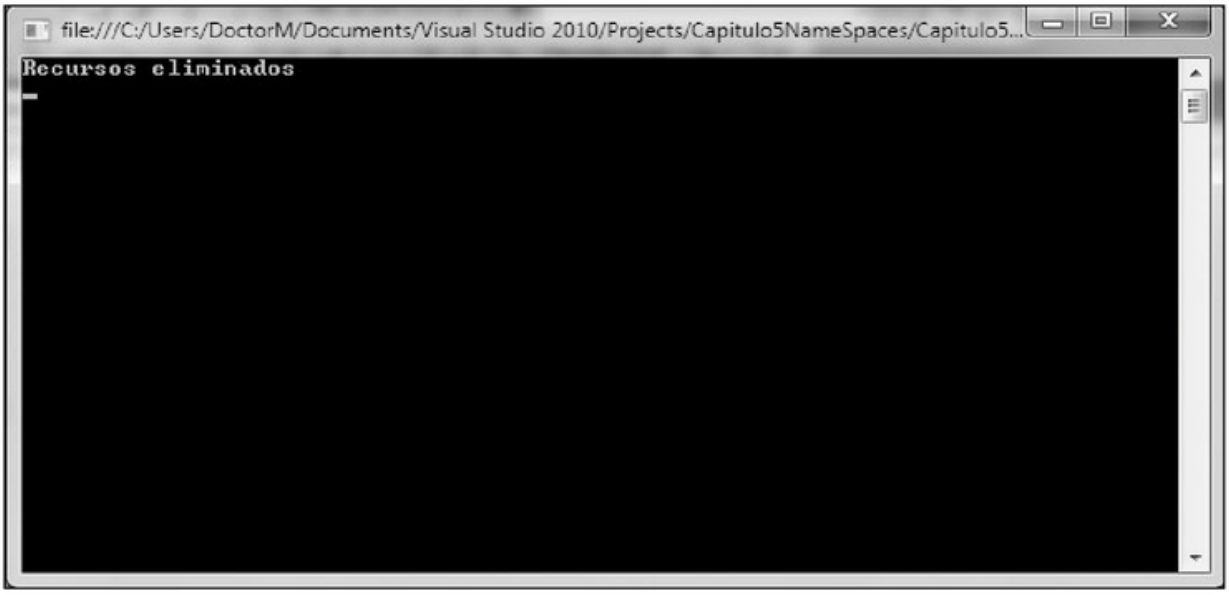


Figura 8. *Una vez ejecutada la función Dispose de nuestro objeto, los recursos utilizados dentro de nuestra clase son liberados utilizando, también, la implementación IDisposable.*

Interfaz IComparable

Esta interfaz nos permite comparar dos objetos basados en una o más propiedades de los objetos dados. Al implementar la interfaz **IComparable**, deberemos crear la función **CompareTo**, función que es parte del contrato de la interfaz y es donde deberemos crear el código de comparación. Esta función será llamada por el código en ejecución cuando se realize la comparación de dos objetos de este tipo.

```
public class Comparador : IComparable
{
     //Implementación de IComparable
    public int CompareTo(object obj)
    {
     ...
     ...
    }
}
```

El valor retornado por la función **CompareTo** también está estandarizado y deberemos respetarlo en el momento de crear la funcionalidad de comparación.

VALOR DE RETORNO	DESCRIPCIÓN
-1	El objeto comparado es menor que el objeto pasado como parámetro.
0	Ambos objetos son iguales.
1	El objeto comparador es mayor que el objeto pasado como parámetro.

***Tabla 2.** Valores retornados por la función CompareTo.*

La implementación de la función **CompareTo** dependerá de nuestras necesidades a la hora de comparar los dos objetos, como vemos en el código fuente a continuación.

```
public int Velocidad { get; set; }

public int CompareTo(object obj)
{
    Comparador objeto = obj as Comparador;
    if (objeto != null)
    {
            //Comparamos sobre la base de la propiedad Velocidad
        return this.Velocidad.CompareTo(objeto.Velocidad);
    }
    else
    {
            //Si el objeto enviado no es del tipo
            //esperado, arrojamos un error
        throw new ArgumentException("El objeto a comparar no es válido");
    }
}
```

Notemos que usamos la propiedad **Velocidad** de la clase como punto de comparación, pero podríamos utilizar tantos elementos como necesitáramos. Para poder comparar en la primera línea, debido a que estamos recibiendo un objeto cualquiera, es necesario que verifiquemos si el objeto es del tipo que nosotros esperamos, si no lo fuere, arrojaremos un error ya que la comparación no podrá ser realizada. Si queremos probar el funcionamiento de nuestro código, deberemos hacer lo siguiente.

```
Comparador comparador1 = new Comparador();
Comparador comparador2 = new Comparador();

comparador1.Velocidad = 10;
comparador2.Velocidad = 0;

Console.WriteLine(comparador1.CompareTo(comparador2));
```

Tanto el objeto **comparador1** como el **comparador2** son del mismo tipo, pero las propiedades **Velocidad** para cada caso contienen valores diferentes: al compararlos, el objeto **comparador1** es mayor a **comparador2**.

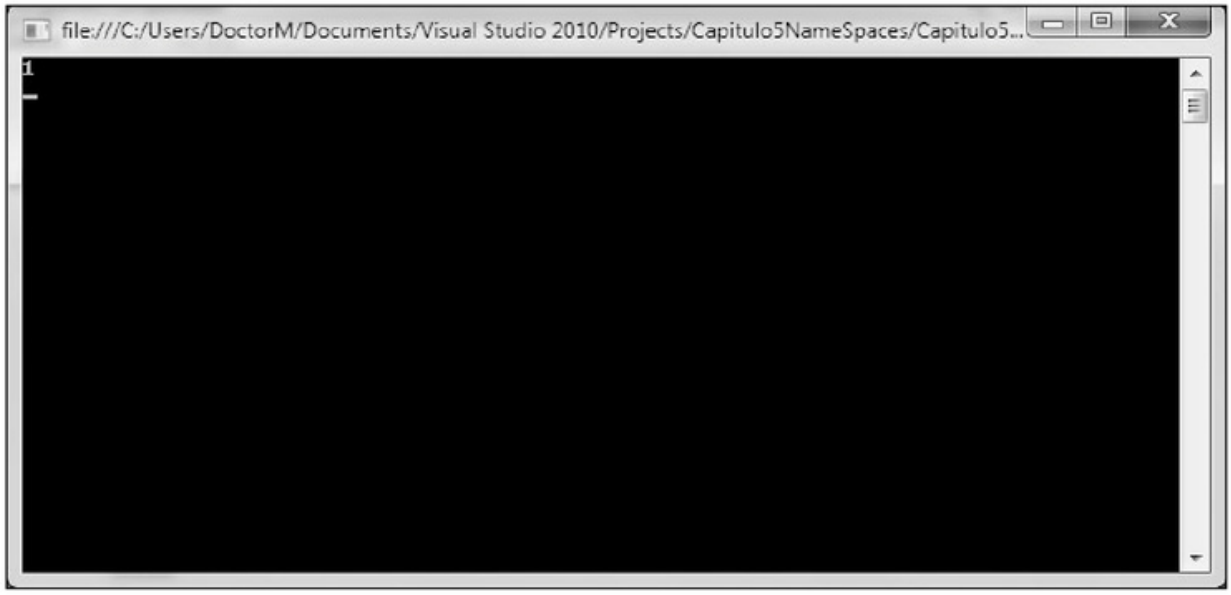


***Figura 9.** El resultado obtenido de la comparación es igual a 1, porque comparador1 es mayor que comparador2.*

Interfaz IEquatable<T>

La tercera interfaz útil es **IEquatable<T>**. Esta interfaz, al igual que **IComparable**, es utilizada para comparar dos objetos, pero en este caso solo para verificar si los objetos dados son iguales entre sí. Además, **IEquatable<T>** posee una nomenclatura diferente de las demás interfaces ya que agrega el descriptor **<T>**. Este

descriptor hace referencia al concepto de **tipos genéricos**. Hablaremos más sobre los tipos genéricos en este mismo capítulo, más adelante. Por ahora, nos limitaremos a utilizar el tipo **<T>**.

```
public class Igualador : IEquatable<Igualador>
{
    public int Valocidad { get; set; }

     //Implementación de IEquatable
    public bool Equals(Igualador other)
    {
        return this.Valocidad == other.Valocidad;
    }
}
```

Al asociar la interfaz a nuestra clase, reemplazamos el descriptor **<T>** por el tipo de objeto que esperamos recibir en la función de comparación. A diferencia de **IComparable** donde obteníamos un objeto y necesitábamos verificar si era del tipo esperado, esta interfaz garantiza que el único tipo de objeto posible para pasar por parámetro sea el que nosotros esperamos. La función **Equals**, además, solo retorna **verdadero** o **falso** y especifica si los objetos dados son o no iguales, basándose en las propiedades que estamos usando para comparar estos objetos. Por supuesto, podremos colocar tantas operaciones como veamos necesario dentro de la función **Equals**, siempre y cuando el resultado de las mismas sea un valor booleano.

```
Igualador igual1 = new Igualador();
Igualador igual2 = new Igualador();

igual1.Valocidad = 10;
igual2.Valocidad = 10;

if (igual1.Equals(igual2))
{
    Console.WriteLine("Los objetos son iguales");
}
else
{
    Console.WriteLine("Los objetos son distintos");
}
```

Con los dos objetos creados y sus propiedades iguales, comparamos el primero con el segundo dentro de la condición. Debido a que ambos contienen el mismo valor, el resultado de la comparación será verdadero.

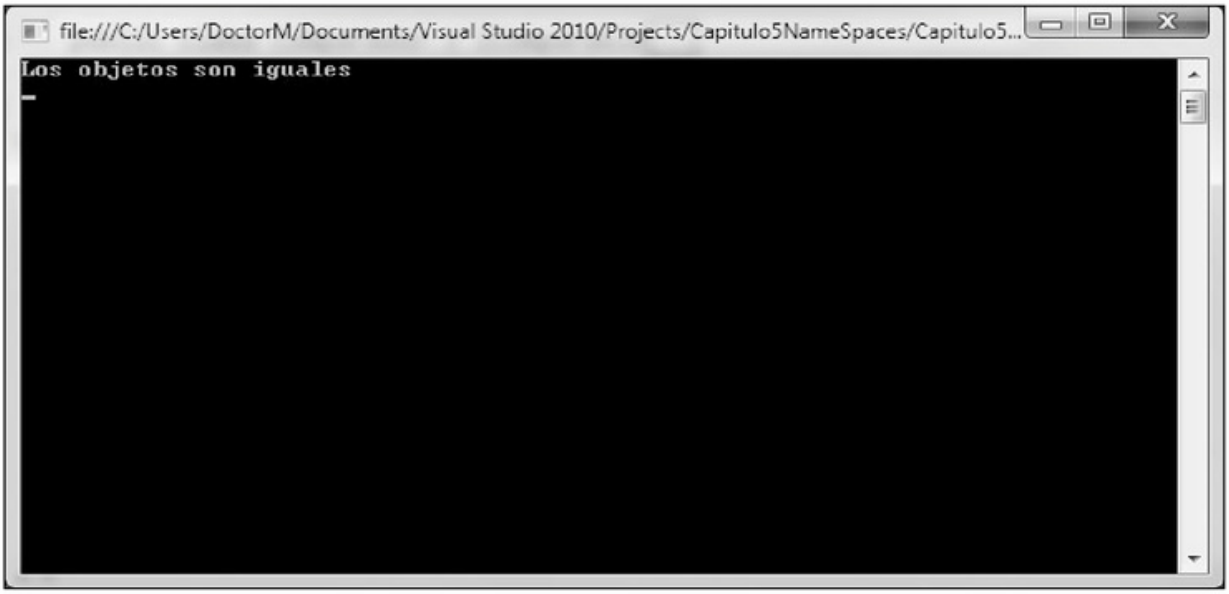


***Figura 10.** Ambos objetos son iguales. Al compararse por medio de la propiedad **Velocidad**, el valor de retorno será igual a **verdadero** y ejecutará la sección verdadera de la condición.*

Sobrecarga de operadores

En los lenguajes orientados a objetos, todos los elementos de este, de una o de otra forma, son objetos o heredan de alguno para adicionar funcionalidad específica. Siguiendo este concepto, en C# los operadores matemáticos y sus funciones, están sujetas también a la especialización e implementación para los diferentes tipos.
Esto quiere decir que el uso del operador **+** para sumar dos números o unir dos cadenas de texto se debe a que este operador implementa funcionalidad para ambos casos. De esta manera, podemos realizar código para operar sobre nuestros propios objetos, algo que no podríamos hacer de forma tradicional. Para entender este concepto, veamos el siguiente código.

III OPERADORES

No siempre será posible que trabajemos con sobrecarga de operadores, ya que algunos operadores de uso común no pueden ser sobrecargados. Los operadores **=** (asignación), **.** (acceso a funciones), **?:** (operador ternario), **new** (constructor), **is** (comparador de tipo), **sizeof** (tamaño), **typeof** (verificador de tipo) no pueden ser sobrecargados.

```
public class Saldo
{
    public int Dinero { get; set; }
}

Saldo saldo1 = new Saldo();
Saldo saldo2 = new Saldo();

...
...

//Arroja error. No se pueden sumar estos objetos
Saldo resultado = saldo1 + saldo2;
```

La clase **Saldo** contiene una propiedad **Dinero** con el monto de una transacción. Si tuviéramos varias transacciones y quisiéramos sumar todas estas sin una función específica para sumar objetos **Saldo**, el operador **+** no sabría cómo hacer la suma debido a que no está especializada en sumar objetos del tipo **Saldo**.

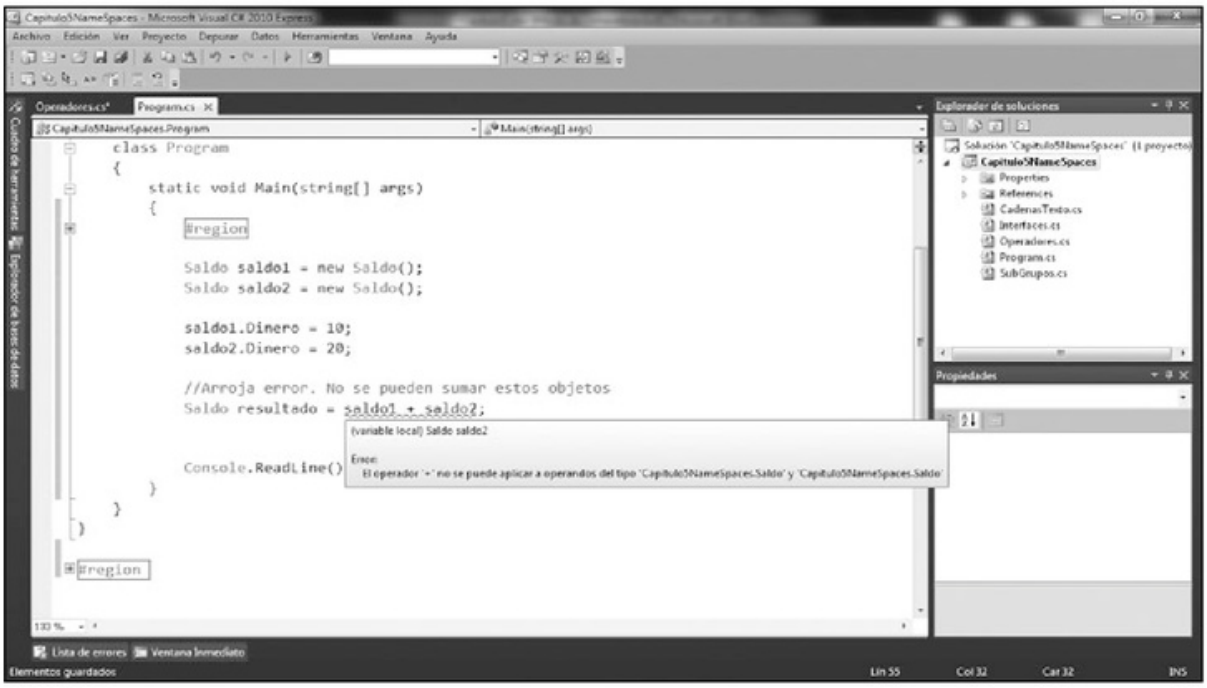


Figura 11. *No contamos con una especialización del operador + para realizar sumas de objetos* ***Saldo****, por lo tanto, obtenemos un error por parte del compilador.*

Si queremos que la suma sea válida, entonces deberemos sobrecargar el operador **+**. La sobrecarga se realiza dentro de la clase con la cual interactuaremos y, además, debe ser declarada mediante el modificador de acceso **static**.

```
public class Saldo
{
     public int Dinero { get; set; }
     public static Saldo operator +(Saldo saldo1, Saldo saldo2)
     {
          Saldo resultado = new Saldo();
          resultado.Dinero = saldo1.Dinero + saldo2.Dinero;
          return resultado;
     }
}
```

Como vemos, además del modificador de acceso, es necesario especificar el tipo de dato que retornará, en este caso un objeto **Saldo**, y usar la palabra reservada **operator** seguida del **operador por sobrecargar**. Los parámetros de entrada en la sobrecarga del operador serán aquellos necesarios para poder realizar la operación; como una suma requiere de dos valores, en este caso dos valores **Saldo**, tanto el primer parámetro como el segundo parámetro representarán los objetos por sumar.

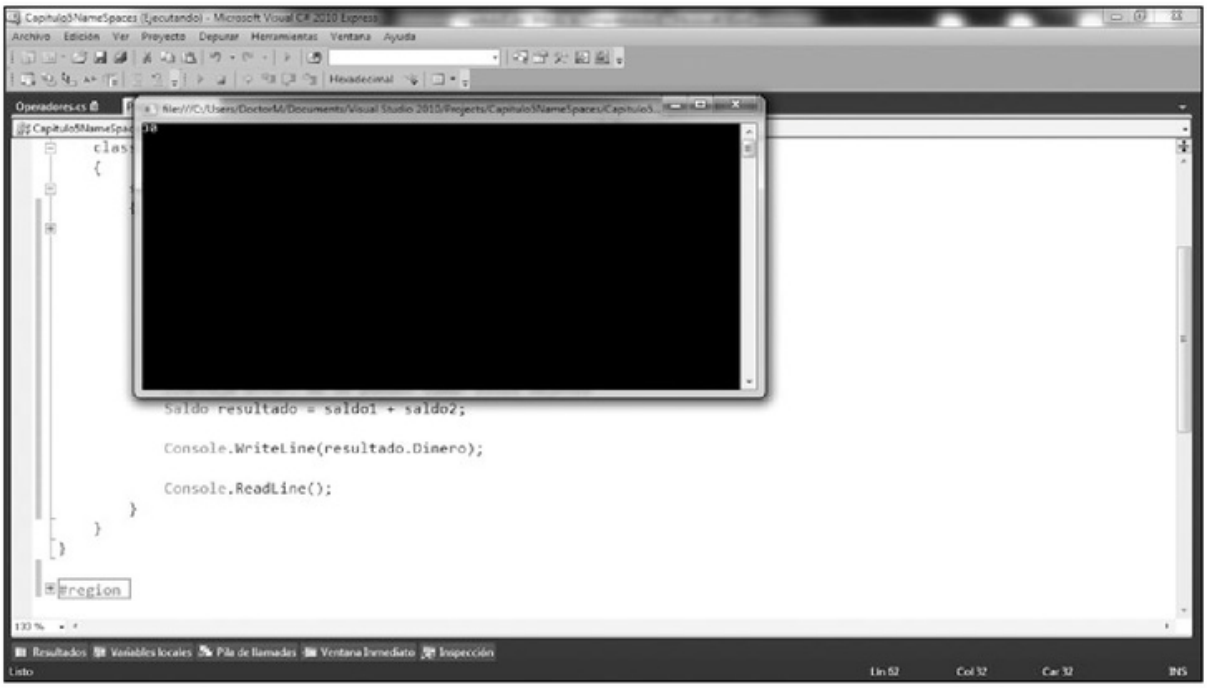


Figura 12. *La suma de los objetos Saldo retorna un nuevo objeto Saldo donde su propiedad es igual a la suma de las propiedades de los dos objetos sumados.*

La sobrecarga de operadores no está sujeta al objeto con el que intentamos interactuar debido a que podemos también utilizarlo para extender los operadores que ya trabajan con los tipos conocidos como **int**, **float**, **double**, y otros, para que además puedan interactuar con nuestros objetos.

```
public static Saldo operator +(Saldo saldo1, int valor)
{
     Saldo resultado = new Saldo();
     resultado.Dinero = saldo1.Dinero + valor;
     return resultado;
}
```

En el código anterior, volvemos a sobrecargar el operador **+**, pero en este caso los parámetros de entrada esperan como valor un tipo **int**. Este es sumado con la propiedad **Dinero** del primer parámetro para retornar un nuevo objeto **Saldo**.

```
Saldo saldo1 = new Saldo();
saldo1.Dinero = 10;
Saldo resultado = saldo1 + 50;
Console.WriteLine("Resultado de la suma con un entero: " +
     resultado.Dinero);
```

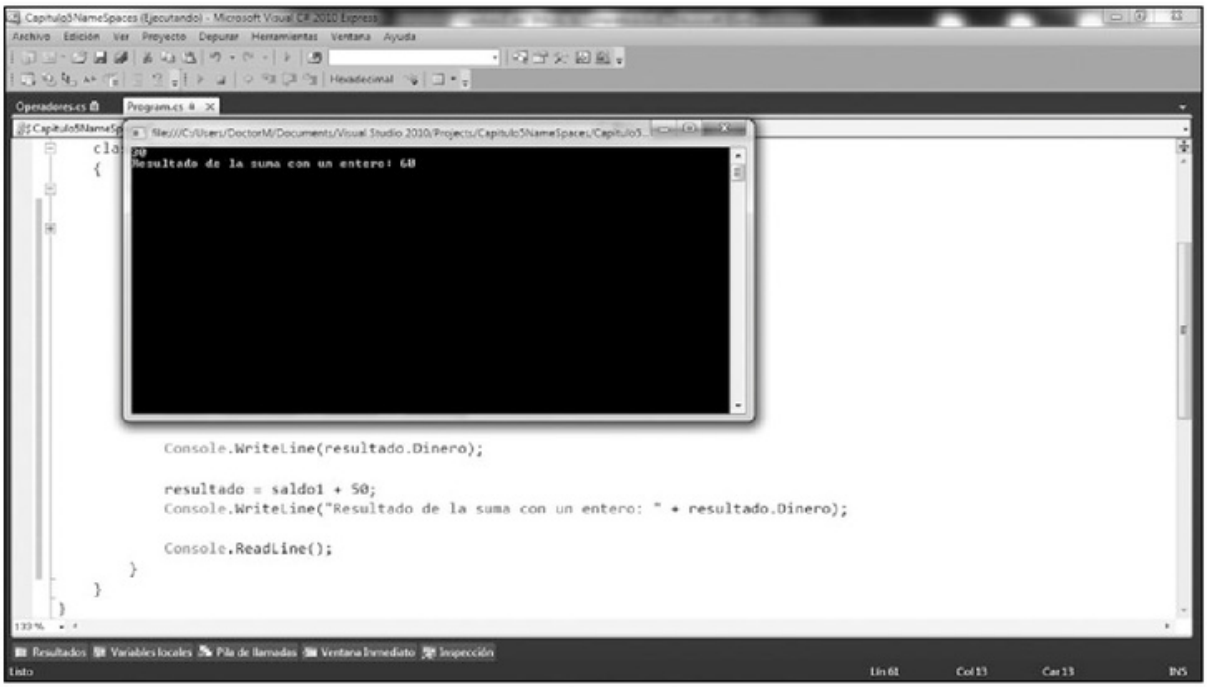


Figura 13. *La sobrecarga de operadores no se limita a operaciones con objetos del mismo tipo.*

Delegados y eventos

En C#, **no tenemos punteros a memoria** debido a que la memoria es manejada por el motor de ejecución del programa. De cualquier forma, contamos con un

elemento similar y que tiene el comportamiento de un **puntero a una función**. Estos tipos son llamados **delegados** y nos permiten mantener una referencia a una función específica para ser ejecutada en cualquier momento.

```
public class Delegados
{
     public delegate void Delegado(int valor);
     ...
     ...
}
```

En el ejemplo, el delegado declarado no posee funcionalidad por sí solo, sino que representa una firma para la función a la cual apuntará. En este caso, la función que asociemos no deberá retornar valor alguno, y recibirá un tipo **int** como parámetro de entrada. Creamos una función con la firma del delegado y otra llamada **ObtenerFuncion** que nos devolverá el puntero a la función apuntando a la primera.

```
private void Funcion(int valor)
{
     Console.WriteLine("Función ejecutada");
}

public Delegado ObtenerFuncion()
{
     Delegado delegado = Funcion;
     return delegado;
}
```

Por lo tanto, cuando llamemos a **ObtenerFuncion** obtendremos una referencia a la función privada de la clase y la posibilidad de ejecutarla.

III OPERADORES

Cuando sobrecargamos operadores, algunos de estos requerirán ser sobrecargados de a pares y de forma opuesta. Por ejemplo, el comparador **==** (igual), al ser sobrecargado, requerirá que su opuesto, **!=** (no igual), también sea sobrecargado. Esto se debe a que ambas operaciones forman parte de un mismo conjunto de operaciones.

```
//Creamos un objeto de la clase Delegados
Delegados punteroAFuncion = new Delegados();

//Obtenemos la función por medio del delegado
Delegados.Delegado funcion = punteroAFuncion.ObtenerFuncion();

//Invocamos la función
funcion.Invoke(10);
```

Mediante el uso de **Invoke** (invocar), podremos ejecutar la función a la cual apunta nuestro delegado. Como la firma de este requiere del paso de un parámetro entero, también deberemos pasar ese valor al momento de intentar ejecutar la función apuntada (vemos esto en la figura que se encuentra a continuación).

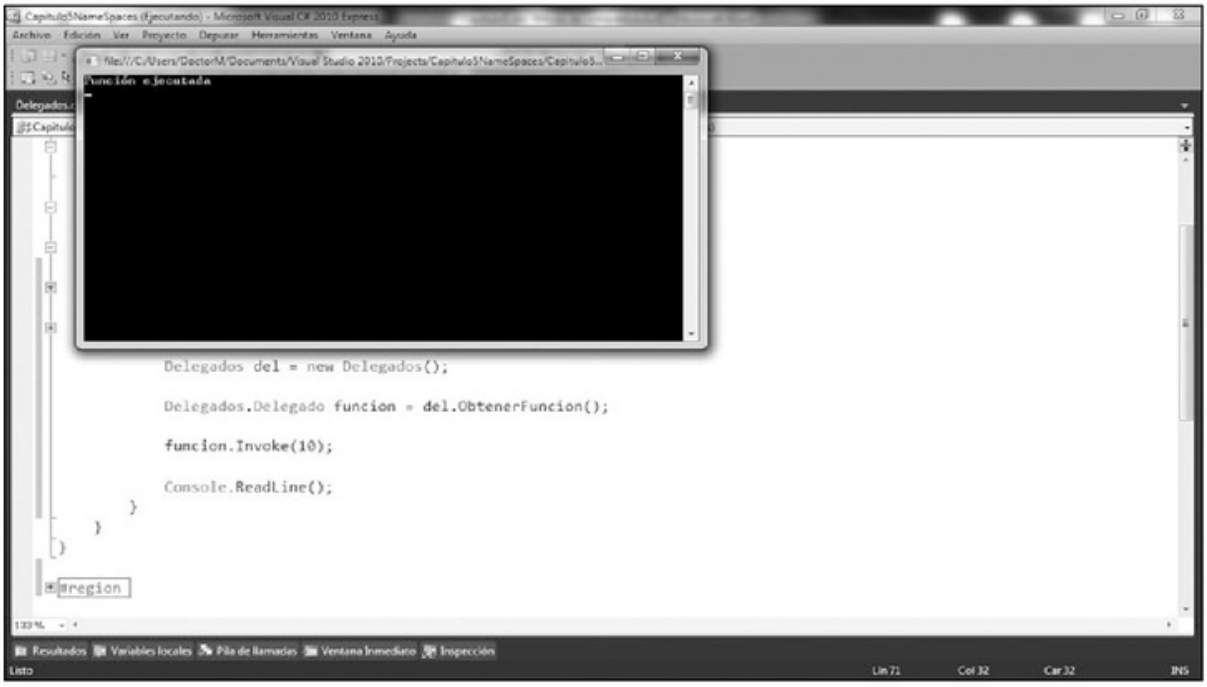


Figura 14. *La ejecución de la función apuntada por el delegado hace que el texto que se escriba mediante* ***Console.WriteLine*** *sea mostrado en la consola, lo que manifiesta la correcta ejecución de la función.*

Pero no solo es posible apuntar a una función existente, sino que, por medio de un delegado, podemos crear funciones que no existen explícitamente en nuestro código. Esto quiere decir que la función no podría ser ejecutada o accedida si no es a través de un delegado. Este tipo de elementos reciben el nombre de **métodos anónimos**.

```
public Delegado ObtenerFuncion()
```

```
{
     Delegado delegado = delegate(int valor)
     {
          Console.WriteLine("Valor enviado: " + valor);
     };

     return delegado;
}
```

Si modificamos la función que nos retorna el delegado, vemos que, en lugar de apuntar a una función específica, se crea una en el momento de la asignación. Esta función conserva esta firma propuesta por el delegado y puede utilizar estos parámetros como parte de su código. Al ejecutar el delegado mediante **Invoke**, vemos que esta función es ejecutada, y el valor enviado se utiliza para escribir el mensaje en la consola.

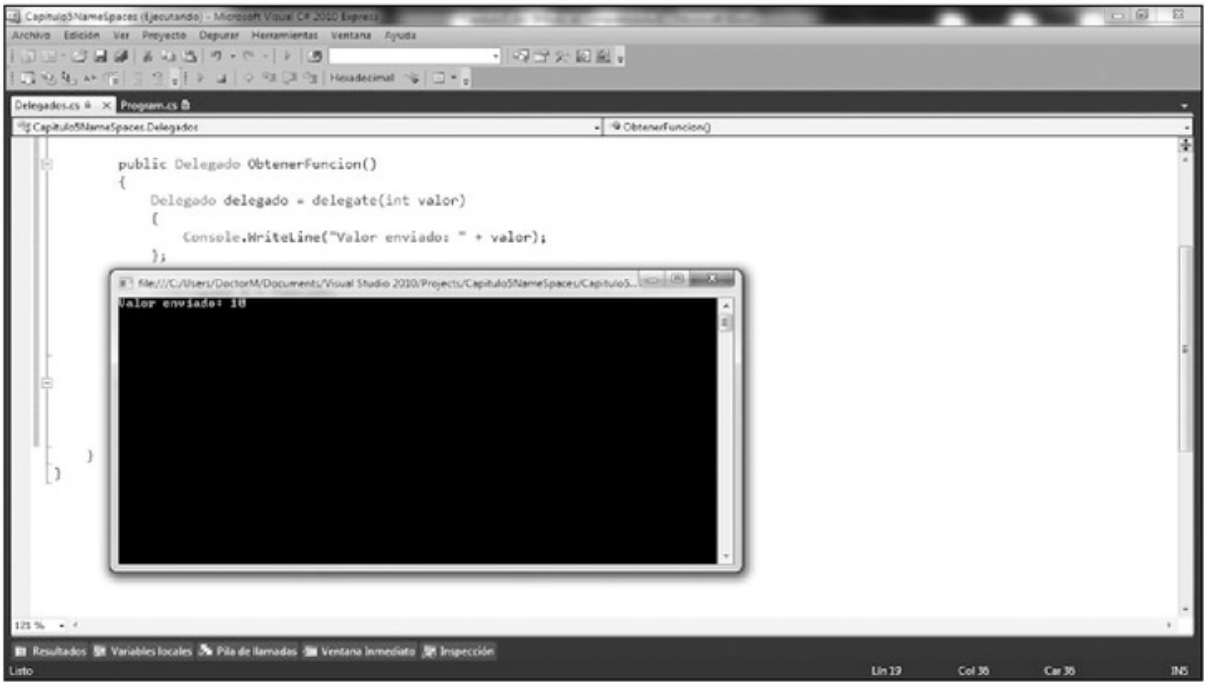


Figura 15. *El método anónimo es referenciado por el delegado y, luego, ejecutado mediante Invoke pasándole el valor entero de 10.*

Junto a los delegados aparece el concepto de **eventos**. Podemos definir los eventos como puntos de ejecución donde un objeto puede avisar a otro que ocurrió algún acontecimiento dentro de él sin estar ligado al flujo de código. Por lo tanto, desde un objeto **A** podríamos crear una instancia de un objeto **B**. Mientras que el código del objeto **A** sigue el flujo de ejecución normal, el objeto **B** podría, en cualquier punto, avisarle a **A** que algo ha ocurrido y ejecutar código en paralelo. Tanto el evento como

la función que recibirá la notificación requieren de un delegado, y es aquí donde se conjugan los dos elementos. Por un lado, una función asociada que espera ser ejecutada, y por otro, un delegado apuntando a ella como camino para su ejecución.

```
public class Eventos
{
     //Declaramos el delegado con la firma
    public delegate void DelegadoEventos(string Mensaje);

     //Declaramos un evento del tipo del delegado
    public event DelegadoEventos Evento;

     ...
     ...
}
```

El evento no posee una firma por sí mismo, sino que requiere del delegado para poseerla. Además, el uso del evento desde la clase que lo contiene requiere de un patrón de implementación común. Debido a que el evento está asociado a una referencia, si nadie se ha asociado a él, este posee una referencia nula, y no podrá ejecutarse, por lo que será necesario verificar la inicialización de la referencia.

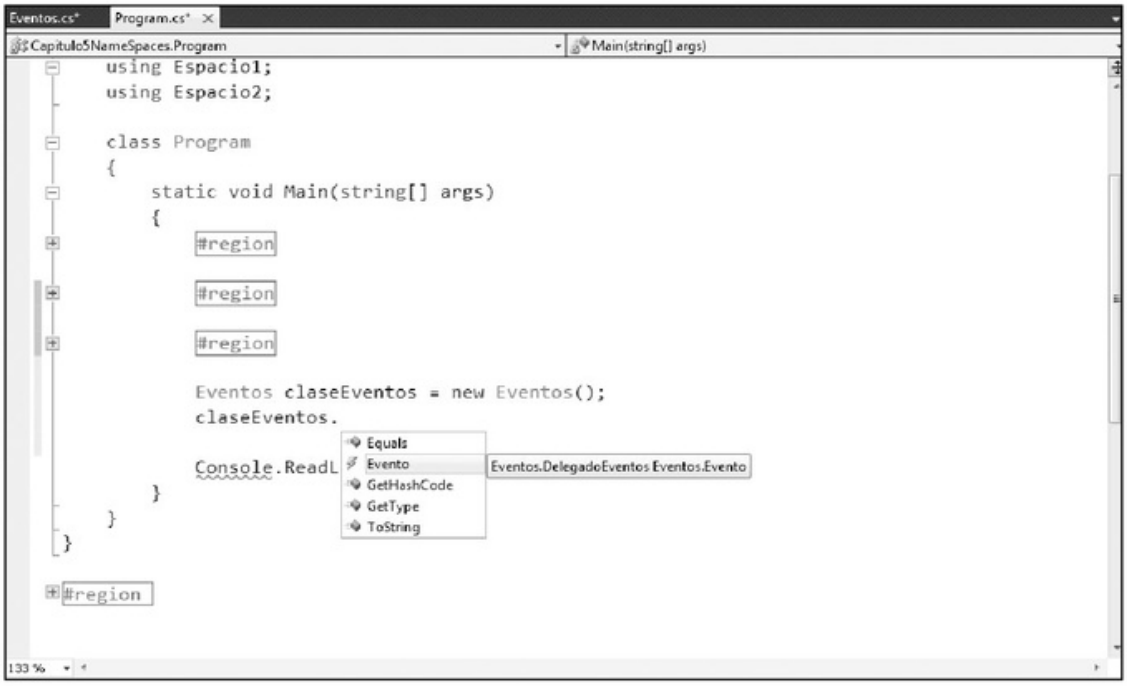


Figura 16. *Todos los eventos de una clase son mostrados con el icono de un rayo. Necesitaremos suscribirnos a estos eventos para obtener las notificaciones de los cambios producidos dentro del objeto.*

```
private void EjecutarEvento()
{
     //Verificamos que exista una referencia
     //asociada al evento antes de ejecutarlo
    if (Evento != null)
    {
        Evento("Evento ejecutado");
    }
}
```

Antes de lanzar un evento, es necesario verificar primero que la referencia exista. Si esta condición se cumple, entonces podremos llamar al evento pasándole los parámetros que hemos especificado en el delegado. Con el patrón de verificación que hemos implementado, nos aseguramos de no llamar al evento.

```
public class CapturaEventos
{
    public CapturaEventos()
    {
        Eventos evento = new Eventos();

            //Asociamos el evento a la función Captura
        evento.Evento += Captura;
    }

    internal void Captura(string Mensaje)
    {
        Console.WriteLine(Mensaje);
    }
}
```

III EVENTOS

Cuando interactuamos con eventos, podemos asociar una función a un evento mediante el uso de **+=** seguido del nombre de la función que recibirá el evento. Si no necesitamos seguir recibiendo notificaciones del evento en cuestión, es posible utilizar el símbolo **-=** para eliminar la referencia del evento hacia nuestra función.

Desde la clase que capturará el evento, asociamos una función interna al evento del objeto mediante el carácter **+=** e implementamos la función que recibirá la lista de parámetros enviados desde el objeto contenedor del evento al momento de que el evento se ejecute. Al ejecutarse el evento del objeto contenedor, la función **Captura**, asociada al evento, es la que reacciona y muestra el mensaje contenido en el parámetro enviado por el evento, gererado por el objeto que lo contiene.

Código genérico

Con las interfaces y las clases base, hemos aprendido cómo romper las dependencias en el código. C# trae una característica para ir un poco más lejos mediante el uso de **genéricos**. Vimos un primer acercamiento en este capítulo cuando aprendimos sobre algunas interfaces útiles, donde podíamos especificar el tipo de objeto por utilizar al momento de interactuar con nuestro código.

```
public class Genericos<T>
{
    public void Mostrar(T parametro)
    {
        Console.WriteLine(parametro.ToString());
    }
}
```

En el ejemplo anterior, en el momento de declarar nuestra clase, utilizamos el símbolo **<T>** para especificar que requeriremos especificar un tipo para **T** cuando debamos crear una instancia de nuestra clase.
Una vez declarado **T**, podemos utilizar este elemento dentro de nuestro código. Para el ejemplo, como un parámetro que deberá ser pasado a la función **Mostrar**, aunque, también podría ser utilizado como una variable dentro de nuestra función, creando una instancia de un objeto o utilizándolo como un tipo.

```
//Declaramos un objeto donde el tipo genérico
//será del tipo int
Genericos<int> generico = new Genericos<int>();
generico.Mostrar(10);

//Declaramos un objeto donde el tipo genérico
//será del tipo string
Genericos<string> generico2 = new Genericos<string>();
generico2.Mostrar("Mensaje");
```

Como vemos en el código anterior, esta clase, al momento de crear una instancia de este objeto define el tipo que tendrá **T**. Para el primer caso, un tipo entero **int**, y para el segundo, un tipo **string**. La función **Mostrar** que hace uso de **T** esperará un dato del tipo definido al momento de su ejecución. Si vemos la **Figura 18**, notaremos que la ayuda contextual nos muestra, al momento de escribir la función, que el parámetro esperado concuerda con el elegido cuando creamos la instancia del objeto.
La ayuda contextual de Visual C# 2010 Express se adapta al tipo utilizado en el momento de la creación de la instancia del objeto. Esto hará que no podamos pasar ningún otro tipo que no sea el pretendido inicialmente.
Al ejecutar ambas funciones, vemos que los valores son escritos en la consola a pesar de que se trate de dos tipos completamente diferentes. Esto se debe a que tanto el tipo **int** como el **string** implementan la función **ToString()**, la cual retorna un texto con el valor contenido dentro de la variable.
Implementar una función genérica puede no resultar fácil; esto se debe a que no todos los tipos posibles pueden implementar aquellas funciones que nosotros estemos necesitando, por lo que requeriremos un mecanismo para restringir, de alguna forma, los tipos que nuestras funciones pudieran necesitar. Para lograr esto, podemos especificar los tipos permitidos mediante el uso de la palabra reservada **where** al momento de declarar la necesidad de un tipo genérico dentro de nuestra clase.

```
public interface IContrato
{
    string Nombre();
}

public class GenericosWhere<T>
    where T : IContrato
{
    public void EscribirNombre(T parametro)
    {

        Console.WriteLine(parametro.Nombre());
    }
}
```

Una solución es crear una interfaz propia a modo de contrato. En este caso, **IContrato** obliga a todos los tipos que utilicemos a que, por lo menos, contengan la función **Nombre**. Luego, en la clase genérica, mediante el uso de **where**, especificamos que el tipo **T** pasado deberá, al menos, implementar nuestra interfaz. Esto nos garantiza que, sin importar el tipo que se le pase, la función **Nombre** se encontrará en él.

```
public class Usuario : IContrato
{
    public string Nombre()
    {
        return "Usuario";
    }
}

public class Administrador : IContrato
{
    public string Nombre()
    {
        return "Administrador";
    }
}
```

Tanto la clase **Usuario** como **Administrador** implementan **IContrato** y, por lo tanto, poseen la función **Nombre**. Por consiguiente, podremos usar estas dos clases como tipos en nuestra clase genérica, como vemo en el código fuente a continuación.

```
GenericosWhere<Usuario> generico1 = new GenericosWhere<Usuario>();
generico1.EscribirNombre(new Usuario());

GenericosWhere<Administrador> generico2 = new
     GenericosWhere<Administrador>();
generico2.EscribirNombre(new Administrador());
```

La función genérica llamará a la función **Nombre** escribiendo lo que cada uno de los objetos retorne desde su propia implementación. Las condiciones **where** para los tipos genéricos no se limitan solo las interfases.

III MÚLTIPLES PARÁMETROS

En todos los ejemplos con genéricos, solemos utilizar un único parámetro **<T>** como tipo de entrada a la clase genérica. De cualquier manera, es posible tener tantos tipos como necesitemos. Para lograr esto, solo deberemos separar con una coma el siguiente tipo de la forma **<T, F, ...>**. Luego, en las funciones, haremos referencia a cada uno de estos tipos mediante las letras elegidas.

Listas genéricas

Una de las principales restricciones de los vectores era que una vez creados no podíamos modificar su tamaño, salvo que creáramos código para copiar los elementos de un vector a otro nuevo con mayor espacio.
Además, buscar elementos dentro de un vector podía resultar engorroso al tener que recorrerlo todo para encontrar el elemento que estábamos necesitando. Una solución es la utilización de **listas genéricas**. Estas listas sirven para contener una sucesión de elementos de un tipo definido, pero con funcionalidades.

```
List<Usuario> usuarios = new List<Usuario>();

Usuario usuario = new Usuario();
usuario.Edad = 20;

//Adicionamos un usuario a la lista
usuarios.Add(usuario);
```

La lista genérica es creada de la misma forma que cualquier tipo genérico, como hemos aprendido en este capítulo, utilizando un tipo para definir el tipo **T** de la lista. Mediante la función **Add** (adicionar), agregamos el objeto **Usuario** a la lista. Las listas genéricas poseen diferentes funciones para interactuar con los elementos.

FUNCIÓN	DESCRIPCIÓN
Add	Adiciona un nuevo elemento a la lista.
AddRange	Adiciona un rango de elementos desde otra colección o lista.
Clear	Elimina todos los elementos de la lista.
Contains	Verifica la existencia de un elemento utilizando otro como parámetro de búsqueda.
Count	Retorna la cantidad de elementos contenidos en la lista.
Insert	Agrega un nuevo elemento basado en un índice.
InsertRange	Agrega un rango de elementos desde otra colección o lista basado en un índice inicial.
Remove	Elimina un elemento de la lista tomando como parámetro de búsqueda otro elemento dado.
RemoveAt	Elimina un elemento de la lista sobre la base de un índice dado.
RemoveRange	Elimina un rango de elementos basado en un índice y un rango numérico.

***Tabla 3.** Principales funciones encontradas en las listas genéricas.*

Junto a este tipo de colecciones de datos, aparece un nuevo concepto para iterar elementos. El **foreach** (para cada uno) es una de las formas más efectivas de obtener todos los elementos de una colección ya que el bucle iterativo recorre automáticamente todos los elementos de la colección de datos retornando, en cada iteración, una referencia al ítem iterado.

```
foreach (Usuario usuario in usuarios)
{
    Console.WriteLine("Edad: " + usuario.Edad);
}
```

En la declaración del **foreach**, colocamos el tipo que retornará cada iteración junto al nombre de variable donde se alojará seguido de la palabra reservada **in** y la lista por iterar. De esta forma, la variable **usuario** contendrá en cada vuelta del bucle la referencia al elemento actual de la lista, y podrá obtener información o asignársela a este elemento. Con frecuencia, las listas genéricas son más usadas que los vectores en el desarrollo de software; asimismo existen otros tipos de listas genéricas que iremos viendo en el transcurso del libro.

... RESUMEN

En este capítulo, hemos podido ver cómo Microsoft .Net y C# abordan la orientación a objetos. Además, hemos podido apreciar cómo es posible mejorar la eficiencia de nuestro código mediante el uso de StringBuilder al trabajar con cadenas de texto, beneficiándonos no solo de la velocidad, sino también del consumo de recursos. De la mano del consumo de recursos, aprendimos algunas tácticas para liberar memoria mediante mecanismos estándares de Microsoft .Net como la implementación de IDisposable. Finalmente, aprendimos una forma para remplazar el uso de vectores por objetos con mayor funcionalidad, como las listas genéricas. Todos estos conceptos nos serán útiles para flexibilizar nuestro desarrollo de software, mejorar la calidad de nuestro código y además poder leer código avanzado. En el próximo capítulo, terminaremos este viaje sobre todo el mundo C# que iniciamos con los conocimientos primordiales de desarrollo y que finalizaremos con los más avanzados.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuántos tipos genéricos podemos utilizar en una clase?

2 ¿Una lista genérica puede albergar objetos de un tipo diferente al declarado?

3 ¿Cuál es el nombre de la función que agrega más texto en un StringBuilder?

4 ¿Para qué sirve la interfaz IEquatable?

5 ¿Qué operadores no pueden ser sobrecargados?

6 ¿Qué tipo debemos crear antes de crear un evento?

7 ¿Qué es un método anónimo?

8 ¿Cuál es el operador utilizado para asociar una función a un evento?

9 ¿Podemos crear un espacio de nombre distinto al creado con la aplicación?

10 ¿Cuál es la diferencia de usar la palabra reservada using dentro de una clase y fuera de ella?

EJERCICIOS PRÁCTICOS

1 Cree una clase genérica que requiera dos tipos genéricos.

2 Intente crear una nueva instancia de esa clase pasándole los dos tipos genéricos.

3 Cree una interfaz común. Una vez creada, agregue algunas clases que implementen esta interfaz. Intente crear una lista genérica que pueda contener una instancia de cada una de estas clases. ¿Qué debería utilizar para definir el tipo genérico de lista?

4 Con un objeto StringBuilder, agregue diferentes cadenas de texto. Luego, intente remover una de ellas.

5 Cree una clase genérica que solo acepte como tipo genérico objetos de los cuales solo se puedan crear nuevas instancias.

Capítulo 6

Microsoft .Net avanzado

En este capítulo, aprenderemos los conceptos más nuevos incluidos en la última versión de Microsoft .Net. Con estos, completaremos el camino que hemos recorrido desde el primer capítulo, aprendiendo los conceptos básicos del desarrollo de software hasta los más avanzados y útiles a la hora de construir aplicaciones con esta tecnología.

MANEJO DE EXCEPCIONES

En cualquier programa de computación, el flujo de ejecución de código no siempre se comporta como esperamos. Es posible que la base de datos a la cual nos estábamos tratando de conectar ya no se encuentre disponible, que el archivo que pretendíamos abrir desde el disco duro de la PC no exista o no tengamos permisos para hacerlo, o simplemente que nos hubiéramos quedado sin memoria para manejar nuestras variables.
Este tipo de comportamiento no esperado dentro de nuestra aplicación hará que esta arroje un error y deje de trabajar, impidiendo su normal funcionamiento. Cuando obtenemos este tipo de comportamiento, decimos que ha sucedido una **excepción no controlada** en nuestra aplicación.

```
int valor1 = 10;
int valor2 = 0;

Console.WriteLine("Resultado de la división: " + (valor1 / valor2));
```

En el código anterior, tomamos dos variables de tipo **int** e intentamos dividirlas. Como podemos notar, la variable **valor2** posee el valor de **0**, por lo que la operación propuesta trataría de dividir el número **10** entre **0**. Al hacerlo, obtendremos un error, ya que no es posible realizar esta división por **0**.

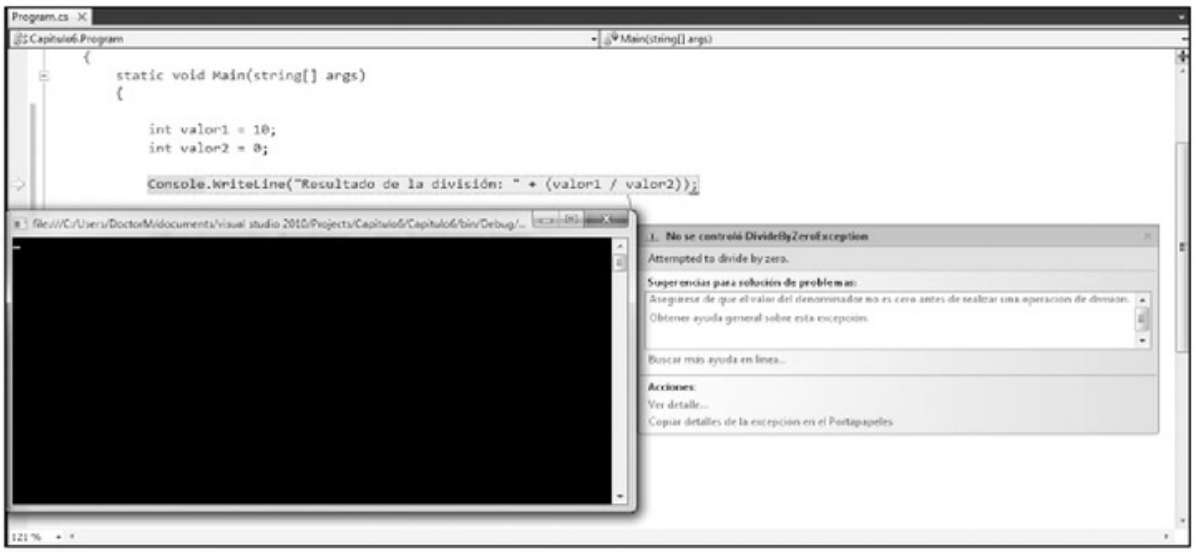


***Figura 1.** Al ejecutarse la operación de división, no se ha realizado ninguna acción preventiva que maneje la excepción. La aplicación arroja esta excepción y deja de funcionar.*

Es necesario que contemos con mecanismos por los cuales podamos capturar y manejar estas excepciones, mediante el uso de palabras reservadas, como **try...catch**.

```
try
{
    Console.WriteLine("Resultado de la división: " + (valor1 / valor2));
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

Con **try...catch**, podemos capturar las excepciones arrojadas por nuestro código e interactuar con ellas. En el ejemplo, la división por cero se sigue realizando, pero dentro del contexto de **try**; si es ejecutado dentro de este contexto fallara, entonces su ejecución secuencial saltará al contexto **catch** que pueda manejar la excepción ocurrida, en este caso, la estructura **catch** representado una excepción general del tipo **Exception**.

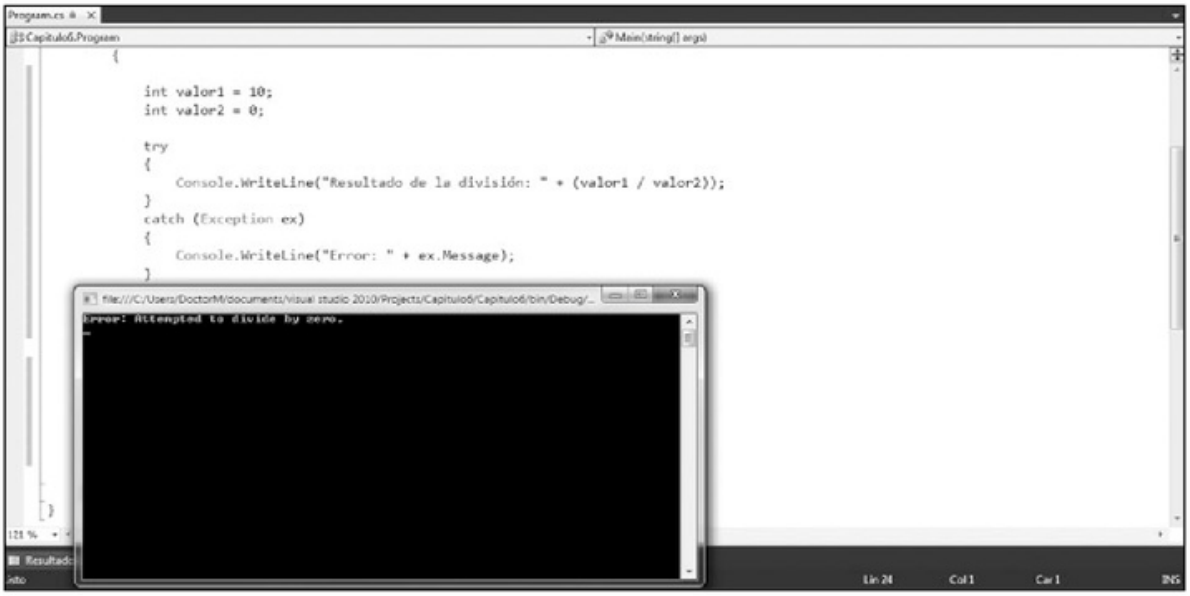


***Figura 2.** Al dividir por cero se produce una excepción. El uso de try...catch previene que nuestra aplicación deje de funcionar, ejecutando el código contenido dentro del contexto catch.*

EXCEPCIONES ANTES DE MICROSOFT .NET

Lenguajes más antiguos tales como **Visual Basic** poseían un manejo de excepciones diferente al planteado por C#. En estos, era posible el uso de instrucciones como **On Error Resume Next** (ante un error, seguir) las cuales simplemente ignoraban cualquier error que se hubiera producido durante la ejecución del código.

Cuando usamos **try...catch** para capturar las excepciones en nuestro código, podemos especializar la captura de estas para que se ejecuten distintas líneas de código sobre la base del error que se ha detectado. En los ejemplos anteriores, hemos utilizado una excepción genérica, esto quiere decir que, sin importar el tipo de excepción arrojada por el código, siempre será capturada por ella. Asimismo, es posible especializar la captura sobre la base del tipo de error que la ejecución de nuestro código arroje.

```
try
{
    Console.WriteLine("Resultado de la división: " + (valor1 / valor2));
}
catch (ArithmeticException arEx)
{
    Console.WriteLine("Error aritmético: " + arEx.Message);
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

En el código anterior, hemos agregado un nuevo grupo **catch** (agarrar), pero en este caso esperando una excepción del tipo **ArithmeticException** (excepción aritmética). Como la división por cero es una excepción de este tipo, el código ingresa por este grupo y no, por la excepción más genérica escrita más abajo.

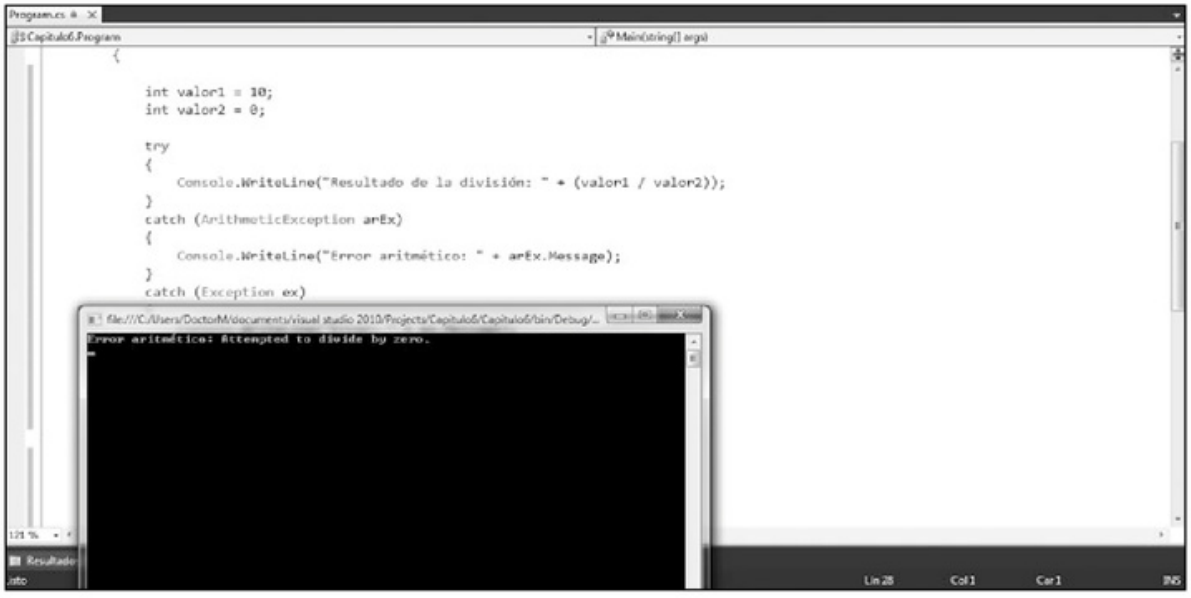


***Figura 3.** Con una excepción más específica al error arrojado por la operación, el error es manejado ahora por el grupo **catch** especializado.*

Muchas veces, necesitaremos realizar otras tareas después de manejar una excepción. Supongamos que hemos abierto una conexión a una base de datos antes de ejecutarle una consulta y, al momento de ejecutar la consulta, recibimos un error que capturamos y manejamos. Pero, luego de esto, necesitamos, por ejemplo, realizar algunas acciones para limpiar la memoria o bien desconectarnos de la base de datos sin importar si hubo o no un error. Para eso, el conjunto **try...catch** agrega otro grupo, el grupo **finally** (finalmente), dándonos la posibilidad de ejecutar código después de haber capturado o no una excepción. Este bloque será de gran utilidad, en especial, para destruir todos los objetos que pudiéramos haber utilizado dentro del bloque **try**.

```
try
{
    Console.WriteLine("Resultado de la división: " + (valor1 / valor2));
}
catch (ArithmeticException arEx)
{
    Console.WriteLine("Error aritmético: " + arEx.Message);
}
catch (Exception ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
finally
{
    Console.WriteLine("Finally ejecutado");
}
```

La estructura **finally** se ejecutará siempre. Si hubiese una excepción, esta primero sería capturada por alguno de los grupos **catch** declarados y, luego, al finalizar, la estructura **finally** ejecutaría el código contenido en ella. En este caso, al ejecutrse **finally** se escribirá el mensaje **Finally** ejecutado en la consola.

III ORDEN EN LAS EXCEPCIONES

Cuando armamos la estructura de los bloques **try...catch**, es necesario tener en cuenta el orden de los bloques **catch**. Al igual que la estructura **switch**, si colocamos al principio el bloque **default**, o sea, el más general primero, los demás bloques serán ignorados. En el caso de los **catch**, siempre deberemos colocar los más específicos primero y solo al final el más general.

***Figura 4.** El error de división por cero es capturado por la excepción ArithmeticException y muestra el mensaje que contiene. Luego, el bloque finally es ejecutado y muestra el mensaje correspondiente.*

Existe una gran cantidad de excepciones dentro de Microsoft .Net. En la **Tabla 1**, podemos ver una lista resumida de las más comunes.

TIPO DE EXCEPCIÓN	DESCRIPCIÓN
ArgumentException	Un parámetro enviado a una función es incorrecto.
ArgumentNullException	Se dispara cuando un parámetro nulo es pasado a una función que no lo permitía.
IndexOutOfRangeException	Se dispara cuando se quiere acceder a un índice inexistente de un vector.
ArithmeticException	Errores que pudieran ocurrir durante operaciones aritméticas.
Exception	Excepción genérica. Sirve para capturar cualquier tipo de excepción.
NullReferenceException	Es arrojada cuando se intenta acceder a un objeto nulo.
OutOfMemoryException	Esta excepción es arrojada cuando la creación de un nuevo objeto mediante el uso de new falla.
StackOverflowException	Se dispara cuando la pila de ejecución posee demasiadas llamadas que no pueden ser resueltas. Un caso típico es el mal uso de llamadas recursivas.

***Tabla 1.** Vemos una lista de algunas de las excepciones más comunes dentro de nuestra aplicación. Es recomendable, si usamos más de un catch, dejar la excepción más genérica al final de ella.*

CAPÍTULOS ONLINE

En la página oficial de **RedUsers**, podemos encontrar los apéndices de este libro, de manera online,: **Apéndice A: Programación alternativa** y **Apéndice B: Nuevas tecnologías en Programación.net**. Los descargamos desde la dirección web siguiente: **www.redusers.com**.

Así como la ejecución de nuestro código puede lanzar excepciones, es posible que nosotros lancemos excepciones para avisar al desarrollador que existe alguna falla imprevista y no manejada. Estos nos resultará de gran utilidad, en casos, donde por reglas propias del desarrollo que estemos realizando, no podamos seguir con la ejecución normal del código.

```
public class ArgumentosNulos
{
     ...
     ...

     public void Argumentos(object objeto)
     {
            //Verificamos que el parámetro
            //no sea nulo
            if (objeto == null)
                  throw new ArgumentNullException("El objeto no puede ser
                  nulo");
     }
}
```

Como vemos en el código anterior, en el momento en que la función **Argumentos** sea llamada, esta verifica que el parámetro enviado no sea igual a nulo. De serlo, crea una nueva excepción y la lanza mediante el uso de la palabra reservada **throw** (lanzar). Una vez lanzada la excepción podremos, desde el código que llamó a la función, capturarla con los mecanismos analizados previamente y actuar de forma acorde.

Métodos extendidos

Otra característica introducida en las últimas versiones de C# son los **métodos extendidos**. Estos métodos nos dan la posibilidad de agregar funcionalidad a objetos ya existentes sin tener que modificar sus códigos fuente. Aunque debemos hacer una distinción en este punto ya que el objeto en sí no es modificado, sino que la nueva función creada aparecerá en la lista de propiedades y funciones de la ayuda contextual como una función más, pero seguirá actuando como una función externa al objeto.

```
public class Matematica
{
     public int Valor1 { get; set; }
```

```
        public int Valor2 { get; set; }

        public int Sumar()
        {
            return Valor1 + Valor2;
        }
    }
```

El código anterior muestra un simple objeto con una función y dos propiedades. La función retorna el valor resultante de la suma de las dos propiedades. Ahora, imaginemos que no tenemos acceso o la posibilidad de modificar este objeto, pero necesitamos que todos los desarrolladores de nuestra aplicación puedan, además de sumar, restar los dos valores guardados en las propiedades, aunque sin tener que crear o llamar a una función independiente, sino a una que exista dentro del mismo objeto, o simule estarlo.

Figura 5. *En la ayuda contextual, podemos ver que existe una función* ***Restar****, aunque esta no existe dentro del objeto creado.*

EXCEPCIONES ESCONDIDAS

Cuando depuramos errores, muchas veces obtendremos algunos de estos sin que hubiéramos tenido la oportunidad de manejarlo; a pesar de esto, el entorno de desarrollo nos avisará deteniendo la ejecución del código. Si necesitamos información sobre los posibles errores, en la ventana inmediata podremos escribir **$exception** para obtener un detalle del error detectado.

Para poder extender un objeto mediante una nueva función, deberemos crearla dentro de una clase estática, declarando la función también como estática.

```
public static class MetodosExtendidos
{
    public static int Restar(this Matematica objeto)

    {
        return objeto.Valor1 - objeto.Valor2;
    }
}
```

En el código, podemos ver cómo la función **Restar** no posee ninguna diferencia respecto de cualquier otra clase, salvo por el primer parámetro utilizado para esta. El operador **this** seguido del tipo de objeto que queremos extender es el que identifica a qué objetos aplicará esta función. De esta forma, estamos especificando que la función **Restar** aplica al objeto de tipo **Matematica**. Una vez dentro de la función, podremos hacer uso de las propiedades del objeto, donde este será una referencia del objeto con el cual estemos interactuando. Esto quiere decir que los valores contenidos en sus propiedades serán los que se hubieran definido inicialmente en el objeto que intenta ejecutar el método extendido. Cualquier objeto del tipo **Matematica** contendrá la extensión **Restar** que al ser llamada pasará este objeto como parámetro para su manipulación.

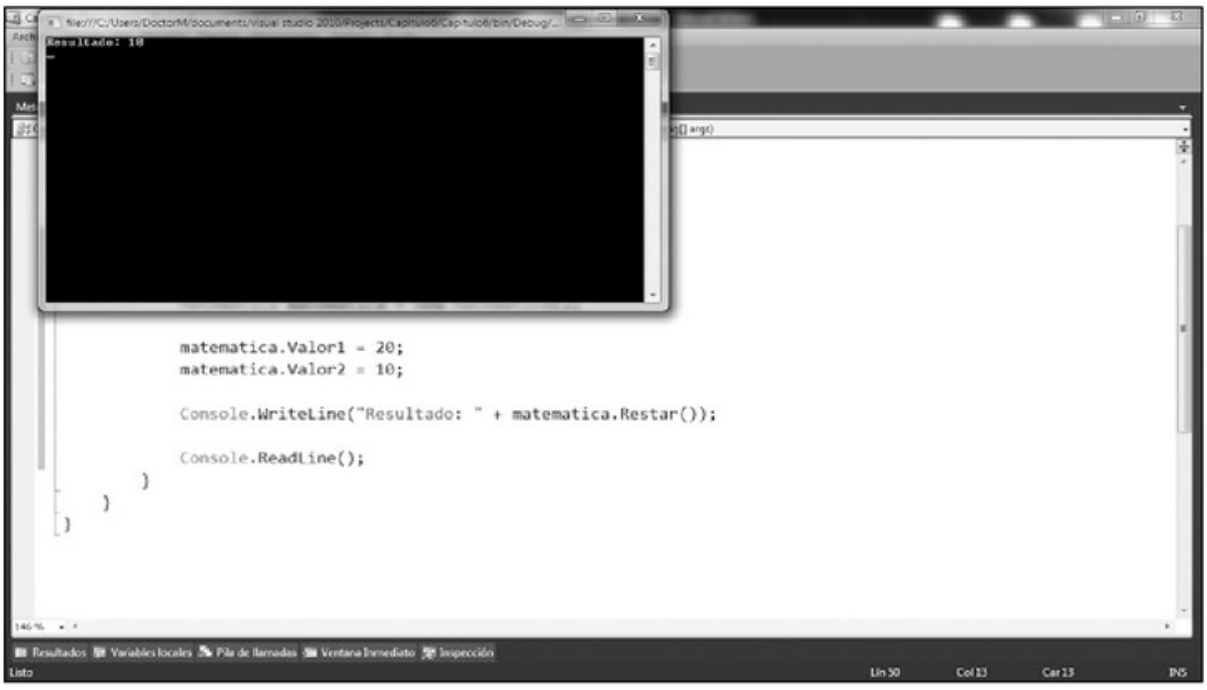


Figura 6. *La ejecución del método extendido del objeto Matematica se ejecuta correctamente como una función del objeto y no una función externa a él.*

MANIPULAR INFORMACIÓN CON LINQ

LinQ (*Language integrated query* o, en español, lenguaje integrado de consultas) fue introducido en la versión 3.5 de Microsoft .Net Framework como una evolución en el lenguaje de programación para unir el mundo de las entidades relacionales, como las bases de datos y los lenguajes orientados a objetos como C#, así como para brindar una plataforma para la realización de operaciones lógicas similares a las encontradas en las consultas SQL.

```
public void Ordenar()
{
     int[] elementos = new int[5] { 10, 23, 1, 6, 0 };
     ...
     ...
}
```

Observemos el código que se encuentra anteriormente y pensemos cuál podría ser la forma más eficiente de ordenar sus elementos de manera ascendente. Es posible que debamos escribir nuestro propio algoritmo de ordenación, consumiendo tiempo valioso y arriesgándonos a que no sea lo bastante eficiente y ágil; en todo caso, podríamos usar LinQ para conseguir esto de forma rápida.

```
var consulta = from e in elementos
                  orderby e ascending
                  select e;
```

Notemos la gran similitud con una consulta SQL. En este caso, definimos la consulta donde **e** está contenida en el vector **elementos**; ordenaremos **e** de forma **ascendente** para seleccionar, por último, el conjunto **e**. Esta consulta es guardada en la variable **consulta** la cual será del tipo **var**, lo que quiere decir que será del tipo

{} HABLANDO CON LA BASE DE DATOS

Las bases de datos poseen un lenguaje propio, lenguaje que difiere del propuesto por los lenguajes orientados a objetos. Es necesario, por lo tanto, encontrar un nexo entre el modelo relacional de la base de datos y el modelo de objetos del lenguaje de programación. LinQ abstrae al desarrollador de la capa de consultas SQL, y muestra solo los objetos utilizados por el código C#.

que retorne la consulta: para este caso, un nuevo vector de enteros. Podemos ver el resultado de la iteración de los elementos ordenados en la **Figura 7**. El tipo de **var** será analizado en el momento de la compilación del código.

***Figura 7.** Al iterar la consulta mediante un bucle, podemos comprobar que, en efecto, el vector ha sido ordenado y los datos son impresos en el orden esperado.*

LinQ no solo es útil para casos de ordenamiento. Debido a que presenta un conjunto completo de elementos sintácticos para la realización de consultas, es posible colocar condiciones en una selección para obtener solo aquellos elementos que cumplen con la expresión booleana dada.

```
var consulta = from e in elementos
               where e >= 10
               orderby e ascending
               select e;
```

III FALLAS CONTROLADAS

Cuando se llama a una función que pudiera fallar, una práctica no recomendada es que esta retorne un valor **booleano** para avisarnos del error. Al hacerlo perderemos el contexto de la falla, por lo tanto, es preferible arrojar una nueva excepción con mayor información. El código que ha llamado a esta función se encargará de controlarla y actuar de la mejor forma.

Con el mismo conjunto de elementos, realizamos una consulta similar, aunque agregamos una condición **where** y esperamos que solo se incluyan en el conjunto de elementos seleccionados aquellos que sean **iguales o mayores** a **10**. Al iterar el resultado, vemos que solo los que cumplen con la condición son mostrados.

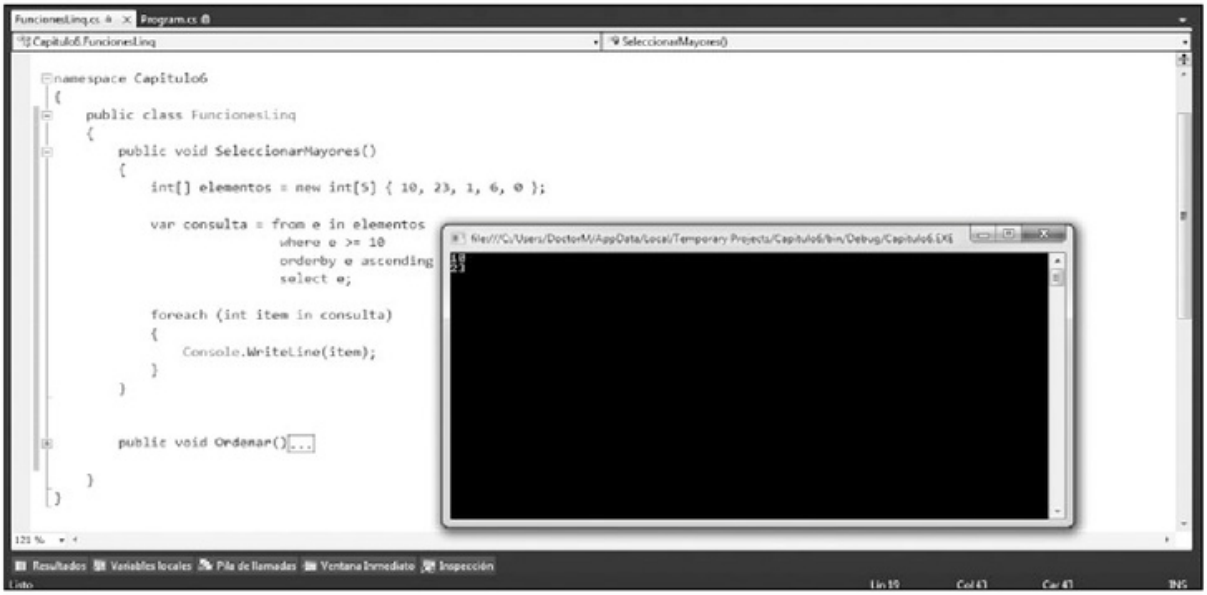


***Figura 8.** La condición where no funciona solo con una expresión booleana ya que es posible agregar más elementos a la condición mediante el uso de && y ||.*

Las consultas que podemos crear con LinQ no se limitan a tipos simples; es posible realizar este tipo de consultas sobre colecciones de objetos propios.

```
public class Usuario
{
    public enum TipoUsuario
    {
        Administrador,
        Usuario,
        Invitado
    }

    public TipoUsuario TipoDeUsuario { get; set; }

    public string Nombre { get; set; }

    public string Departamento { get; set; }
}
```

```
List<Usuario> usuarios = new List<Usuario>();

usuarios.Add(new Usuario()
    {
            Departamento = "Investigación",
            TipoDeUsuario = Usuario.TipoUsuario.Administrador,
            Nombre = "Usuario 1"
    });

usuarios.Add(new Usuario()
{
    Departamento = "Investigación",
    TipoDeUsuario = Usuario.TipoUsuario.Invitado,
    Nombre = "Usuario 2"
});

...
...
```

La clase **Usuario** define los tipos de usuarios que podremos tener dentro de nuestra aplicación. Luego, con este modelo, cargamos una lista genérica de usuarios con diferentes valores para sus propiedades. Estos datos podrían provenir desde una base de datos o cualquier otra fuente de datos, en todo caso, con la lista cargada, podremos realizar diferentes operaciones sobre ella. Seleccionemos todos los usuarios que pertenecen al departamento de investigaciones y además son del tipo **Usuario**.

```
var consulta = from u in usuarios
               where u.TipoDeUsuario == Usuario.TipoUsuario.Usuario
               && u.Departamento.Equals("Investigación")
               select u;
```

III TIPOS DE LINQ

LinQ no es solo para trabajar con objetos y colecciones de ellos. Dentro de la gama de LinQ, encontramos también especializaciones para acceder y manipular texto en formato XML, así como consultas a bases de datos. Con esto, LinQ cubre toda la gama de paradigmas para el almacenaje y manipulación de datos desde nuestras aplicaciones.

```
foreach (Usuario item in consulta)
{
    Console.WriteLine("Nombre: " + item.Nombre +
        ", Departamento: " + item.Departamento);
}
```

En el código anterior, vemos cómo la consulta incorpora la comparación por un tipo básico mediante la propiedad **Departamento** y, al mismo tiempo, incluye la restricción de la búsqueda de la enumeración que especifica qué tipo de usuario es. Al iterar el resultado, vemos que solo se muestran los usuarios correspondientes al filtro aplicado en la consulta, además que el objeto resultante de la consulta incorpora todas las propiedades que tenía el objeto **usuario**.

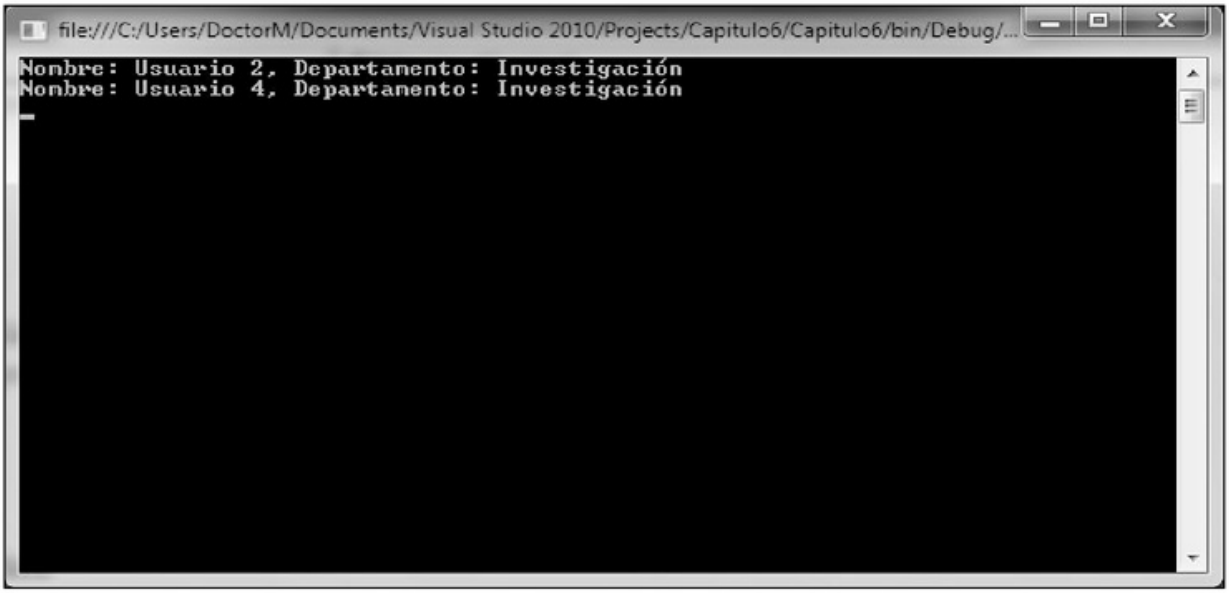


Figura 9. *Una consulta LinQ puede aplicarse a colecciones de tipos más complejos. Si creamos una lista de tipo Usuario, filtramos el contenido sobre la base de sus propiedades, incluida una enumeración.*

Siempre hemos seleccionado el objeto resultante de la consulta en su totalidad. Esto quiere decir que la instrucción **select** seguida del objeto declarado en el **from** hace que el elemento contenido dentro de la lista de objetos se seleccione con todas sus propiedades, pero muchas veces solo necesitaremos algunas de esas propiedades. Esto es posible realizarlo en el momento de la selección de la consulta LinQ.

```
var consulta = from u in usuarios
               where u.TipoDeUsuario == Usuario.TipoUsuario.Usuario
               && u.Departamento.Equals("Investigación")
```

```
select new { NombreUsuario = u.Nombre, Dpto =
  u.Departamento };
```

En este código, hemos utilizado esta consulta para traer los distintos usuarios que concuerden con el filtro aplicado, pero, en el momento de la selección, en vez de seleccionar **u** en su totalidad, definimos un nuevo tipo. Este nuevo tipo será un tipo anónimo, similar a los métodos anónimos vistos en el **Capítulo 5**. Al redefinir el objeto seleccionado, la ayuda contextual mostrada cuando escribimos código nos advierte de las propiedades encontradas dentro de este tipo anónimo (**Figura 10**).

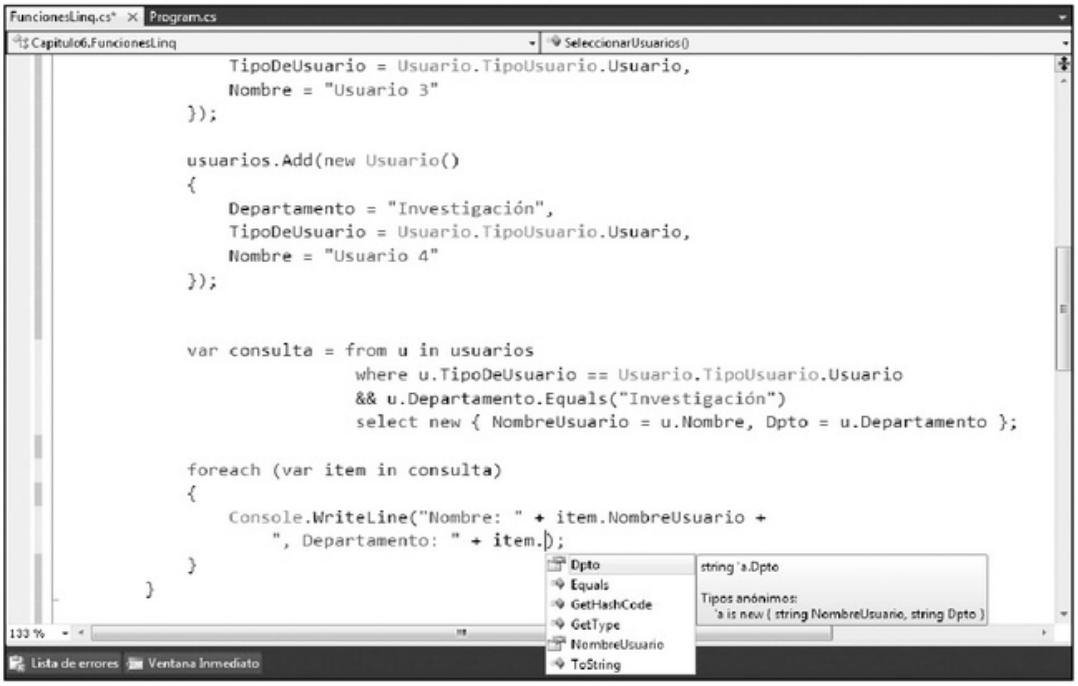


***Figura 10.** La ayuda contextual nos advierte de las propiedades encontradas en el tipo anónimo. Este es un nuevo tipo definido en el momento de la creación de la consulta.*

Además de la forma en la cual hemos aprendido a escribir consultas con LinQ, similares a las consultas SQL, podemos escribir consultas de forma que pueda resultar más armónico en el concepto tradicional de desarrollo de código. Tanto las colecciones,

ALGORITMOS DE ORDENACIÓN

Cuando necesitamos ordenar objetos en lenguajes de programación con menos prestaciones, es necesario que recurramos a algoritmos conocidos para ordenar elementos. Algoritmos como el de **ordenación por burbuja** resultan útiles para cuando no tenemos un modelo que respalde nuestro desarrollo. LinQ resultará uno de los caminos más fáciles para solucionar esto.

como las listas y vectores, poseen, por parte del lenguaje, funciones extendidas para realizar las mismas operaciones que pudiéramos escribir de la forma que ya hemos visto. Veamos cómo seleccionar los elementos de un vector de enteros, que sean mayores o iguales a diez, de forma idéntica a la que hicimos al principio de este apartado.

```
int[] elementos = new int[5] { 10, 23, 1, 6, 0 };

var consulta = elementos.Where(u => u >= 10).OrderByDescending(u => u);
```

El vector **elementos** posee una extensión llamada **Where** así como **OrderByDescending**. Tanto **Where** como **OrderByDescending** esperan para funcionar en un conjunto de condiciones basadas en el objeto por iterar. En este caso, **u** representa cada uno de los elementos del vector, por lo tanto, la expresión **u => u >= 10** se ejecutará tantas veces como elementos tenga el vector. Debemos leer la expresión de la siguiente forma: siendo **u** un parámetro de entrada, ejecutar (**=>**) la extensión **Where** para **u** mientras que **u** sea **mayor o igual** que (**>=**) **10**. Luego, sobre los resultados arrojados por la extensión **Where**, ejecutamos **OrderByDescending** mediante el mismo patrón: siendo **u** un parámetro de entrada, ejecutar la extensión utilizando el valor de **u**.

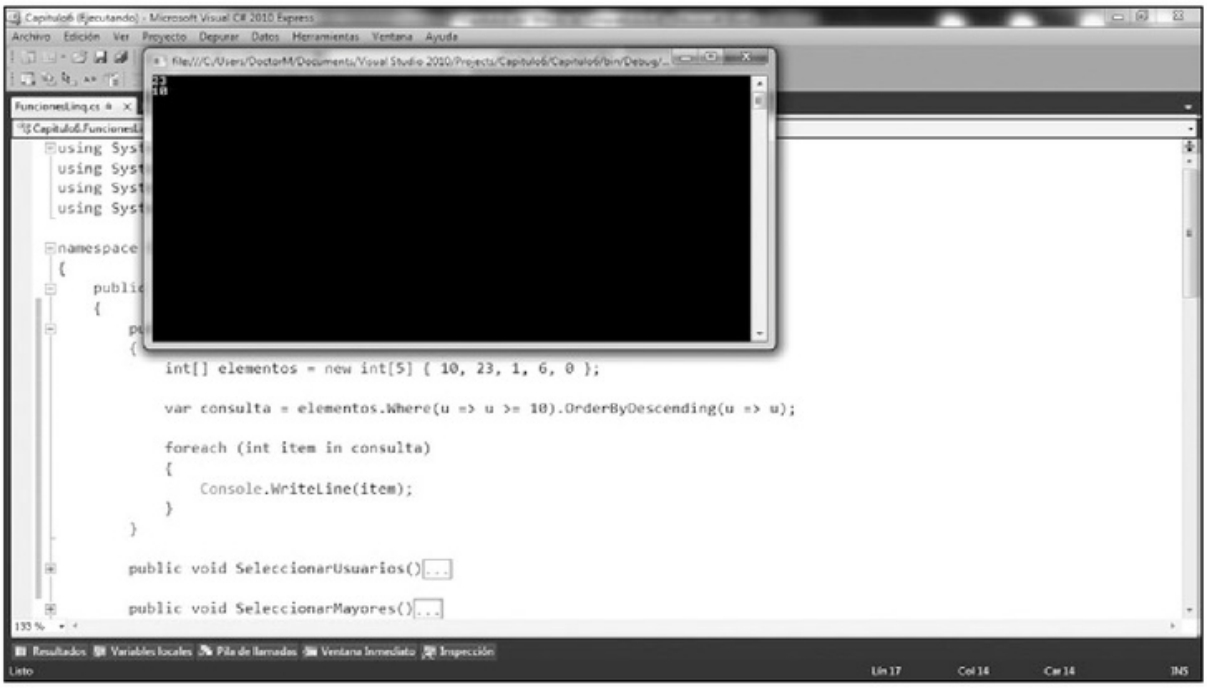


Figura 11. *El uso de las funciones extendidas sobre el vector de enteros arroja el mismo resultado que el arrojado por la consulta creada en LinQ.*

El uso de esta forma de escribir una consulta o la forma sintáctica propuesta por LinQ dependerá, principalmente, de la comodidad que sintamos al momento de escribirlas. Si bien el objetivo principal de LinQ es la selección de elementos, existen muchas

formas de hacerlo; en la **Tabla 2**, que se encuentra a continuación, podemos ver una lista de algunas de las funciones contenidas en LinQ.

MÉTODO	DESCRIPCIÓN
SelectMany	Selector que permite realizar una consulta anidada con más de una fuente de datos, donde las distintas fuentes presentan una relación entre ellas.
Take	Especifica la cantidad de elementos de una consulta específica por retornar.
Skip	Descarta los elementos de la consulta sobre la base del número indicado. Con base 0, hace que el número especificado no sea incluido en los resultados de la consulta.
ToArray	Transforma el resultado de la consulta a un vector de elementos del tipo especificado.
ToList	Transforma el resultado de la consulta a una lista de elementos del tipo especificado.
First	Toma el primer elemento de la colección retornada por la consulta.
FirstOrDefault	Toma el primer elemento de la colección retornada por la consulta o el valor por defecto que estos elementos pudieran tener si no se encontró ningún elemento para retornar.
Any	Dada una condición, verifica si alguno de los elementos de la colección concuerda con la expresión dada.
Count	Cuenta y retorna el número total de registros encontrados en la consulta realizada.

***Tabla 2.** LinQ posee diferentes funciones para imitar el comportamiento de las consultas SQL dentro del código C#. Aquí se presentan las principales funciones disponibles a la hora de trabajar con colecciones de datos.*

PROGRAMACIÓN DE HILOS

Si ya tenemos algo de experiencia en el desarrollo de software, o hemos estado realizando algunos de los ejemplos de este libro, podremos haber notado que las aplicaciones suelen bloquearse mientras están ejecutando las líneas de código, en especial cuando creamos bucles que iteran grandes cantidades de elementos; solemos ver este comportamiento en forma más clara en caso de desarrollos con interfaz gráfica, que deja la interfaz inutilizada hasta que el bucle termina de ejecutarse.
Esto se debe a que nuestras aplicaciones se ejecutan bajo el concepto de **hilos de ejecución**. Estos hilos son ramificaciones que crea el sistema operativo o la capa

{} ITERACIÓN CON LINQ

Una cualidad interesante de LinQ es el momento en que se ejecuta la consulta. A diferencia de lo que pudiéramos creer, los resultados de una consulta LinQ solo serán ejecutados y obtenidos en el momento en el cual estos sean iterados. Este caso especial se aplica cuando utilizamos LinQ para consultar elementos de una base de datos.

superior de ejecución que tenga control sobre nuestra aplicación para poder brindarle un contexto de ejecución, así como tiempo a los procesos internos para que ejecuten parte de ellos y dejen espacio de ejecución a otros procesos de esta aplicación.
Pensemos que, mientras nosotros estamos ejecutando cualquier aplicación dentro de la PC, existen otros programas que también se están ejecutando al mismo tiempo; así como podríamos estar creando algunas líneas de código en Visual C# 2010 Express, también podríamos tener abierto un **navegador web**, o un **procesador de texto**, o incluso el mismo **sistema operativo**.
Todos estos programas están, en apariencia, trabajando al mismo tiempo, pero, en realidad, cada uno de ellos recibe una fracción de segundo para realizar alguna tarea y luego ceder el control de ejecución a otro programa que así lo requiera. De esta forma, nuestra aplicación podría crear diferentes hilos de ejecución para separar la ejecución de funcionalidad y así no bloquear, por ejemplo, la interfaz visual debido a que un bucle está realizando tareas de iteración largas.

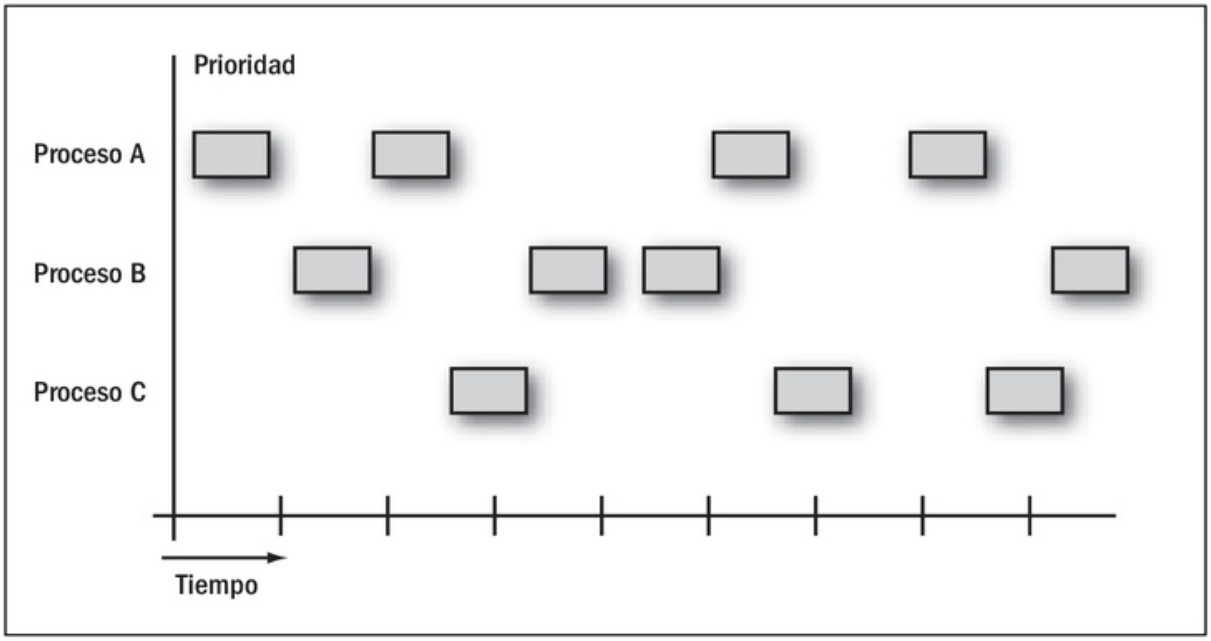


Figura 12. *Este esquema muestra cómo el sistema operativo otorga un contexto de ejecución a cada aplicación en el transcurso del tiempo, dándole un momento muy pequeño para realizar alguna tarea y luego permitir a otro programa ejecutar otra tarea.*

III PAGINADO DE DATOS

Muchas veces, es útil mostrar los resultados de una consulta por páginas que agrupen cierta cantidad de elementos fijos. Con LinQ, podemos combinar tanto **Take** como **Skip** para consultar los datos de forma paginada. Con **Skip**, descartamos un número de elementos que correspondan a páginas viejas y, con **Take**, tomamos desde ese punto, solo los datos de la página actual.

En las aplicaciones con interfaz gráfica, el manejo de los objetos colocados en la pantalla, así como el código de ejecución, suelen ser colocados bajo el mismo hilo. Esto quiere decir que, para poder redibujar cualquiera de los objetos (cajas de texto, listas desplegables, botones, etcétera), se necesitará esperar que el código en ejecución termine de realizar su trabajo para luego redibujar la pantalla. Pero podríamos colocar parte de ese código en un hilo de ejecución separado, y apartarlo de otros para que las distintas funcionalidades puedan ejecutarse sin bloquearse una a la otra. Esto le daría a nuestra aplicación, en especial al usuario, una experiencia e interacción con la misma más fluida.

```
public Hilos()
{
    EjecutarBucle();
    EjecutarBucle2();
}

private void EjecutarBucle()
{
    for (int i = 0; i < Int16.MaxValue; i++)
    {
        Console.WriteLine("i: " + i);
    }
}

private void EjecutarBucle2()
{
    for (int j = Int16.MaxValue; j > 0; j–)
    {
        Console.WriteLine("j: " + j);
    }
}
```

III HILOS

Si bien el uso de hilos puede ser de gran ayuda para la ejecución de tareas en paralelo, tenemos que tener en cuenta que este no es un recurso inagotable y, mientras más hilos creemos y tareas ejecutemos en paralelo, el rendimiento de nuestra aplicación no será constante. Es importante no sobreusar estos elementos en nuestro código.

Vemos dos funciones que ejecutarán dos bucles. El primero contando desde cero al valor máximo de un tipo **Int16**, y el segundo desde el valor máximo hasta llegar a **1**. Pero la ejecución de la segunda función deberá esperar a que termine la primera.

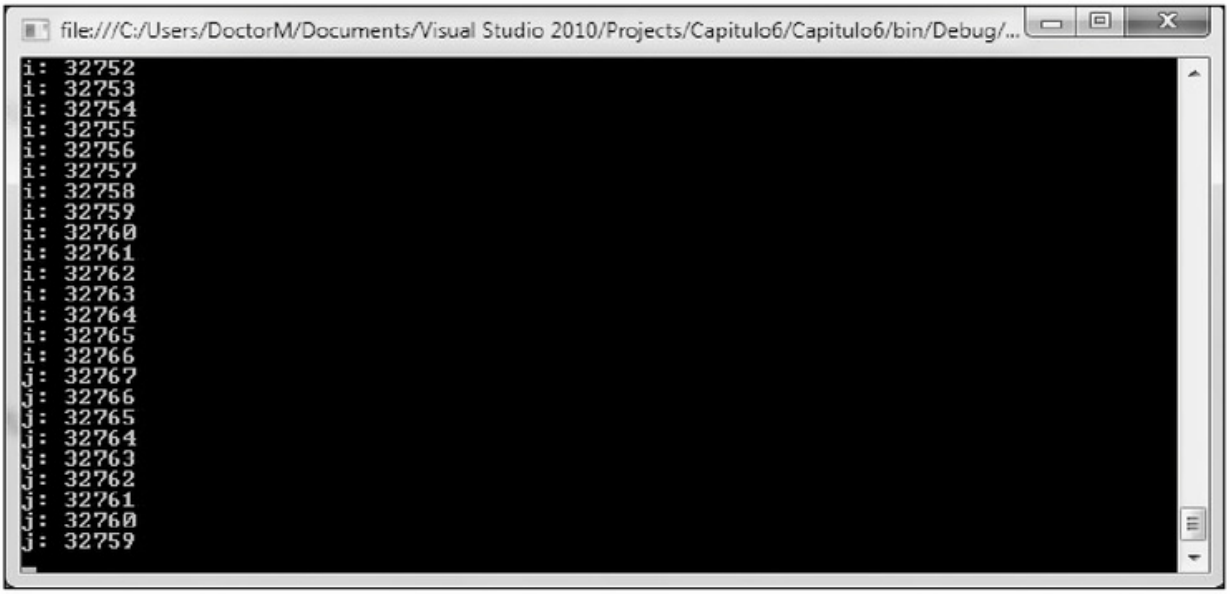


***Figura 13.** Solo se empieza a escribir el contenido de la variable j en el momento en que el primer bucle, que contiene la variable i, termina su ejecución y le da la posibilidad de ejecutarse a la función siguiente.*

Por lo tanto, podríamos crear hilos de ejecución para que estas dos funciones se ejecutaran en paralelo. De esta forma, la segunda función no deberá tener que esperar la finalización de la primera para poder ejecutarse.

```
Thread hilo = new Thread(new ThreadStart(EjecutarBucle2));
hilo.Start();
EjecutarBucle();
```

Mediante el uso del objeto **Thread** (hilo), creamos un nuevo hilo de ejecución. El parámetro de este objeto es un delegado, un puntero a la función que será alojada

LINQ 101

El uso de LinQ puede ahorrarnos la escritura de gran cantidad de líneas de código, pero para lograr una consulta consistente es importante dominar este modelo. Una excelente forma de aprender su uso es mediante ejemplos. En el sitio web de Microsoft sobre LinQ podremos encontrar una lista completa de ejemplos para practicar: **http://msdn.microsoft.com/en-us/vcsharp/aa336746**.

en este hilo y la referencia a la función que alojaremos en el hilo. Al mismo tiempo, cambiamos el orden de las líneas de código para garantizar que primero se ejecute la declaración del hilo y luego la función que no será alojada en un hilo diferente. La función **Start** (iniciar), da comienzo al hilo de ejecución y hace que la siguiente línea de código también se ejecute.

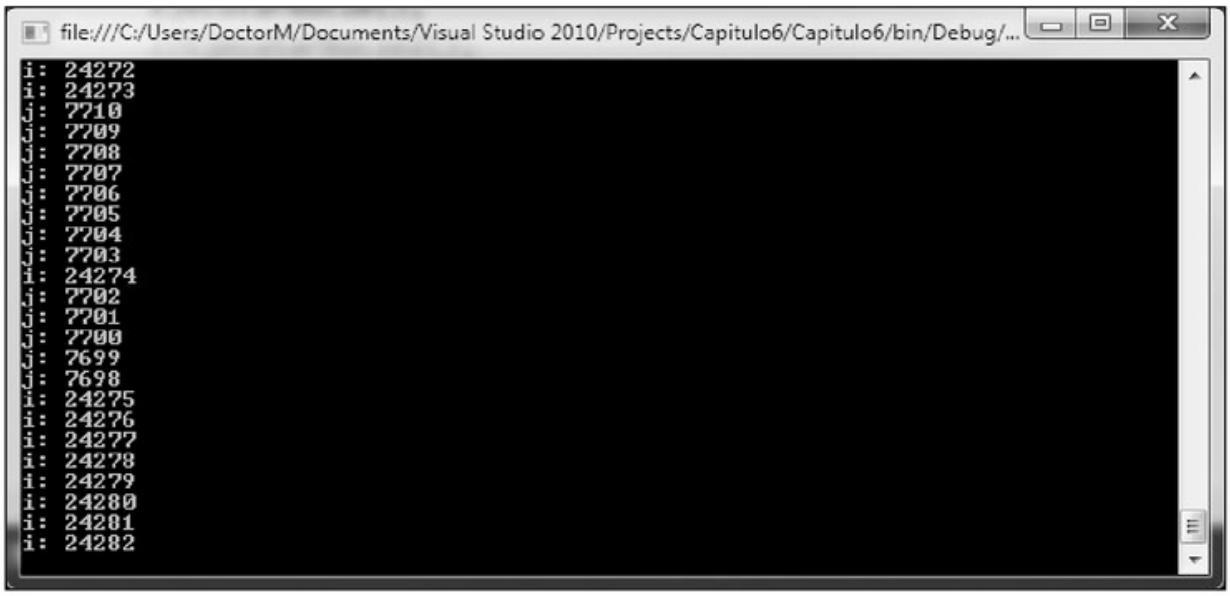


Figura 14. *Tanto la ejecución del bucle que interactúa con la variable i, como la del que lo hace con j se ejecutan en paralelo. El no ver los resultados se debe al tiempo de ejecución que se le otorga a cada hilo para realizar las tareas.*

El delegado utilizado en este caso es del tipo **ThreadStart** (inicio de hilo), pero, además de este, contamos con otros delegados para ejecutar diferentes tipos de funciones. Por ejemplo, podría requerir la ejecución de una función la cual recibe parámetros iniciales al momento de su ejecución.

```
public Hilos()
{
     Thread hilo = new Thread(new
```

LINQ EN PARALELO

La computación actual requiere cada vez de más recursos para poder brindar una mejor experiencia al usuario. Con las nuevas versión por venir de C# y Microsoft .Net Framework se pretende incluir la computación paralela. En la actualidad LinQ ya posee algunas de estas características. Podemos conocer más en: **http://msdn.microsoft.com/en-us/library/dd460688.aspx**.

```
        ParameterizedThreadStart(EjecucionParametrizada));
        hilo.Start("Datos iniciales");
}
private void EjecucionParametrizada(object valor)

{
        //Realizamos diferentes procesos
        ...
        ...
}
```

La función **EjecucionParametrizada** espera un valor como parámetro para su ejecución. En este caso, podemos usar el delegado **ParameterizedThreadStart**, el cual nos permitirá pasar este parámetro en el momento de iniciar la ejecución de la función en el nuevo hilo de ejecución.

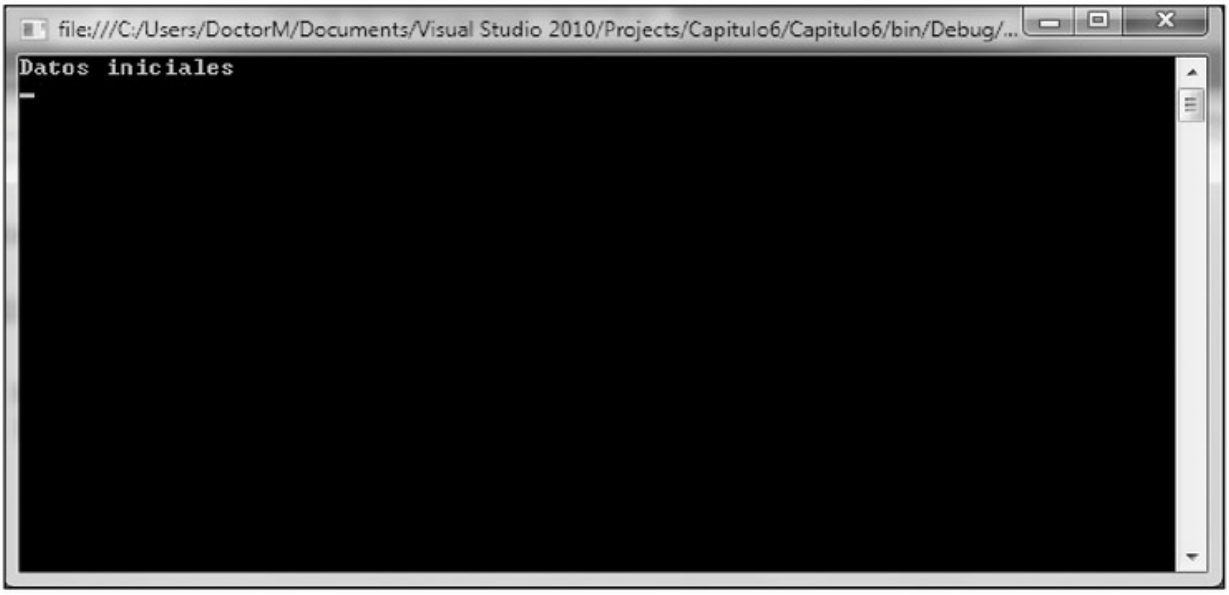


***Figura 15.** Al iniciarse el hilo de ejecución, el valor enviado como parámetro es pasado a la función contenida por el hilo.*

THREAD.SLEEP()

El uso de **Thread.Sleep()** en los ejemplos es solo para ilustrar un comportamiento de carga en la aplicación. Ya que **Thread.Sleep()** detiene momentáneamente la ejecución del hilo actual, el uso en aplicaciones finales podría causar un deterioro en la velocidad de ejecución de estas.

Así como podemos separar las funciones en distintos hilos de ejecución, es posible detener la ejecución de un hilo durante una cantidad de tiempo determinada.

```
private void DetenerHilo()
{
    DateTime inicio = DateTime.Now;
    DateTime fin;

    Thread.Sleep(5000);

    fin = DateTime.Now;

    Console.WriteLine("Hora de inicio: " + inicio. ToLongTimeString());
    Console.WriteLine("Hora de fin: " + fin. ToLongTimeString());
}
```

Mediante la llamada a **Thread.Sleep**, podemos lograr que el hilo actual detenga su ejecución sobre la base de milisegundos. En el ejemplo, el hilo se detiene durante cinco segundos antes de continuar con la siguiente línea de código. Tomamos la fecha y hora antes de detener el hilo de ejecución. Una vez transcurridos los cinco segundos en los que el hilo se detuvo, volvemos a capturar la fecha y la hora. Al mostrar los valores en la consola, vemos que entre las dos capturas, pasaron cinco segundos. Esta técnica no es recomendada en aplicaciones que estén ya en producción.

Inspección por reflejo

Una característica particular del código creado con Microsoft .Net y C# es que todo código compilado contiene información sobre sí mismo que puede ser utilizada para crear nuevas instancias de los tipos contenidos en el código en tiempo de ejecución, leer las posibles decoraciones en el código, restringir la ejecución de funcionalidad sobre la base de políticas de seguridad, entre otros. Este mecanismo recibe el nombre de **reflection** (reflejo) debido a que el código puede mirarse a sí mismo y actuar sobre la base de esto.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
internal sealed class DecoradorAccesoDatos
    : Attribute
{
    public DecoradorAccesoDatos(string nombreColumna)
    {
```

```
    NombreColumna = nombreColumna;
     }

     public string NombreColumna
     {
         get;
         internal set;
     }
}
```

El código anterior crea una clase como las que ya hemos creado anteriormente, pero agrega un nuevo elemento. Al principio de esta, encontramos una **declaración decorativa**. Esta decoración le permite saber al código y a los programas que interactúan con esta clase que esta puede ser utilizada para decorar otras clases de la misma en que esta lo está. Por lo tanto, estamos creando con esta clase una que será utilizada para decorar otras clases.
Para este caso, especificamos que esta clase solo aplica a propiedades de otras clases mediante **AttributeTargets.Property**, y que no podrá aplicarse más de una por propiedad, ya que el atributo **AllowMultiple** (permitir múltiples) es falso. Una vez que tenemos este código, Visual C# 2010 Express lo detectará y nos permitirá usarlo en el código. Al escribir el atributo sobre la propiedad, Visual C# 2010 Express nos brinda la ayuda contextual sobre ella, ya que ha detectado correctamente la existencia del atributo decorativo. El siguiente código representa la clase **Producto** que podría contener los distintos productos leídos desde una base de datos.

```
public class Producto
{
    [DecoradorAccesoDatos("Id_Producto")]
    public int Id { get; set; }
    [DecoradorAccesoDatos("NombreProducto")]
```

III ENSAMBLADO EXTERNO

El uso de **Reflection** (Reflejo) no está limitado a la inspección o interacción con los distintos objetos de nuestro proyecto. Es posible también leer y ejecutar piezas de código externas en tiempo de ejecución. Mediante el uso de **Assembly.Load** podremos leer y cargar en memoria otros ensamblados siempre y cuando nuestro código conozca los diferentes tipos que allí existan.

```
 public string Nombre { get; set; }

    [DecoradorAccesoDatos("DescProducto")]
    public string Descripcion { get; set; }

    [DecoradorAccesoDatos("PrecioProducto")]

    public decimal Precio { get; set; }

    [DecoradorAccesoDatos("Disponible")]
    public bool Disponible { get; set; }
}
```

De esta forma, teniendo nuestra clase **Producto** decorada para cada una de sus propiedades, sería muy fácil crear algunas líneas de código para leer los datos desde la base de datos, y que, automáticamente, lea los valores retornados desde un objeto **DataReader** y asigne los resultados de la consulta a esta clase.

```
internal List<T> CrearObjeto(IDataReader reader)
{
     //Creamos una lista de tipos genéricos
    List<T> objetoGenerico = new List<T>();

     //Recorremos el DataReader
    while (reader.Read())
    {
            //Creamos un objeto concreto del tipo T
        T objetoConcreto = new T();

            //Recorremos todas las propiedades del tipo T
        foreach (PropertyInfo property in
            objetoConcreto.GetType().GetProperties())
        {
             object[] propertyList =
             property.GetCustomAttributes(typeof(DecoradorAccesoDatos),
             true);

             if (propertyList.Length > 0)
             {
```

```
    string nombreColumna = (propertyList[0] as
                    DecoradorAccesoDatos).NombreColumna.ToString();

    property.SetValue(objetoConcreto, reader[nombreColumna],
                    null);
                }
            }


            objetoGenerico.Add(objetoConcreto);
        }

        return objetoGenerico;
}
```

El código que vemos anteriormente sirve para llenar y retornar una lista de objetos genéricos, llenándolos sobre la base de las decoraciones de la clase del tipo **T** enviado. Esto quiere decir que usaremos el concepto de reflejo para leer los atributos del tipo **T**, y así poder saber cómo están decorados.

```
foreach (PropertyInfo property in objetoConcreto.GetType().GetProperties())
```

Esta línea es la que nos deja leer todas las propiedades contenidas en el tipo enviado. **objetoConcreto** es una instancia de **T**, por lo que si **T** fuera igual a la clase **Producto** entonces, al llamar a **GetProperties**, obtendríamos cada una de las propiedades contenidas en esta clase. Estas propiedades pueden ser evaluadas e inspeccionadas. En la consola, escribimos el nombre de cada una de ellas. El siguiente paso es tomar los atributos decorativos de cada una de las propiedades encontradas y verificar si son del tipo que nosotros hemos creado.

```
object[] propertyList =
     property.GetCustomAttributes(typeof(DecoradorAccesoDatos), true);
if (propertyList.Length > 0)
     {
            string nombreColumna = (propertyList[0] as
     DecoradorAccesoDatos).NombreColumna.ToString();
            ...
            ...
```

En el código, recibimos un vector de atributos decorativos. Esto se debe a que puede haber más de un atributo decorativo asignado a una propiedad. Como solo hemos decorado nuestra clase **Producto** con un único atributo, lo obtenemos accediendo al índice cero del vector. Con el nombre descriptivo contenido en el atributo, podremos usar este para especificar la columna desde la cual queremos obtener los datos contenidos en el objeto **DataReader**.

```
property.SetValue(objetoConcreto, reader[nombreColumna], null);
```

SetValue fuerza la inserción de un valor dentro de la propiedad. Habiendo obtenido el valor de la columna mediante el atributo decorativo, solo necesitaremos especificarlo como índice del **DataReader** y utilizar su contenido como el valor por insertar en la propiedad. Al ejecutar el código, vemos cómo los datos son obtenidos desde la base de datos, y la lista de productos es rellenada. Una vez que los datos son recuperados desde la base de datos, y la lista es cargada usando la técnica de reflejo, podemos recorrer los resultados y verificar que han sido cargados en forma correcta. El uso de esta técnica no se reduce solo a la interacción con atributos en las clases. Existen diferentes patrones de código que utilizan la característica de la inspección de los tipos de datos para saber cómo comportarse. Controles visuales que solo requieren una lista de objetos para mostrar la información en filas y columnas, ordenamientos dinámicos basados en el nombre de la propiedad por la cual queremos ordenar, manipulación de datos en una base de datos utilizando solo esquemas de objetos desde nuestro código y otras técnicas avanzadas son posibles gracias al uso de esta característica provista por el lenguaje.

... RESUMEN

Este capítulo nos ha llevado a conocer las máximas cualidades de Microsoft .Net y C#. Hasta aquí, hemos recorrido todos los puntos del desarrollo de software con C#, desde sus inicios hasta lo más avanzado. Ahora ya estamos listos para llevar todo esto a las dos áreas más grandes del desarrollo de software. Por un lado, el desarrollo de escritorio y las aplicaciones para Windows. Por el otro, el desarrollo para la Web, la creación de sitios web con ASP.net. Si bien en los siguientes capítulos nos dedicaremos exclusivamente a los dos mundos mencionados, no debemos olvidarnos de que lo aprendido es aplicable a todos los desarrollos y, mientras más dominemos estos conceptos, más fácil nos resultará dar el siguiente paso.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué delegado deberíamos usar para declarar un hilo para una función que requiere parámetros de entrada?

2 Necesitamos detener el hilo de ejecución actual durante 10 segundos, ¿qué línea de código deberíamos usar?

3 ¿Para qué sirve la sentencia Finally?

4 ¿Qué tipo de excepción debería utilizar si se requiere capturar un error producido al haber querido acceder a un índice de vector inexistente?

5 ¿Qué es un método extendido?

6 ¿Los métodos extendidos pueden ser declarados en cualquier clase?

7 ¿Existe alguna versión de LinQ para interactuar con XML?

8 ¿Podemos utilizar una sentencia try sin colocar ningún catch?

9 ¿Es posible redefinir el tipo resultante de una consulta con LinQ?

10 ¿De cuántas formas podemos escribir una consulta LinQ?

EJERCICIOS PRÁCTICOS

1 Utilizando la clase que define la utilización de atributos decorativos, cree otra que solo pueda ser utilizada a nivel de clase.

2 Con la clase anterior, decore un nuevo tipo e intente encontrarlo utilizando la técnica de reflejo.

3 Provoque una excepción que acumule valores más allá del límite de un tipo entero utilizando un bucle.

4 Con el error anterior, captúrelo utilizando un bloque catch lo más específico que pueda.

5 Intente extender el tipo string agregando funcionalidad para escribir, una a una, las letras del texto contenido en una variable de este tipo.

Capítulo 7

Programación para escritorio

Las aplicaciones de escritorio son aquellas con las cuales estamos más familiarizados, desde navegadores web y procesadores de texto, hasta hojas de cálculo o programas de diseño y edición de imágenes. Este capítulo está dedicado enteramente a la programación de aplicaciones de escritorio bajo Windows.

INTERFACES VISUALES

Las aplicaciones de escritorio se caracterizan por la utilización de interfaces visuales, las cuales pueden contener un conjunto de controles con los cuales el usuario puede interactuar. Las principales funcionalidades de estas interfaces son provistas por el sistema operativo; esto quiere decir que el sistema operativo nos provee de un marco común para todas las aplicaciones que escribamos, respetando las reglas básicas que todas las aplicaciones respetan. Estas también reciben el nombre de **formularios** debido a la forma en cómo son utilizados para diseñar y maquetar la aplicación.

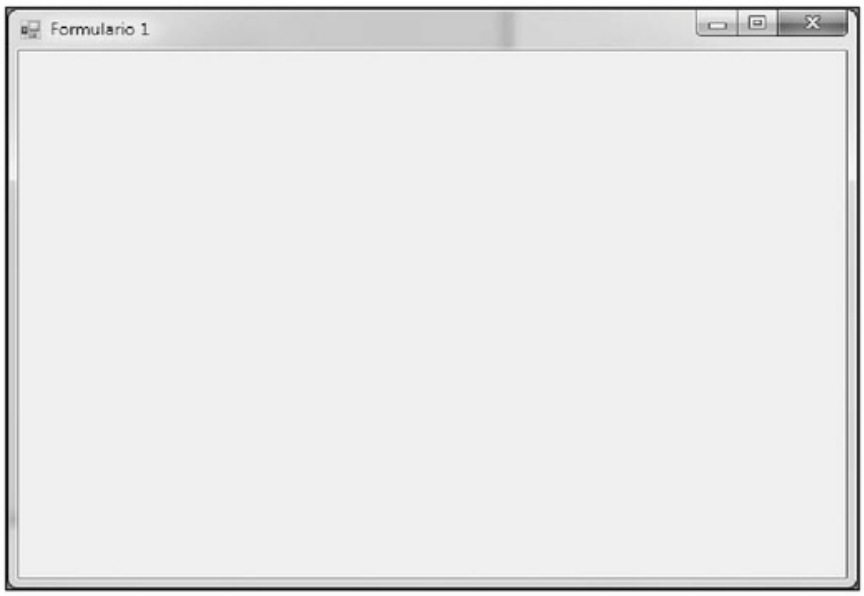


Figura 1. *El icono del formulario, la barra de título y los botones de acción son provistos por defecto con el formulario. Es posible ocultar o modificarlos desde código, pero todos los formularios implementan las mismas reglas básicas.*

Los diferentes botones a la derecha del formulario así como el icono principal y la barra de título son algunas de las reglas que el sistema operativo espera que nuestros formularios contengan e implementen. Esto quiere decir que las funcionalidades detrás de cada botón son **heredadas** y no requieren la realización de código por parte nuestra para que se ejecuten. Por lo tanto, al crear un formulario para

III CONTROLES

Existen dos tipos de objetos que podemos utilizar en el desarrollo de aplicaciones de escritorio para Windows. Los **controles** son aquellos que proveen funcionalidad empaquetada adicional para el desarrollo de aplicaciones que incluyen el soporte de interfaces visuales. Cuando hablamos de controles, nos referimos a piezas de código que pueden interactuar con el usuario.

nuestra aplicación, si presionamos sobre el botón **Maximizar**, **Minimizar** o **Cerrar**, estos ejecutarán dichas acciones sin que nosotros tengamos que intervenir con código propio. Como dijimos, los formularios nos sirven para crear la interfaz visual de nuestras aplicaciones y, como tales, sirven de contenedores de elementos, piezas de código con soporte visual que dan vida a la aplicación. Algunos de los elementos más conocidos son los **botones**, **cajas de texto**, **calendarios**, **casillas de verificación**, **listas desplegables**, entre otros (**Figura 2**). Por lo tanto, para construir aplicaciones de escritorio para Windows, utilizaremos los formularios como los contenedores de la funcionalidad con la cual el usuario deberá interactuar.

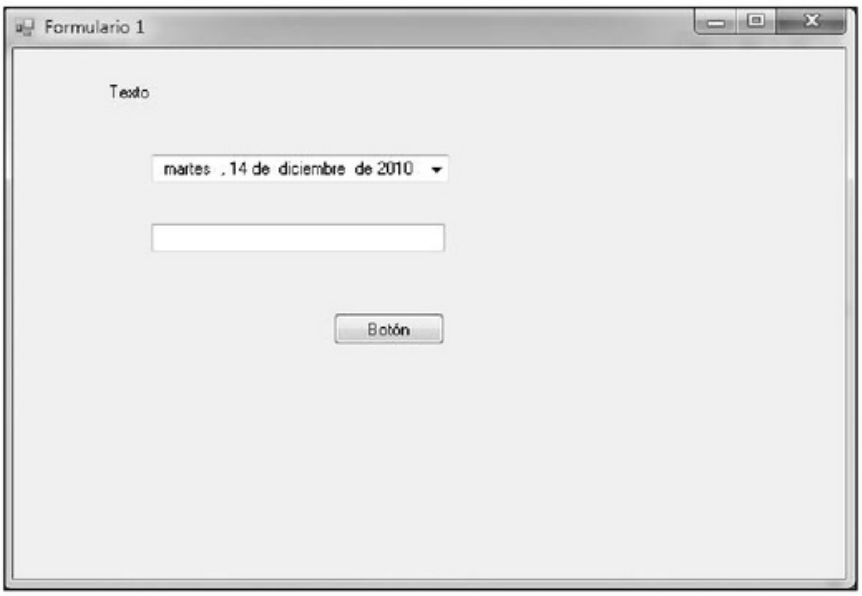


Figura 2. *Microsoft .Net posee un conjunto variado y esencial de controles visuales listos para ser usados dentro de nuestros formularios y que nos ayudan en la construcción rápida de las aplicaciones de escritorio.*

La primera aplicación

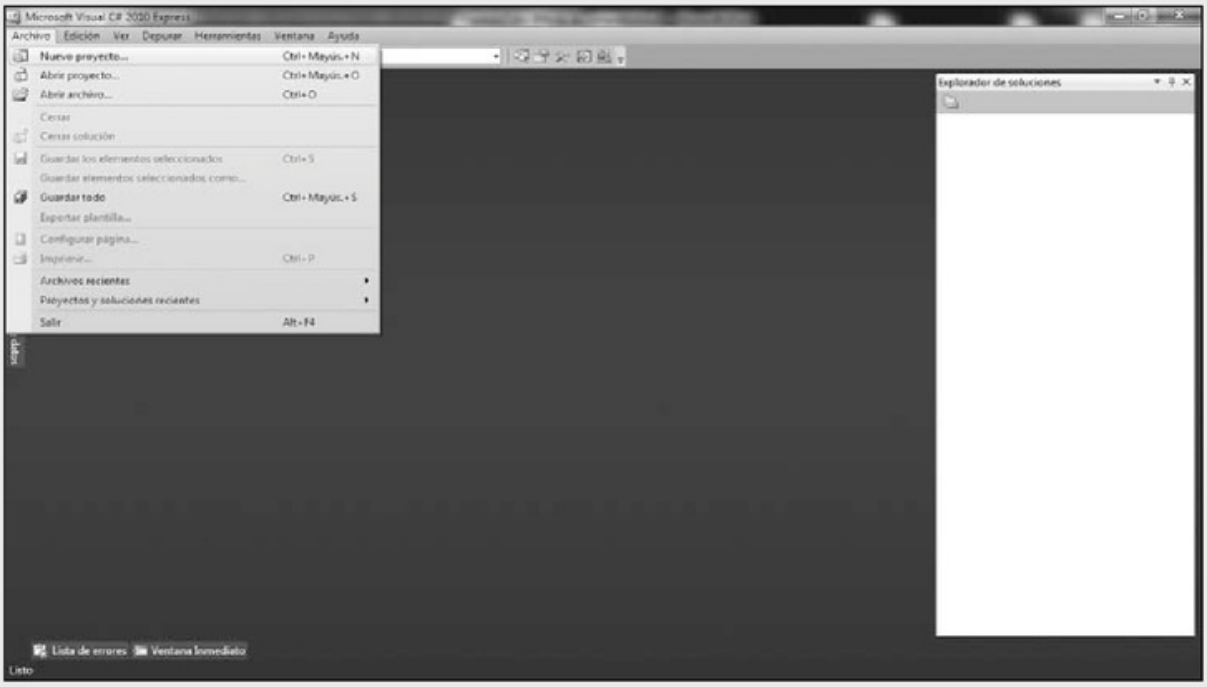
Crear una aplicación de escritorio no difiere del proceso de creación de aplicaciones que hemos estado realizando en los capítulos anteriores. Solo deberemos seleccionar otro tipo de aplicación en vez de una aplicación de consola.

III COMPONENTES

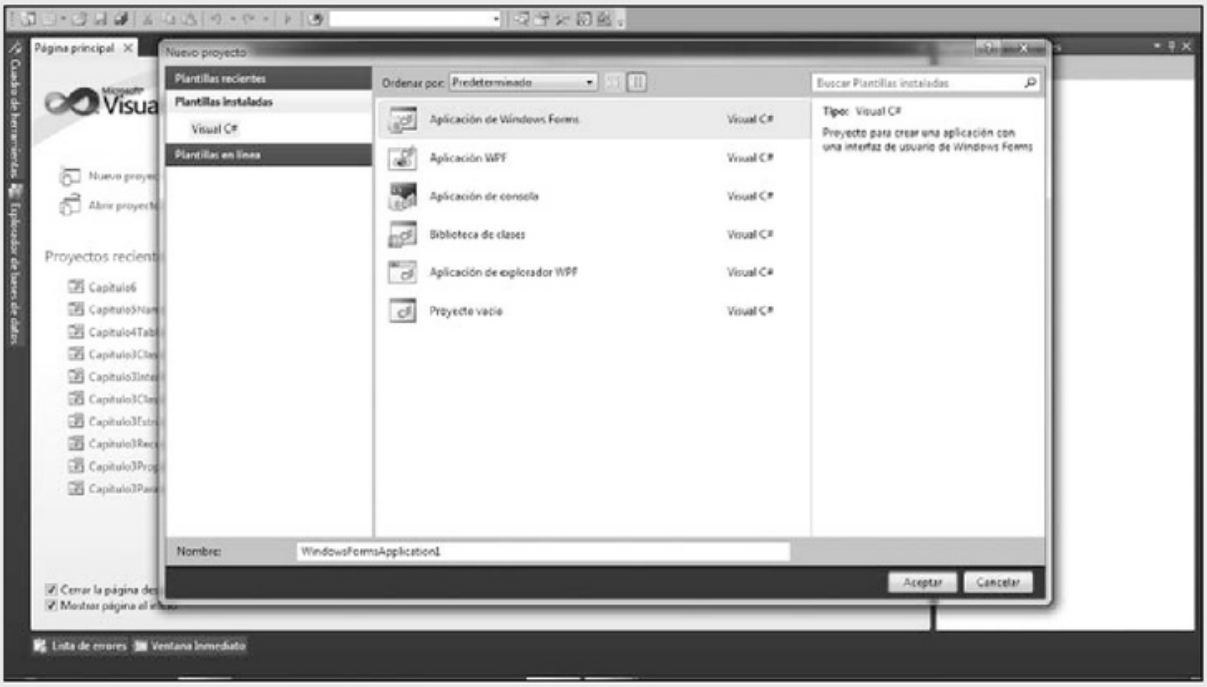
Los **componentes** son el segundo conjunto de objetos con funcionalidad empaquetada para el desarrollo de aplicaciones. A diferencia de los controles, que proveen interfaces visuales hacia el cliente, los componentes no lo hacen. En cambio, brindan mayor soporte para el desarrollador de código con funcionalidad avanzada no provista por el entorno de desarrollo.

Creación de una aplicación de escritorio PASO A PASO

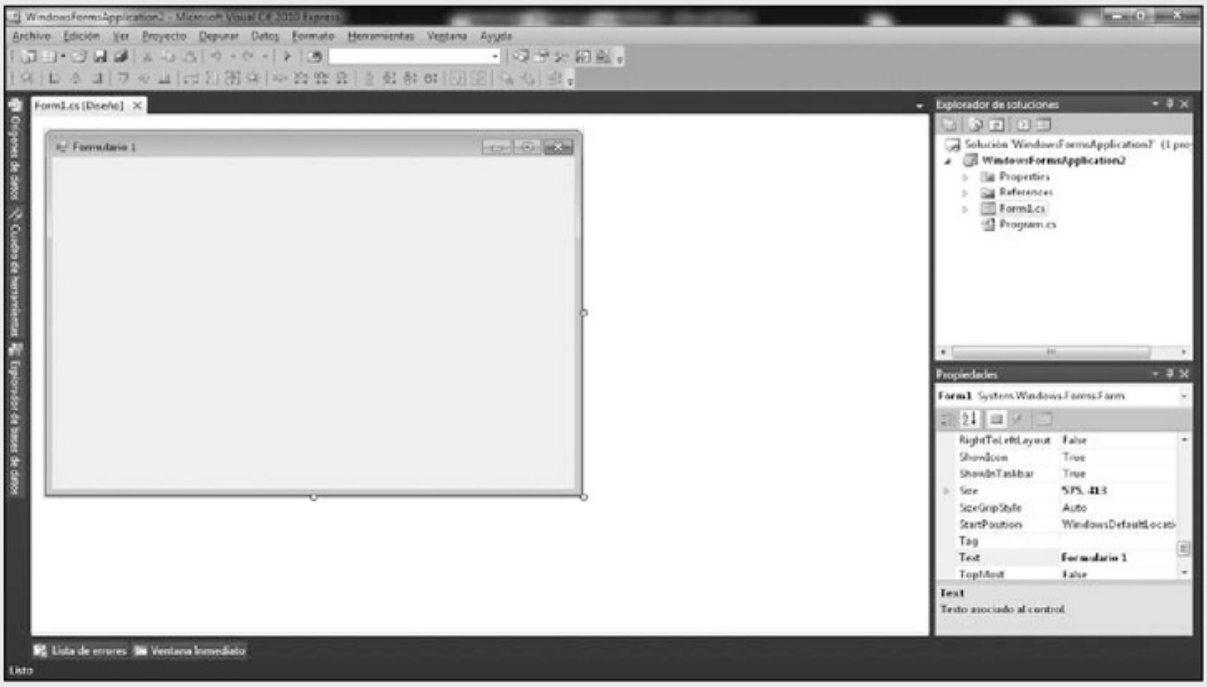
1 Como primer paso, en el Visual C# 2010 Express, presione sobre el menú **Archivo** y luego en **Nuevo Proyecto**.



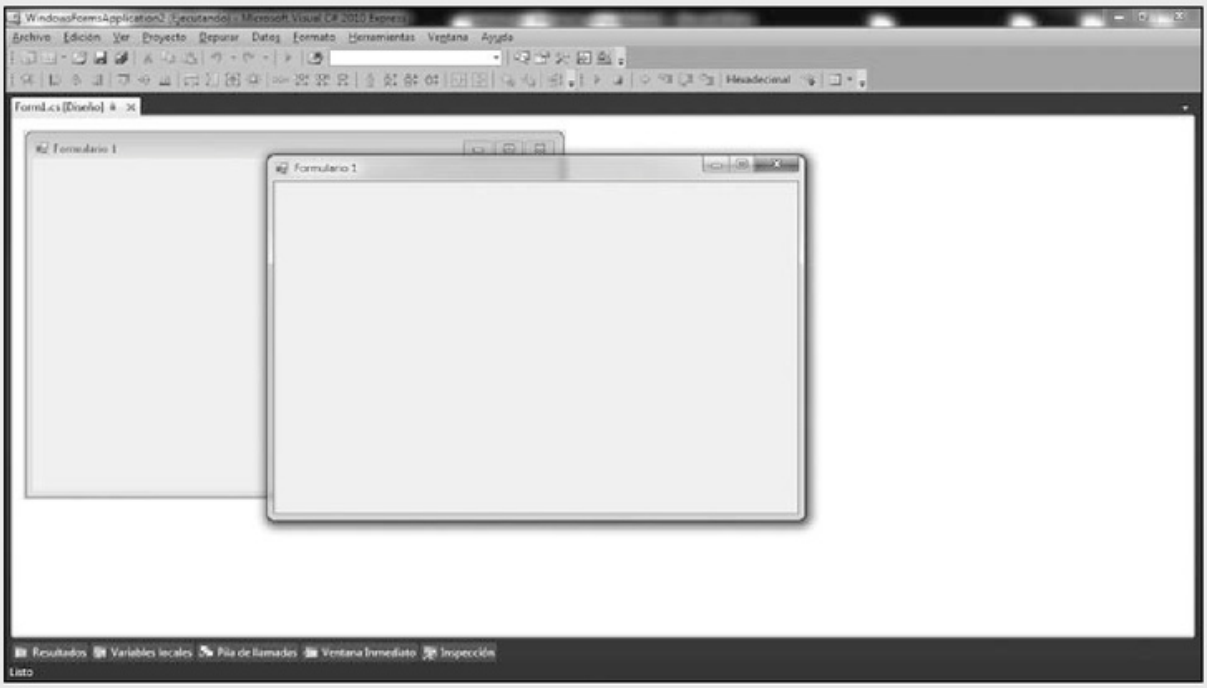
2 A continuación, en la ventana de opciones, debe seleccionar **Aplicación de Windows Form** y colocar un **nombre** cualquiera a la aplicación en el campo Nombre. Luego presione el botón **Aceptar**.



3 Presione dos veces con el botón derecho del mouse sobre el elemento **Form1** contenido en el **Explorador de soluciones** para ver el formulario principal de nuestra aplicación. El formulario se mostrará en **modo diseño**.



4 Para ejecutar el código en desarrollo presione la tecla **F5**, la cual dará inicio a la aplicación en mododepuración. Para volver a Visual C# 2010 cierre el formulario de la aplicación que se acaba de abrir.



El último paso realizado en la sucesión de pasos ejecuta la aplicación recién creada y nos muestra nuestra primera aplicación de escritorio en funcionamiento. Notaremos que esta no contiene mayor funcionalidad que la de poder maximizarse, minimizarse y cerrarse, dando como resultado de esta última acción la finalización de la aplicación, liberando todos los recursos que pudiera utilizar.

Código autogenerado

Cuando creamos la aplicación de escritorio y el nuevo formulario fue adicionado a esta, algunas líneas de código fueron autogeneradas por Visual C# 2010 Express para manejar las distintas características de este. Esas características están asociadas directamente con propiedades del formulario, como el título, el nombre, el tamaño, entre otros. Es posible modificar estas propiedades de forma visual seleccionando el formulario y actualizando las propiedades desde la caja de **Propiedades** (**Figura 3**).

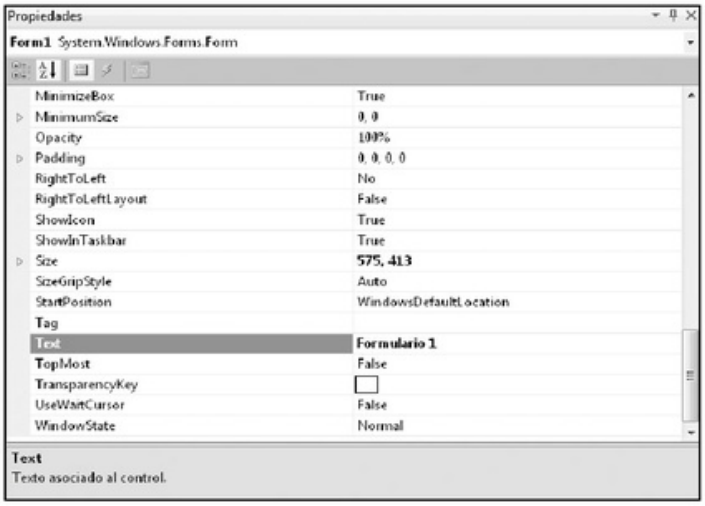


***Figura 3.** La cantidad de propiedades listadas en la caja de **Propiedades** depende directamente del elemento seleccionado. Desde esta, no solo pueden ser modificados los formularios. Otros controles también pueden ser alterados desde esta caja.*

Si bien modificar los valores de forma visual puede resultar sencillo, en muchas ocasiones nos veremos obligados, por flexibilidad, a realizar estas modificaciones por código. En el caso de los formularios, todas las propiedades así como los controles

FORMULARIOS EN EL PASADO

Antes de la aparición de Microsoft .Net, los lenguajes de programación, en especial los orientados a objetos, rara vez proveían una forma fácil de interactuar con los modelos visuales del sistema operativo. Para poder crear un formulario, se requerían cientos de líneas de código, no solo para definirlo visualmente, sino para controlar todo su comportamiento.

que agreguemos en este serán declarados en un archivo que acompaña a la definición del formulario. Este archivo recibe el nombre del formulario acompañado de la extensión **Designer.cs** (archivo de formulario para diseño de C#).

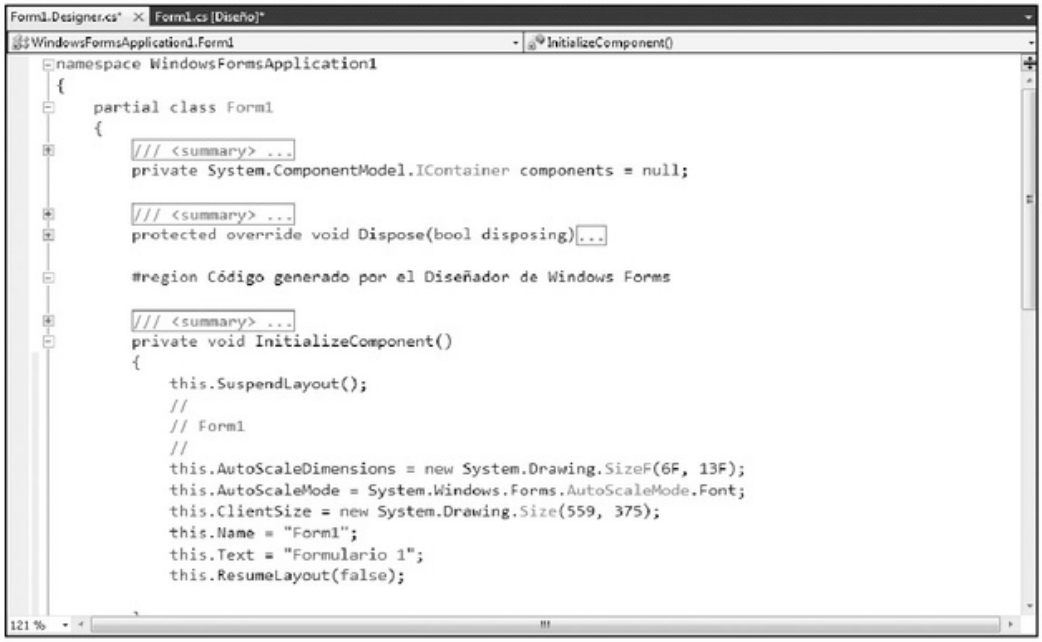


Figura 4. *Al abrir el archivo* ***Form1.Designer.cs****, notamos que en este se encuentra la implementación del comportamiento visual del formulario que estamos creando. Modificar los valores significará que la apariencia visual del formulario cambie.*

Las propiedades que pudiéramos modificar de forma visual en la caja de propiedades son traducidas a código dentro de este archivo, por lo tanto, si modificáramos cualquiera de estas líneas, esto repercutiría de manera automática en el formulario.

```
this.ClientSize = new System.Drawing.Size(559, 375);
this.Name = "Form1";
this.Text = "Formulario 1";
```

Las líneas de código anteriores especifican algunas de las características del formulario con el cual estamos trabajando, como por ejemplo, su tamaño y su título. Podríamos modificar estas propiedades desde el código, de la siguiente forma.

```
//Modificamos el tamaño
this.ClientSize = new System.Drawing.Size(300, 200);
this.Name = "Form1";
//Modificamos el título
this.Text = "Formulario principal de la aplicación";
```

Si observamos nuevamente el formulario, notaremos que ahora el título y el tamaño han cambiado (como vemos en la **Figura 5** que se encuentra a continuación).

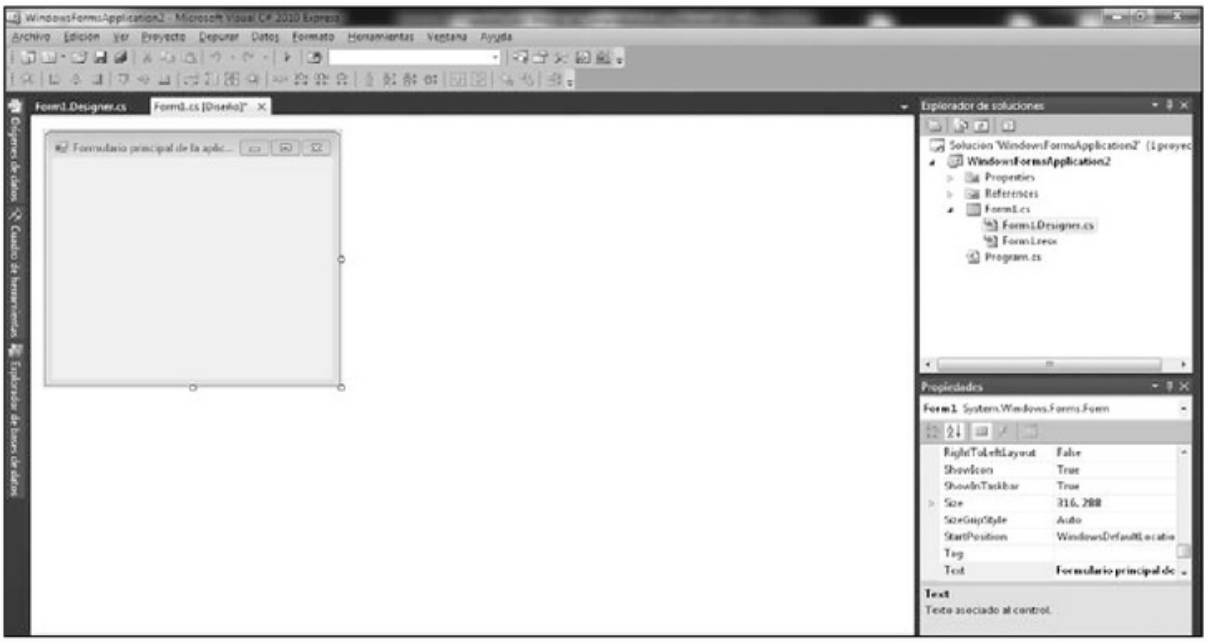


***Figura 5.** Al encontrarse separado el esquema del formulario y su implementación visual por medio de líneas de código, obtenemos mayor control sobre su comportamiento.*

La caja de herramientas

Junto a los formularios, existe un conjunto de controles y componentes que nos ayudan en la construcción de las aplicaciones. Microsoft .Net nos provee aquellos que son usados con mayor frecuencia en el desarrollo bajo Windows. La lista completa de controles disponibles se encuentra en el **Cuadro de herramientas** (**Figura 6**).

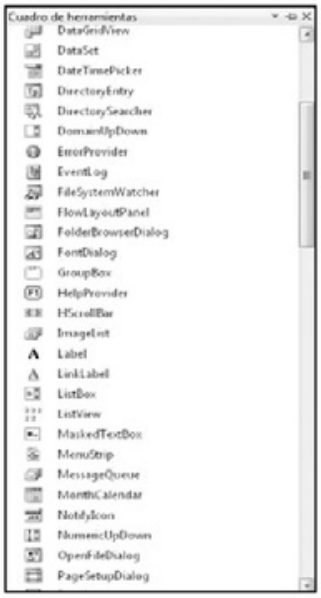


***Figura 6.** El Cuadro de herramientas provee todo el abanico de controles listos para ser usados en nuestras aplicaciones.*

En la **Tabla 1**, podemos ver una lista de los controles que se consideran como los más usados y con los cuales deberemos estar familiarizados.

CONTROL	DESCRIPCIÓN
Botón	Un control que representa el botón tradicional encontrado en las aplicaciones de escritorio. Se pueden ejecutar distintas funcionalidades tras presionar con el mouse sobre él.
Etiqueta	Utilizada para agregar textos descriptivos asociados a los distintos controles.
Calendario	Despliega y permite seleccionar una fecha mostrando una caja emergente que contiene un calendario.
Cuadro de imagen	Muestra una imagen determinada dentro del formulario de nuestra aplicación.
Selector circular	Sirve para seleccionar una opción entre varias alternativas teniendo solo una como válida.
Caja de texto	Campo de texto utilizado para la introducción de texto por parte del usuario.
Cuadro de verificación	Al igual que el selector circular, este permite marcar o desmarcar una opción en particular. Suele ser representado como un cuadrado con una cruz interna cuando está seleccionado.
Lista desplegable	Muestra una lista de opciones que se despliega y contrae ocupando poco espacio en la interfaz visual.
Tira de menú	Utilizado para crear el menú superior de las aplicaciones. Puede contener un menú principal y una estructura en forma de árbol lógico con submenús.
Tira de herramientas	Muestra una lista de opciones en la parte superior de la ventana; esta puede contener imágenes, botones y otros controles estándares.
Grilla de datos	Muestra una grilla en formato filas y columnas que pueden contener diferentes datos dentro de cada celda.
Temporizador	Componente sin interfaz visual que sirve para disparar eventos cada un tiempo previamente establecido.
Cajas de diálogos	Estas cajas de diálogos poseen diferentes funcionalidades. Desde la selección de archivos para su posterior lectura, como para ser guardados en el disco duro, para seleccionar colores de una paleta de colores y hasta para imprimir un documento en una impresora local.

***Tabla 1.** En esta lista, se presentan solo algunos de los controles de mayor uso en el desarrollo de aplicaciones para escritorio con Microsoft .Net.*

En la **Tabla 1**, solo nombramos algunos de los tantos controles disponibles. Es importante saber cómo podemos agregar estos controles a nuestro formulario y luego modificar sus propiedades. Conociendo los mecanismos básicos que rigen los distintos controles, adaptarse a nuevos es cuestión de realizar un análisis simple. Notaremos que muchas de las propiedades en tres controles son colunes, como la propiedad **Text** (texto).

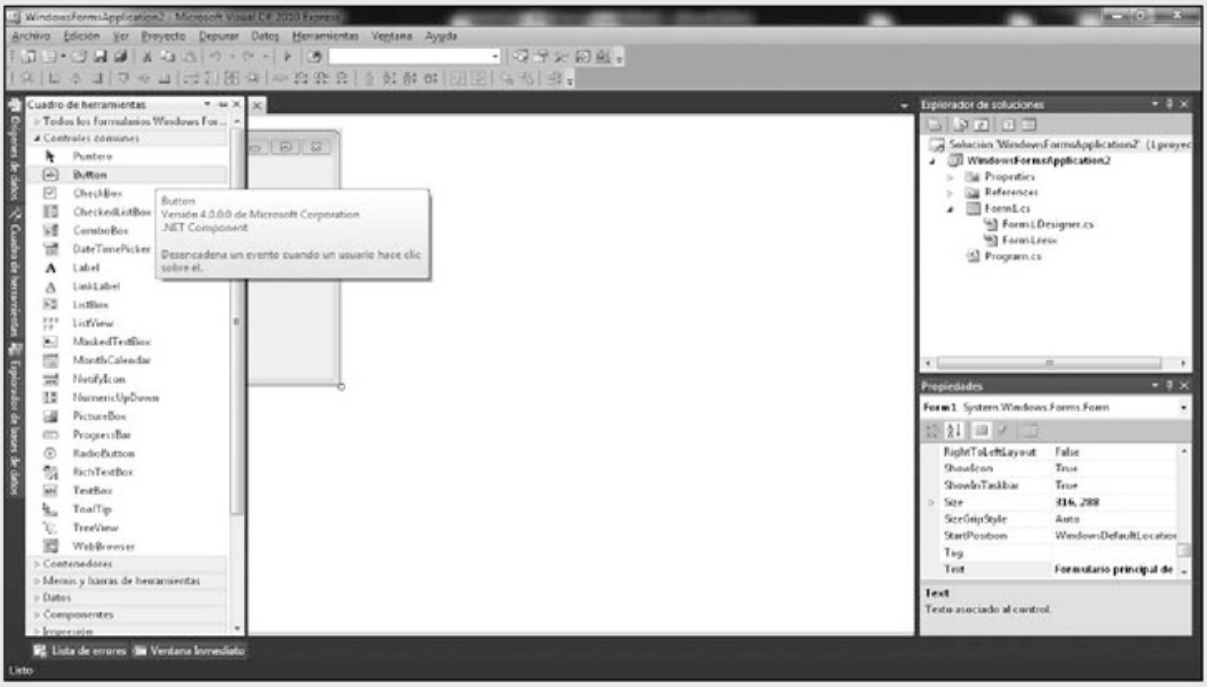
III OTROS CONTROLES

Los componentes y controles provistos por Microsoft .Net para el desarrollo de aplicaciones de escritorio no son los únicos posibles y válidos de utilizar. Diferentes empresas alrededor del mundo se encargan de generar este tipo de objetos, empaquetarlos y venderlos. El desarrollador puede beneficiarse de estos y ahorrarse largas horas de trabajo.

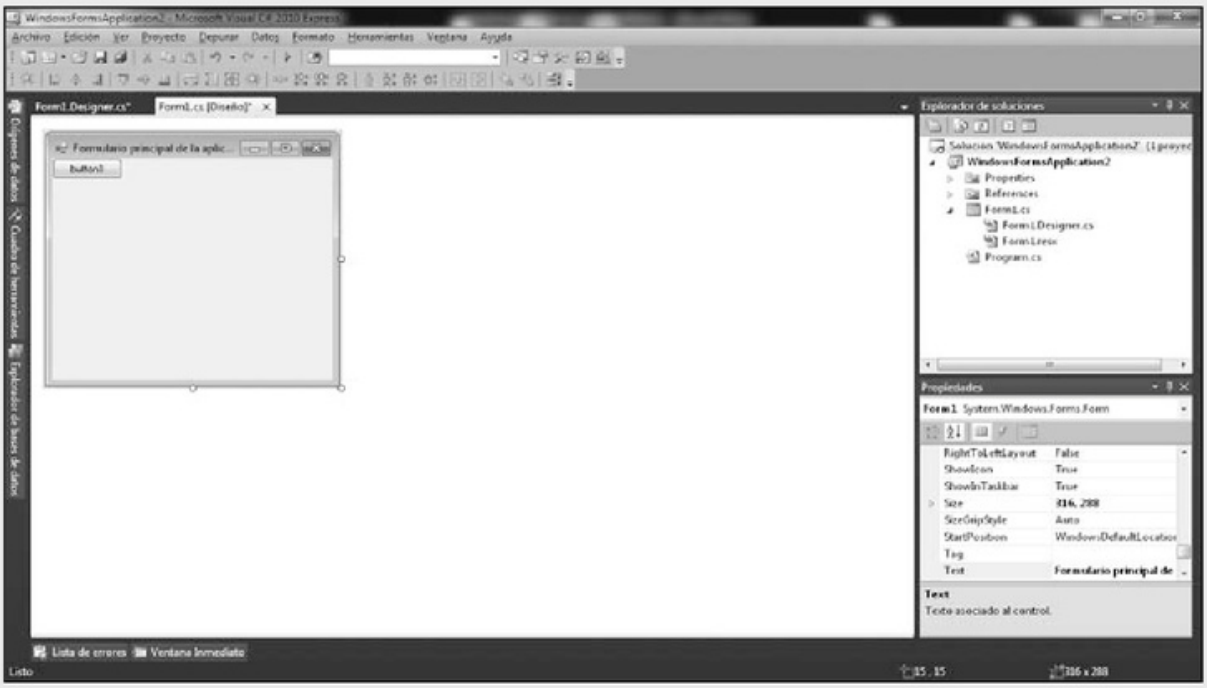
Agregar controles a un formulario

PASO A PASO

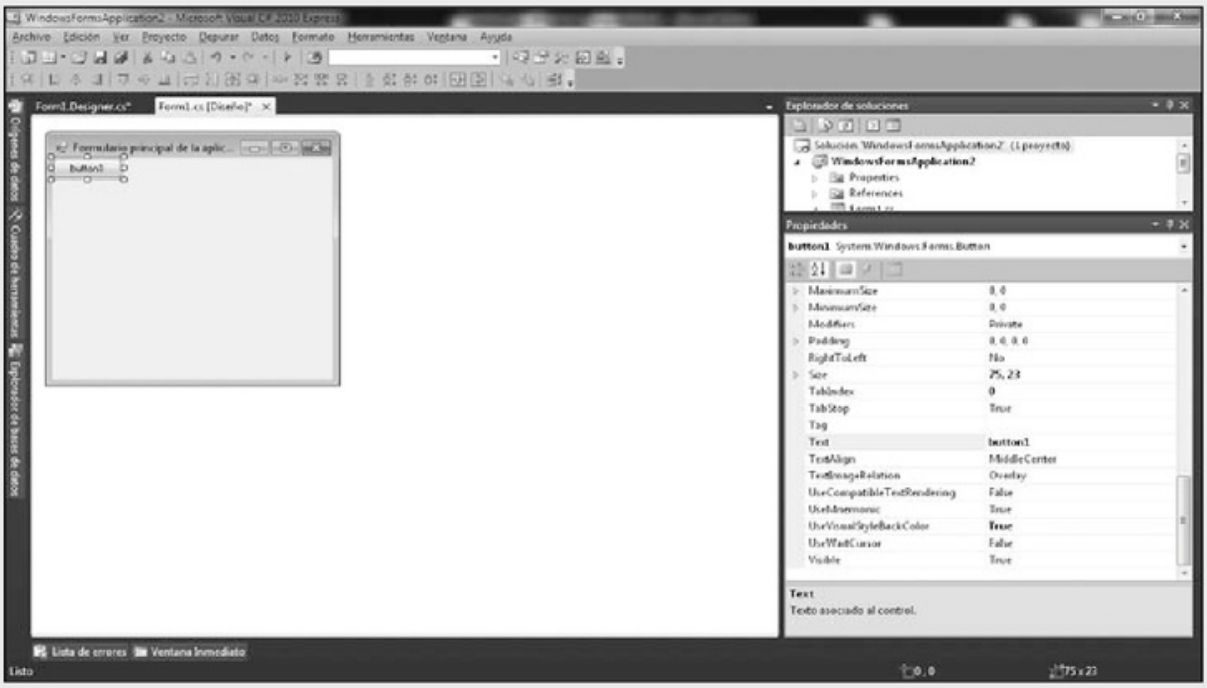
1 Para comenzar, abra el **Cuadro de herramientas** colocando el mouse sobre él y expanda la sección **Controles comunes**.



2 Seleccione el control **Button** y presione dos veces sobre él con el botón izquierdo del mouse o sobre el formulario, presionando el botón izquierdo del mouse y, a continuación, arrástrelo para dibujar el botón.



3 Cierre el **Cuadro de herramientas** y seleccione el control adicionado para modificar sus propiedades en el cuadro **Propiedades**.



Una vez que un control ha sido adicionado al formulario, desde el cuadro **Propiedades** podremos modificar sus valores como en el caso del formulario. Por otra parte, y al igual que el formulario, el control es representado por líneas de código. Si abrimos el archivo **Designer.cs** además de las líneas que definen el formulario, podremos notar, ahora, las que definen el control de tipo botón. Cualquier modificación realizada sobre estas líneas impactarán sobre el botón en cuestión.

```
//
// button1
//
this.button1.Location = new System.Drawing.Point(0, 0);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "button1";
this.button1.UseVisualStyleBackColor = true;
```

Todas las propiedades configuradas hasta el momento sobre el control de tipo botón son reflejadas en el código anterior (**Figura 7**).

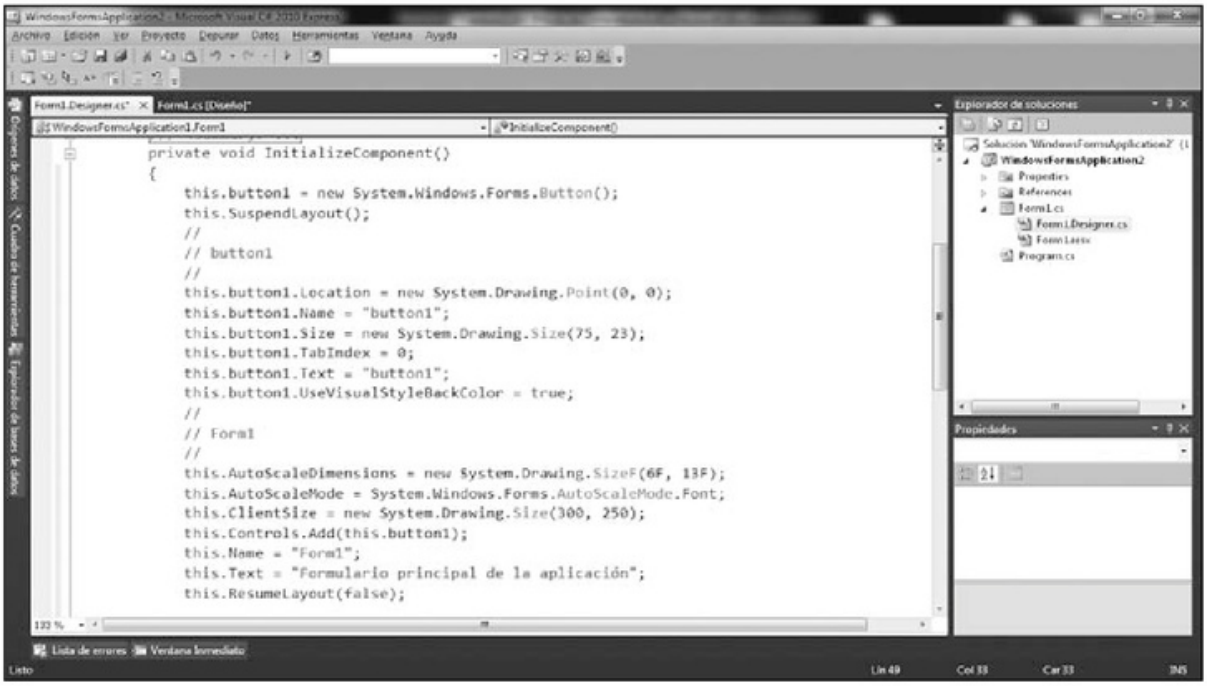


Figura 7. *Todos los controles que son adicionados al formulario irán agregando diferentes líneas de código con sus configuraciones en el archivo* ***Designer.cs****.*

Eventos de formulario

Así como contamos con un archivo para alojar las líneas de código, el formulario en sí mismo es un conjunto de líneas de código con características especiales.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
```

El formulario es solo una clase, pero con la particularidad de que hereda su funcionalidad desde la clase **Form** del espacio de nombres **System.Windows.Forms**; por lo tanto,

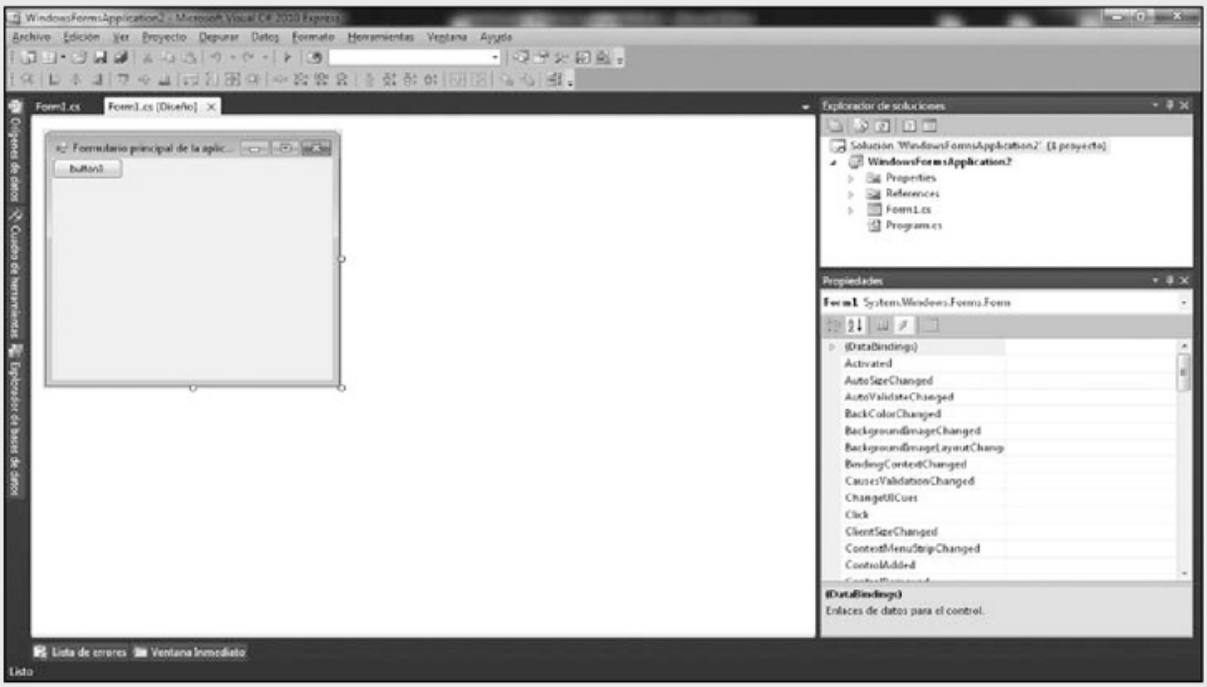
III RAÍZ DE LOS EVENTOS

Como muchas clases, controles y componentes en Microsoft .Net, los parámetros en los eventos respetan una cadena de herencias hasta llegar a una raíz común. Es posible explorar esta cadena de herencias seleccionando cualquier parámetro recibido en un evento y, sobre él, presionar la tecla **F12**. Esto nos irá llevando más adentro en la jerarquía de clases hasta la raíz.

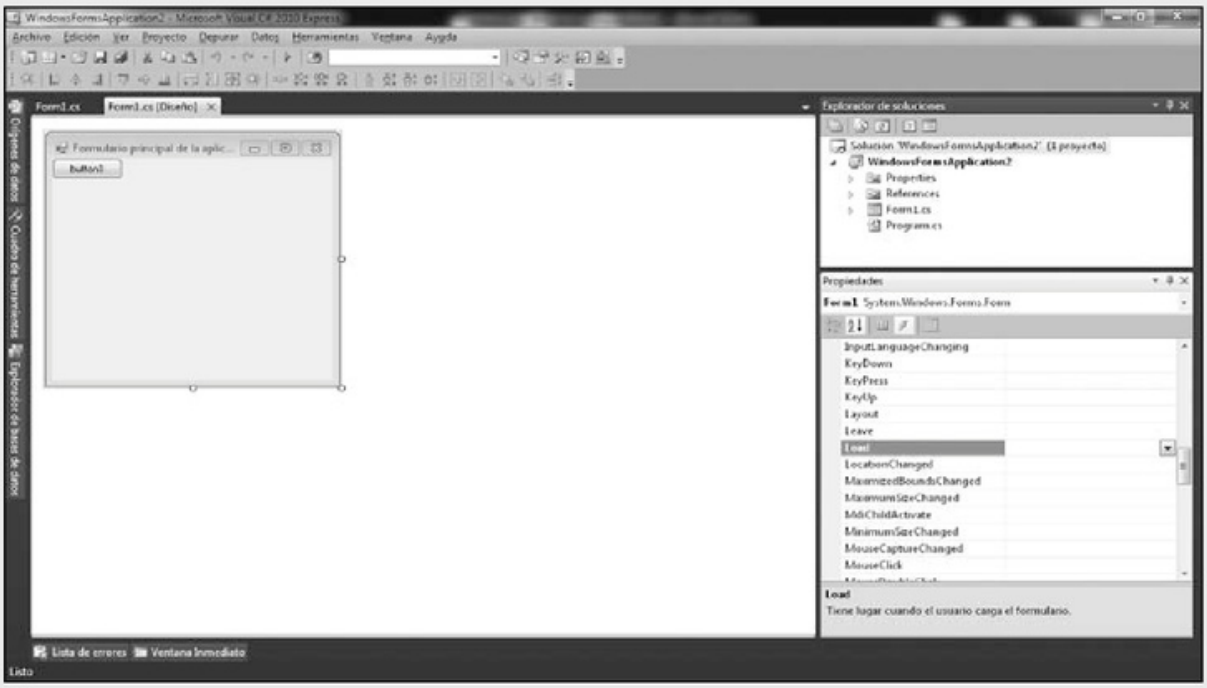
si necesitamos crear líneas de código que interactúen con el usuario, este será el lugar indicado. La mayoría de esta funcionalidad es disparada por medio de eventos.

Agregar eventos a un formulario PASO A PASO

1 Seleccione en el cuadro de **Propiedades** el icono con forma de rayo.



2 Navegue por las opciones hasta encontrar el evento **Load**.



3 Presione dos veces con el botón izquierdo del mouse sobre el espacio vacío al lado del evento. Esto creará el código necesario para él.

Las instrucciones del **Paso a paso** anterior nos permiten agregar el código que enlaza nuestro formulario con los eventos que se irán disparando a medida que el código es ejecutado. En este caso, el evento **Load** (cargar) se dispara cuando el formulario es inicializado, sus controles creados y la interfaz visual se encuentra lista para ser utilizada. Este evento es ideal para inicializar la interfaz de cara al usuario.

```
private void Form1_Load(object sender, EventArgs e)
{
    MessageBox.Show("Formulario Inicializado");
}
```

El ejemplo anterior utiliza el evento **Load** para mostrar un mensaje que avisa al usuario que el formulario ha sido cargado (**Figura 8**).
Como podemos notar, también en este código los eventos suelen recibir dos parámetros de entrada. El primero del tipo **object** llamado **sender** (emisor), define quién está provocando que el evento se dispare; en este caso, sería este formulario el que dispara el evento. Debido a que el tipo enviado es un **object**, si quisiéramos acceder al formulario contenido en él, deberíamos transformar el mismo en el mismo de formulario. Esto puede ser logrado fácilmente anteponiendo la forma **Form 1** al objeto **sender**.

Figura 8. *Después de inicializarse el formulario, el evento* ***Load*** *es disparado. Debido a esta acción, el código que maneja la ejecución del evento es disparado y le muestra un mensaje al usuario.*

El segundo parámetro del tipo **EventArgs** (argumentos del evento) representa un objeto que contiene información sobre el evento que se está disparando. Si un evento fuera provocado por el mouse, en este objeto podríamos ver la posición del mouse con relación al objeto con el que interactúa y cuáles botones fueron presionados, entre otras características. En otros casos, es posible anular la ejecución de eventos gracias a estos parámetros.

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    ...
}
```

En el código anterior, el último argumento del evento es del tipo **FormClosing EventArgs** (argumentos de cierre de formulario) que, al igual que **EventArgs**, representa los argumentos asociados al tipo de evento disparado. El evento **FormClosing** (formulario cerrándose) es disparado en el momento en que el usuario presiona el botón de cierre del formulario (el icono de equis en la parte superior derecha). En la **Figura 9**, podemos notar que la propiedad seleccionada llamada **Cancel** provee la funcionalidad de anular la ejecución de dicho evento, por lo tanto, si cancelásemos este evento, el formulario nunca se cerraría por más que el usuario presionara sobre el icono de cierre del formulario.

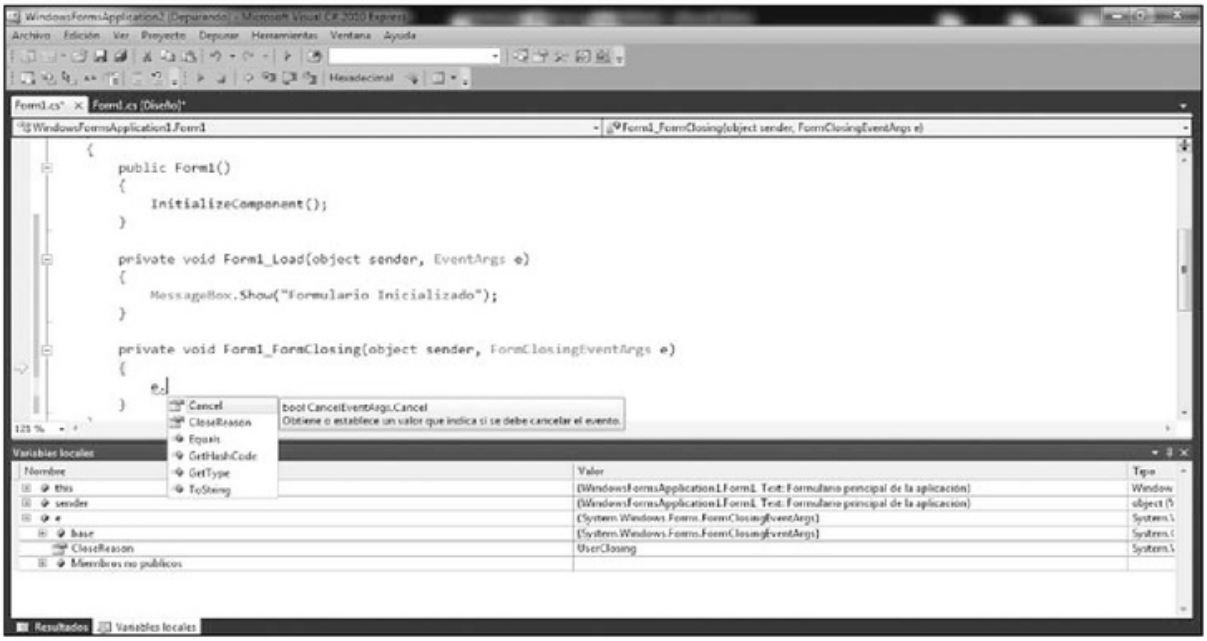


***Figura 9.** Los distintos objetos son inspeccionados desde el inspector de **Variables locales**. Aquí vemos cómo el objeto **sender** contiene nuestro formulario, y el objeto **e**, las propiedades del evento ejecutado.*

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

El código anterior prevendría el cierre del formulario por parte del usuario. Además de este, existe una amplia cantidad de eventos útiles relacionados con el formulario.

EVENTO	DESCRIPCIÓN
MouseClick	Se dispara cuando el botón izquierdo del mouse es presionado sobre el formulario.
ControlAdded	Cada vez que un nuevo control es adicionado al formulario se disparará este evento.
ControlRemoved	Si un control es removido del formulario este evento será activado.
CursorChanged	Si el icono del puntero del mouse es cambiado se disparará este evento.
MouseDoubleClick	Ocurre cuando se realiza doble clic sobre el formulario con el mouse.
FormClosed	Una vez que el formulario se ha cerrado por completo, este evento es disparado.
KeyDown	Si una tecla es presionada sobre el formulario y se mantiene pulsada, este evento es disparado.
KeyPress	Este evento se ejecuta cuando se presiona y se libera una tecla del teclado.
KeyUp	En el momento en que una tecla es liberada, se disparará este evento.
MouseEnter	Cuando el puntero del mouse ingresa en la zona del formulario, este evento es ejecutado.
MouseMove	Este evento se disparará cada vez que el mouse se mueva sobre el formulario.

***Tabla 2.** Los formularios poseen una amplia gama de eventos. En esta tabla, se muestran algunos de los más importantes.*

Comunicar información

Una tarea importante en las aplicaciones de escritorio es la de compartir información entre los distintos formularios. Supongamos que necesitamos mostrar un formulario con un conjunto de datos para que el usuario seleccione y, sobre la base de dicha selección, mostrar otro formulario dependiente de los datos del primero. Esto quiere decir que el segundo, en el momento de su creación, podría requerir de cierta información que solo el primero está en posición de facilitarle. Como primer paso, adicionaremos un nuevo formulario a nuestro proyecto (**Figura 10**).

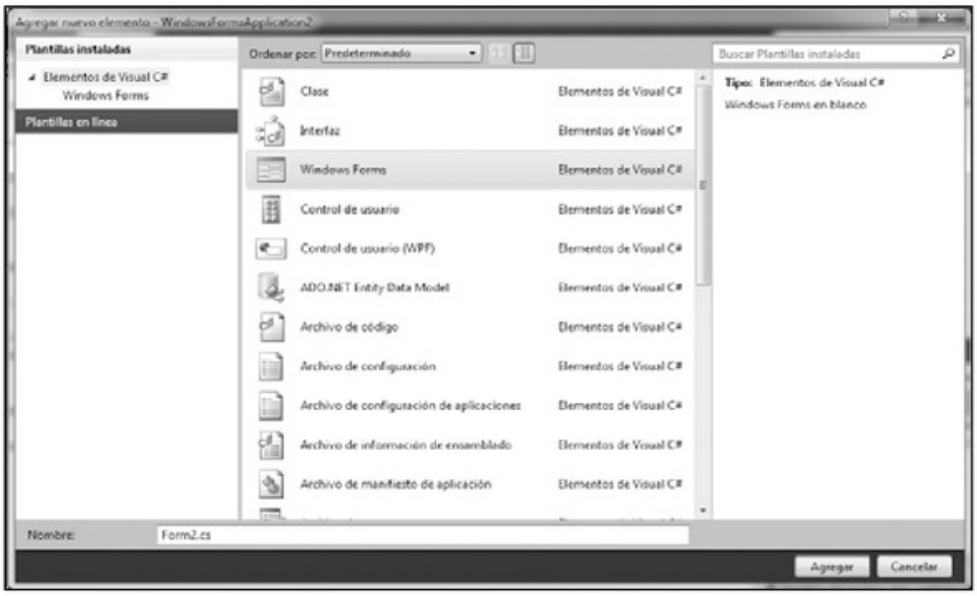


Figura 10. *Una aplicación de escritorio para Windows puede contener tantos formularios como sean necesarios. Estos formularios serán, en gran medida, los que contengan la lógica de la aplicación.*

Una de las posibilidades para obtener información desde otros formularios es mediante el uso de propiedades. Como hemos dicho, un formulario es una clase que hereda su funcionalidad de un tipo específico, por lo tanto, todas las reglas aplicables a clases se aplican a estos (como se puede ver en el código fuente a continuación).

```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
    }
   public string Mensaje { get; set; }
    public int Identificador { get; set; }
}
```

Como vemos en el código anterior, la declaración de las propiedades **Mensaje** e **Identificador** dentro de este formulario pueden ser utilizadas como medio para enviarle información en el momento de su creación. Así, desde el primer formulario, utilizaremos estas como puntos de contacto.

```
Form2 formulario = new Form2();
formulario.Mensaje = "Un mensaje";
formulario.Identificador = 10;
formulario.Show();
```

Con este código, hemos creado un nuevo objeto del segundo formulario adicionado al proyecto y asignado valores a sus propiedades. Como último paso, deberemos ejecutar la función **Show** (mostrar) para que el formulario aparezca en pantalla (**Figura 11**).

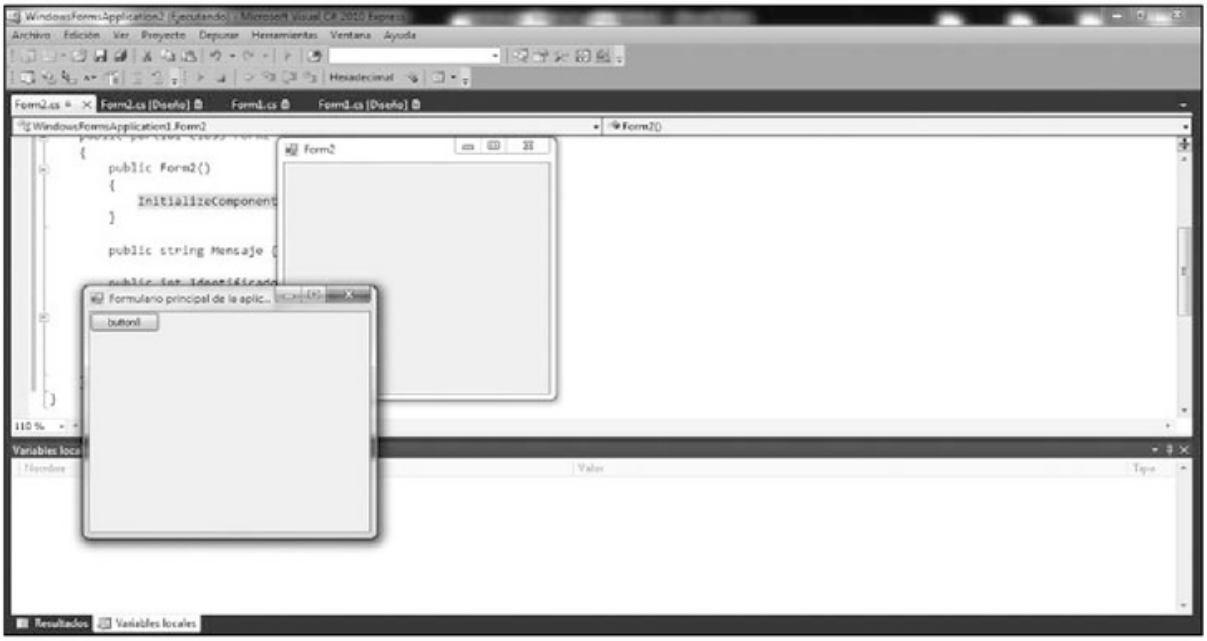


Figura 11. *Al utilizar la función Show del formulario, este es abierto como un nuevo componente de toda la aplicación, por lo tanto, será necesario cerrar todos los formularios para que nuestra aplicación deje de funcionar por completo.*

III FORMULARIOS MODALES

En los ejemplos solemos utilizar la línea de código **MessageBox.Show** para mostrar distintos mensajes al usuario. Este tipo de cuadro de alerta es un tipo de formulario modal, por lo tanto, al ejecutar este código el formulario que realiza la llamada quedará bloqueado hasta que el usuario presione sobre alguno de los botones mostrados en el nuevo mensaje y este sea cerrado.

Con esta técnica, creamos dos formularios independientes. Esto quiere decir que cada uno de ellos es parte de la aplicación, pero funcionan de forma independiente el uno del otro. Existen ocasiones en las cuales el flujo de nuestro programa requiere que el usuario complete o interactúe con información complementaria el formulario principal, no pudiendo acceder al formulario principal hasta no cerrar el nuevo formulario. Esta técnica es conocida por el nombre de **Formularios Modales**, donde el nuevo formulario es creado y visualizado dependiendo de otro. Esto hace que el formulario que crea el segundo quede bloqueado al usuario hasta que este no termine de interactuar con el formulario recién creado. Para poder crear un formulario de este tipo, es necesario cambiar ligeramente la forma en cómo se invoca el nuevo formulario.

```
Form2 formulario = new Form2();
...
...
formulario.ShowDialog(this);
```

En vez de utilizar la función **Show** como en el caso anterior, para mostrar un formulario modal necesitamos utilizar **ShowDialog** (mostrar como diálogo). Esta función requiere como parámetro una referencia al formulario que será el padre, o el formulario raíz, del nuevo formulario. Utilizamos **this** (este) como referencia ya que, al encontrarnos originalmente sobre el formulario número uno, su clase es una referencia a dicho formulario (**Figura 12**). Esto quiere decir que estaremos pasando la referencia del formulario **uno** hacia el formulario modal número **dos**.

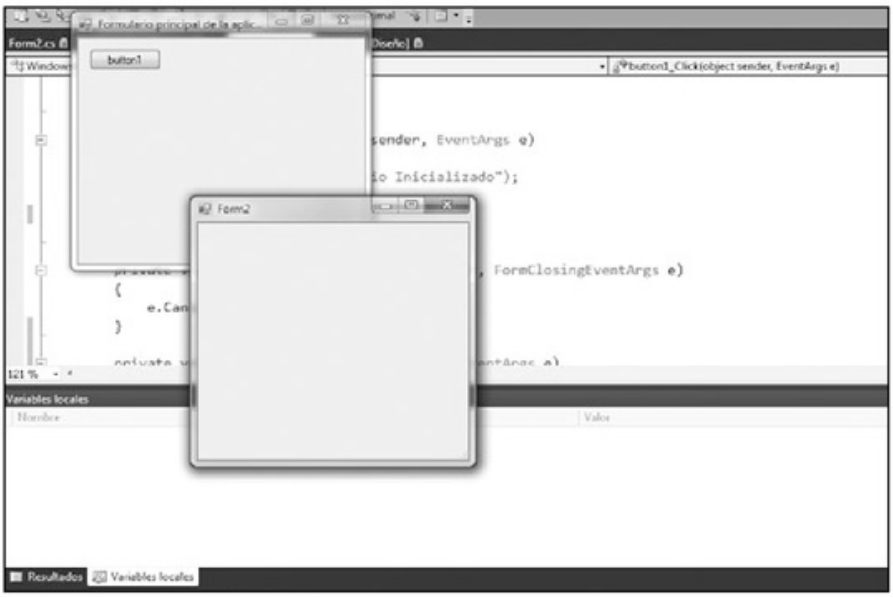


Figura 12. *Si presionamos con el mouse sobre el formulario que está por detrás del nuevo formulario abierto, notaremos que el que se encuentra adelante parpadeará alertándonos de que no es posible acceder al primero hasta cerrar el segundo.*

Formularios MDI

El último concepto sobre formularios son los formularios de tipo **MDI** (*Multiple document interface*, o en castellano, interfaz para múltiples documentos). Estos formularios tienen el objetivo de proveer una forma fácil de contener múltiples formularios con funcionalidad propia como si se tratase de documentos de un procesador de textos. En este, cada documento puede ser visualizado dentro de un contexto que engloba a todos los documentos abiertos como si se tratasen de objetos independientes. Para poder trabajar con formularios MDI, necesitaremos crear un contenedor especial, ver la **Figura 13** que se encuentra a continuación.

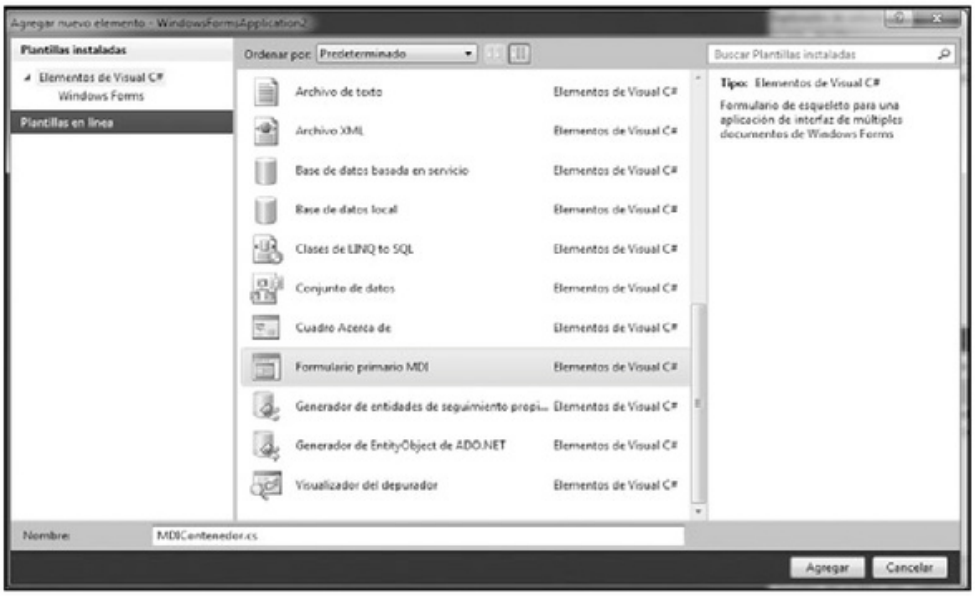


***Figura 13.** Al agregar un nuevo elemento a nuestro proyecto, en la lista de opciones deberemos elegir el tipo **Formulario primario MDI**.*

Una vez agregado el nuevo formulario, veremos que Visual C# 2010 Express además le adiciona una serie de controles con las funcionalidades básicas de una aplicación similar a un procesador de texto (**Figura 14**). Estos valores son tomados desde la plantilla seleccionada. Si este tipo de proyectos no se ajusta a nuestras necesidades, es posible modificar dicha plantilla dirigiéndonos al lugar donde hemos instalado Visual C# 2010 Express.

CONTROLES DE TERCEROS

Una empresa de renombre que provee controles y componentes para el desarrollo de aplicaciones es la empresa **Infragistics**. Esta compañía también posee una gran variedad de productos para potenciar nuestro desarrollo. Podemos visitar el sitio web de esta empresa en la siguiente dirección: **www.infragistics.com** y así facilitar el desarrollo de nuestras aplicaciones.

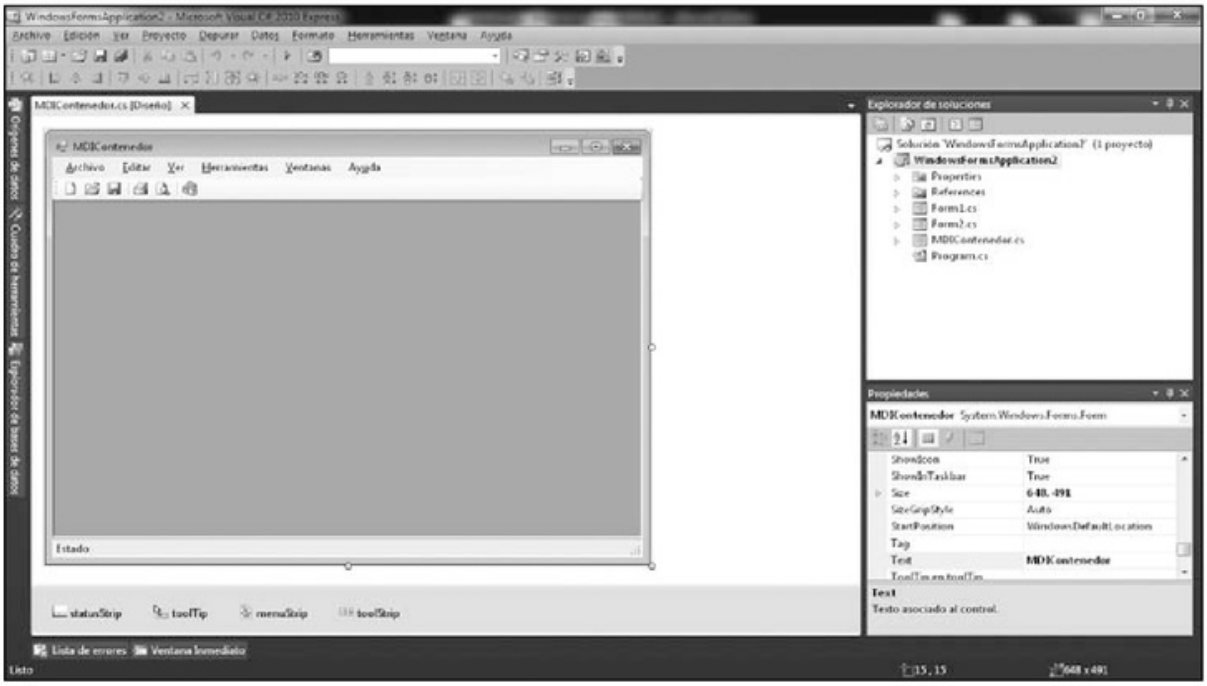


Figura 14. *Adicionar un formulario MDI a nuestro proyecto hace que Visual C# 2010 Express agregue un prototipo casi terminado similar a una aplicación para el procesamiento de texto.*

No es necesario que nos ajustemos a este diseño ya que simplemente ha sido creado de una plantilla incorporada en la interfaz de desarrollo. Si necesitamos realizar cualquier ajuste, solo basta con que eliminemos los controles adicionados en forma automática y colocar los nuestros. Para entender el concepto de estos formularios, los utilizaremos como han sido provistos por la herramienta de desarrollo.

```
Form1 formulario = new Form1();
formulario.MdiParent = this;
formulario.Show();
```

De forma similar a los formularios modales, los formularios que serán contenidos por contenedores **MDI** poseen una propiedad llamada **MdiParent**. Esta propiedad

III PLANTILLAS

Cada vez que adicionamos un nuevo proyecto o conjunto de archivos a nuestros proyectos son creadas una serie de líneas de código por defecto, así como diferentes archivos. Estas plantillas son, en sí, proyectos de Visual Studio mismos que pueden ser modificados desde las carpetas de instalación del IDE de desarrollo.

especifica cuál será el contenedor del nuevo formulario una vez mostrado mediante la función **Show**. El código presentado es escrito en la clase del formulario MDI contenedor en el evento que es ejecutado mediante el menú **Nuevo** (**Figura 15**).

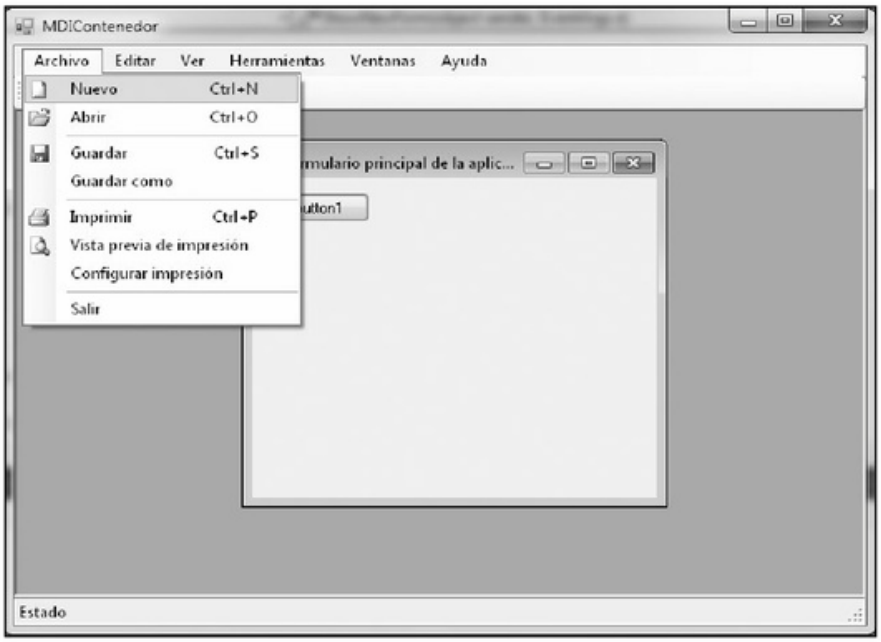


***Figura 15.** Al presionar sobre el menú **Nuevo**, el formulario es creado y mostrado dentro del contenedor. Si presionamos otra vez sobre este menú, otro formulario con esta funcionalidad será creado.*

Controles para Windows

Existen una gran cantidad de controles para el desarrollo de aplicaciones para escritorio, no solo los provistos por Microsoft .Net, sino también por terceros, los cuales podemos adquirir para obtener otras funcionalidades dentro de las aplicaciones que construyamos. En esta sección, describiremos algunos de los principales controles y su funcionalidad, cómo interactuar con ellos y las reglas comunes que se aplican a la mayoría. En la **Tabla 3**, podemos ver algunas de las propiedades comunes de todos los controles para aplicaciones de escritorio. Estas propiedades son heredadas del tipo base Control, ya que todos los controles usan este tipo para su definición.

HILOS DE EJECUCIÓN

Normalmente la interfaz visual de nuestros formularios en las aplicaciones de escritorio trabajan bajo el mismo hilo de ejecución que el código de la aplicación. Por tal motivo los procesos extensos como la interacción con bucles o recuperación de datos de la base de datos puede bloquear la interfaz. En lo posible deberemos usar nuevos hilos para estos procesos extensos.

PROPIEDAD	DESCRIPCIÓN
Text	Recupera y asigna un valor de texto, el cual puede ser visualizado por el control.
Size	Especifica el tamaño del control.
Anchor	Fija el control a una posición determinada.
Location	Coloca al control en una posición de X e Y pixeles desde la parte superior izquierda de su contenedor.
Dock	Hace que el control ocupe toda la zona especificada. Puede ser colocado en la parte superior, derecha, inferior, izquierda o central de su contenedor.
Enabled	Habilita o deshabilita el control.
Click	Evento que se dispara cuando el control es presionado con el mouse.
Enter	Cuando el control obtiene el foco de acción, este evento es disparado.
Leave	Cuando el foco pasa del control que lo tenía a otro, este evento se dispara para el primero.
TabIndex	Especifica el orden en el cual se hará foco sobre el control cuando el usuario presione la tecla Tab.

Tabla 3. *Todos los controles presentan propiedades y eventos en común que podemos utilizar en forma independiente del tipo de control con el que estemos trabajando.*

Control Button

El control **Button** (botón) es uno de los grupos de controles más comúnmente usados, utilizado para la ejecución de acciones por parte del usuario. Aceptar cuadros de diálogos, guardar o cargar información, entre otros. Si bien podemos definir el control de forma visual, todas sus propiedades pueden ser escritas desde el código.

```
this.button1.Location = new System.Drawing.Point(94, 104);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "Click aquí";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Vemos cómo se define el botón que será mostrado en el formulario. Desde su posición mediante el uso de **Location**, su tamaño mediante la propiedad **Size**, el texto por mostrar con la propiedad **Text** y el evento **Click**, el que se disparará cuando el usuario presione sobre él. Con este último evento definido, estamos en posición de escribir código que se ejecute en el momento en que el botón es presionado.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Botón presionado");
}
```

En el código anterior, dentro de la función que se ejecutará cuando el botón sea presionado, mostramos un mensaje de alerta al usuario (**Figura 16**).

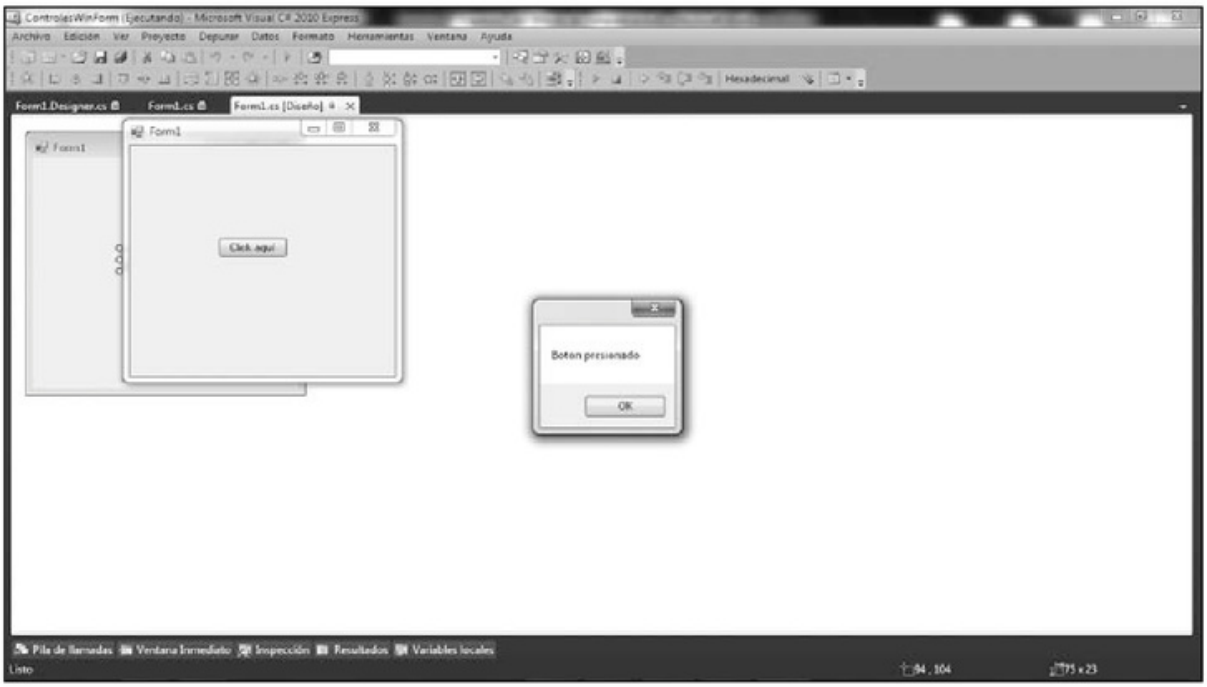


Figura 16. *El evento Click del botón, al estar asociado a una función en nuestro código, hace que esta se ejecute en el momento en que el usuario presiona sobre el control.*

Control TextBox

Las cajas de texto o **TextBox** son los recuadros en las aplicaciones, que le permiten al usuario ingresar texto mediante el uso del teclado. Estas poseen las mismas características listadas en la **Tabla 3**, pero agrega algunas particulares.

```
this.textBox1 = new System.Windows.Forms.TextBox();
this.textBox2 = new System.Windows.Forms.TextBox();
this.textBox3 = new System.Windows.Forms.TextBox();

//Caja de texto simple
this.textBox1.Name = "textBox1";
...
...

//Caja de texto multi línea
this.textBox2.Multiline = true;
this.textBox2.ScrollBars = System.Windows.Forms.ScrollBars.Both;
this.textBox1.Name = "textBox2";
...
```

```
...

//Caja de texto para contraseñas
this.textBox3.Name = "textBox3";
this.textBox3.UseSystemPasswordChar = true;
...
...
```

En el código anterior, declaramos tres cajas de texto. Las tres son del tipo **TextBox**, pero modifican sus propiedades para formar tres tipos de caja de texto diferentes. La primera de ellas es una caja de texto común, para la entrada de texto en una sola línea. La segunda es una caja de texto para el ingreso de múltiples líneas de texto. La propiedad **Multiline** (múltiples líneas) es la que habilita esta cualidad. Junto a la propiedad **ScrollBars** (barras de desplazamiento), nos permite agregar barras de desplazamiento para que el usuario pueda navegar por el texto introducido. Para configurar esta propiedad, utilizamos una enumeración que especificará qué tipo de barras de desplazamientos se mostrarán; podemos elegir entre **None** (ninguna), **Horizontal**, **Vertical** y **Both** (ambas barras). Por último, la tercera caja de texto es una que permite el ingreso de una sola línea de texto, pero para la introducción de contraseñas. La propiedad **UseSystemPasswordChar** (usar el carácter de contraseña provisto por el sistema) hace que la caja de texto muestre círculos que representan cada carácter que el usuario haya ingresado en ella (**Figura 17**).

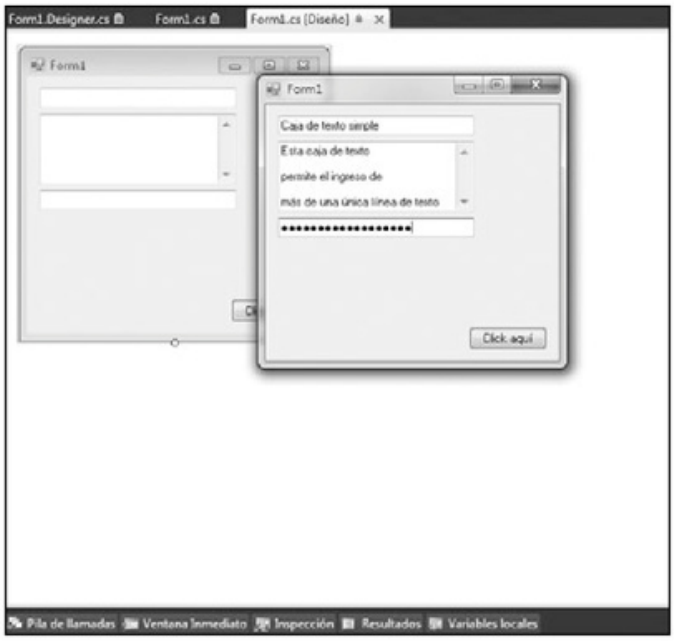


Figura 17. *Las tres cajas de texto son creadas desde este tipo TextBox, pero modificando sus propiedades para obtener comportamientos diferentes en cada una de ellas.*

Sin importar las propiedades que configuremos en las cajas de texto, para obtener el valor introducido por el usuario tenemos que usar la propiedad **Text**; si necesitamos enviar información a la caja de texto, tendremos que usar dicha propiedad.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Su contraseña: " + this.textBox3.Text);
}
```

Este código toma el valor de la propiedad **Text** de la caja de texto configurada para comportarse como caja de contraseñas y lo muestra como un mensaje (**Figura 18**).

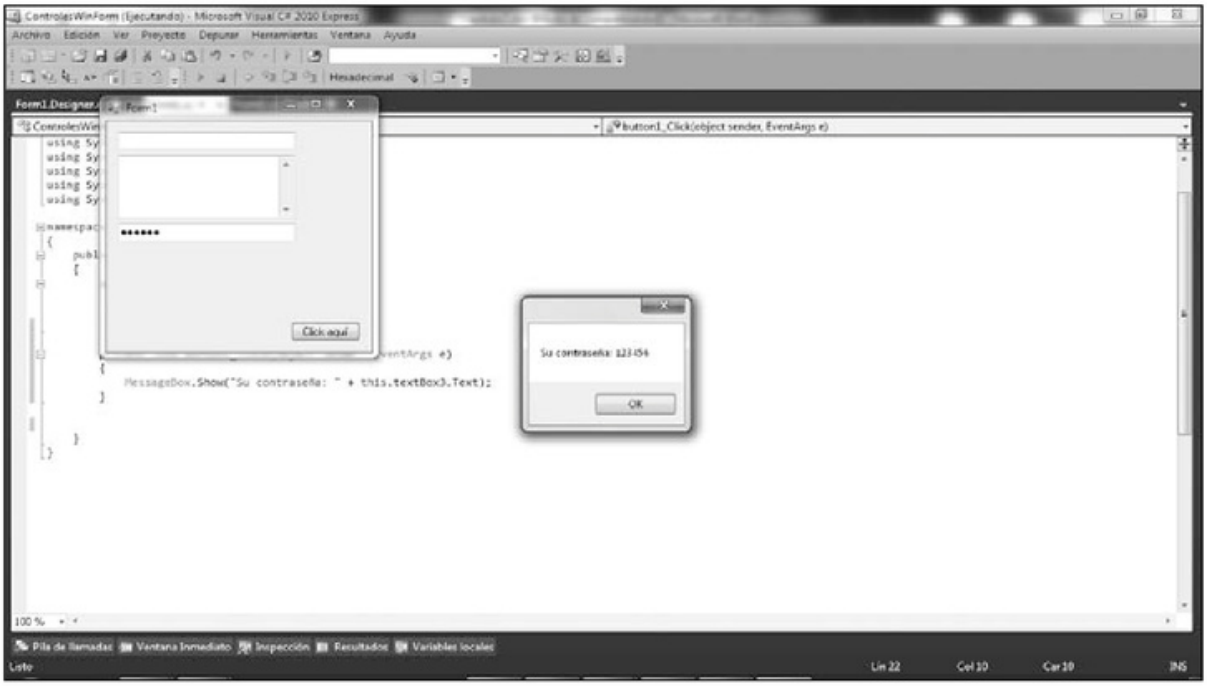


Figura 18. *A pesar de que la caja de texto con formato de contraseña no muestra los caracteres introducidos por el usuario, el texto es almacenado en forma correcta y puede ser leído desde el código.*

Control Label

En las aplicaciones, casi siempre se muestran descripciones al lado de los distintos controles, así como mensajes de acción o texto descriptivo general. Esto se consigue mediante el uso del control **Label** (etiqueta); en este control, la propiedad **Text** es utilizada para configurar el texto por mostrar.

```
this.label1.Text = "Caja de texto simple";
...
```

```
...

this.label2.Text = "Caja de texto para muchas líneas";
...
...

this.label3.Text = "Caja de texto para contraseñas";
...
...
```

La configuración creada con el código anterior se puede observar en la **Figura 19**.

Figura 19. *Las etiquetas son controles simples para mostrar textos descriptivos. Por lo general, se asocia una etiqueta a un control con el cual el usuario puede realizar tareas determinadas.*

Control ComboBox y ListBox

Estos dos controles, **ComboBox** y **ListBox**, poseen funcionalidades similares; sus configuraciones son también similares, solo varía la forma en cómo muestran los datos. Mientras el primero presenta una serie de elementos que aparecen en una lista desplegada por el usuario, el segundo muestra estos elementos ya desplegados en una caja de selección. El uso principal de estos dos controles radica en el aprovechamiento de espacio dentro de la interfaz del formulario: el segundo requiere de mayor espacio para desplegar los elementos, mientras que el primero los muestra temporalmente en una caja desplegable que luego de la selección desaparece. Para adicionar elementos dentro de este control, deberemos pasarle una colección de ítems.

```
public class Item
{
    public int Id { get; set; }
```

```
    public string Valor { get; set; }
}

...
...

List<Item> items = new List<Item>();

items.Add(new Item()
{
    Id = 1,
    Valor = "Item 1"
});

items.Add(new Item()
{
    Id = 2,
    Valor = "Item 2"
});

...
...

this.comboBox1.DataSource = items;
this.listBox1.DataSource = items;
```

Como vemos en el código anterior, hemos creado una clase **Item** que contiene dos propiedades. Luego, creamos y llenamos una lista de ítems del tipo de nuestra clase. Mediante la propiedad **DataSource** del **ComboBox** y el **ListBox**, asignamos la lista para que sea mostrada. Con esta asignación, es necesario que especifiquemos, para los dos controles, cuál será la propiedad de nuestra clase que servirá para mostrar el texto que el usuario puede seleccionar; adicionalmente, podremos especificar una propiedad que servirá como identificador del ítem seleccionado.

```
this.comboBox1.ValueMember = "Id";
this.comboBox1.DisplayMember = "Valor";

this.listBox1.ValueMember = "Id";
this.listBox1.DisplayMember = "Valor";
```

La propiedad **ValueMember** (miembro para el valor) especifica cuál será la propiedad de la colección de datos pasada al control que se utilizará para representar el valor del ítem seleccionado. La propiedad **DisplayMember** (miembro para mostrar) es utilizada para determinar la propiedad de la colección de datos que se utilizará para mostrar el texto en el control que el usuario puede ver y seleccionar (**Figura 20**).

***Figura 20.** Con los valores correctamente configurados y la colección de datos cargada, ambos controles muestran la información que les hemos pasado y asociado.*

Para conocer cuál ítem ha sido seleccionado, deberemos utilizar la propiedad **SelectedItem** (ítem seleccionado), como vemos en el código fuente a continuación.

```
private void button2_Click(object sender, EventArgs e)
{
    Item itemSeleccionado = this.comboBox1.SelectedItem as Item;
    Item itemSeleccionado2 = this.listBox1.SelectedItem as Item;

    MessageBox.Show("Item seleccionado en ComboBox: " +
itemSeleccionado.Valor + "\n" +
        "Item seleccionado en ListBox: " + itemSeleccionado2.Valor);
}
```

III CONVERTIDOR AS

Muchas veces, obtendremos como parámetros de salida las funciones tipos **object**, lo que quiere decir que podrían contener en su interior un tipo distinto. Usando la palabra reservada **AS**, podemos intentar convertir este objeto en el tipo que esperamos o creemos que contiene. Si la conversión es exitosa, obtendremos nuestro objeto, si no, un valor nulo.

El código anterior obtiene el elemento seleccionado de ambos controles mediante la propiedad llamada **Valor** uniéndolos en una única cadena de texto para luego mostralos en un único mensaje en la pantalla del usuario.

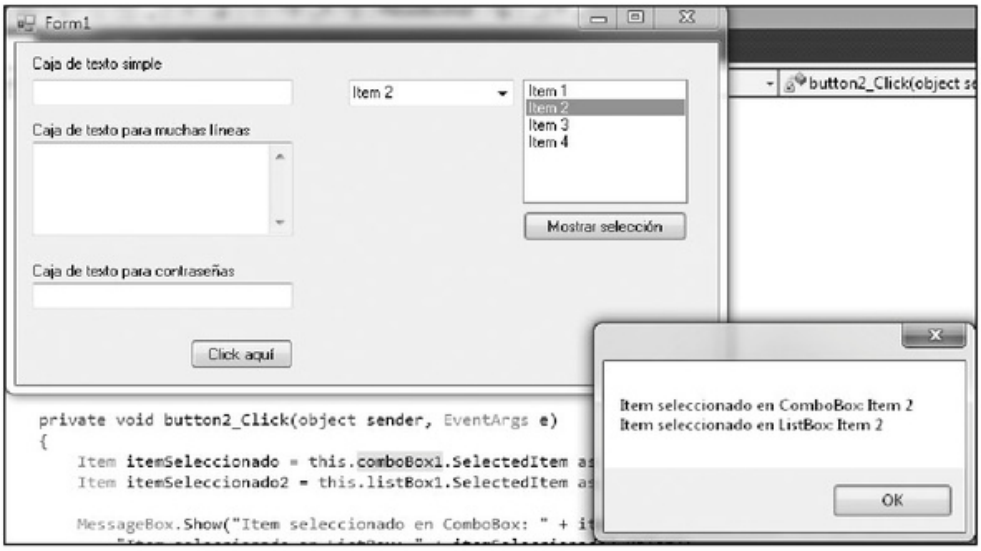


***Figura 21.** La función SelectedItem de los controles ListBox y ComboBox retorna un tipo object. Es necesario transformar este objeto al tipo de dato que le hemos pasado inicialmente para poder leer la información.*

Control CheckBox y RadioButton

El último conjunto de controles son los **CheckBox** (cajas de verificación) y los **RadioButton** (botones de selección circular). El primero de estos dos sirve para marcar una opción específica de una lista de opciones. Esta caja de verificación podrá contener dos estados, uno seleccionado (**True** o **verdadero**) y otro no seleccionado (**False** o **falso**), aunque es posible hacer que estos posean un estado intermedio mediante la configuración de la propiedad **ThreeState** (tres estados) (**Figura 22**). Esto le dará al usuario la propiedad de elegir entre **seleccionado**, **no seleccionado** y **parcialmente seleccionado**.

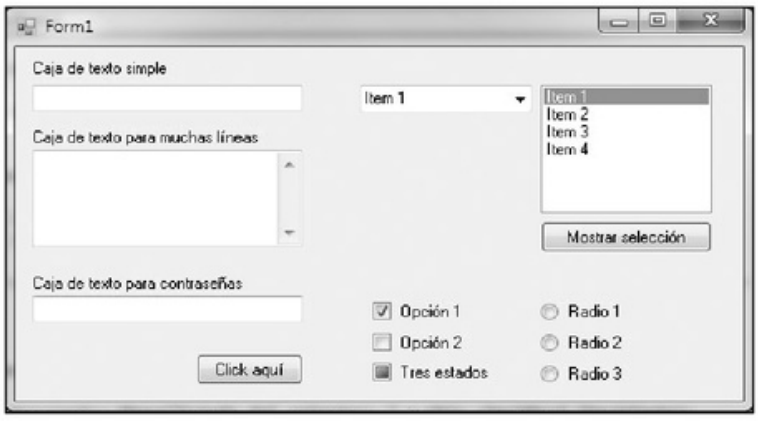


***Figura 22.** Las tres cajas de verificación creadas en el formulario muestran los tres estados posibles que estos controles pueden obtener.*

```
private void checkBox3_CheckedChanged(object sender, EventArgs e)
{
    MessageBox.Show("Estado del selector: " + this.checkBox3.CheckState);
}
```

En el código anterior, vemos que, en el momento en que el estado del control cambia, podemos verificar el valor del mismo por un lado capturando el evento de cambio de estado y, por otro, mediante la propiedad **Checkstate** (Estado de verificación).

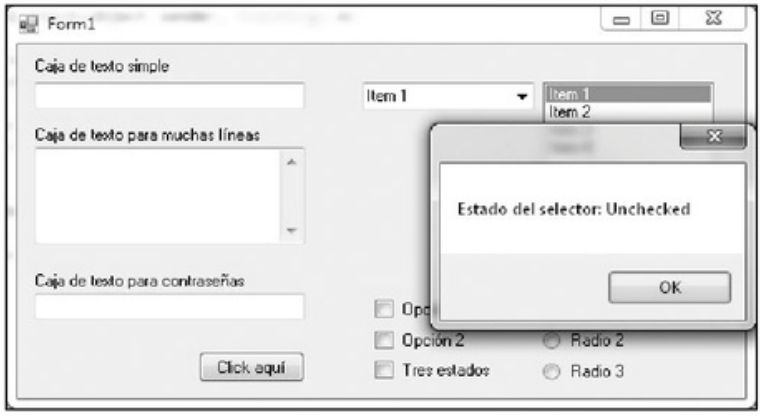


***Figura 23.** Cada vez que el usuario presiona sobre la caja de verificación, su estado se muestra en un mensaje. Esta caja está configurada para poder mantener tres estados.*

Las cajas de verificación son independientes unas de las otras; esto quiere decir que podemos cambiar su estado sin afectar las otras presentes en este formulario. Esto resulta diferente en el caso de los botones circulares, ya que estos trabajan en conjunto, lo que quiere decir que todos los botones circulares que coloquemos dentro del mismo formulario, esto es, dentro del mismo contenedor, trabajarán en conjunto haciendo que solo podamos mantener seleccionado uno de los controles por vez, como vemos en el código fuente que se encuentra a continuación.

```
if (this.radioButton1.Checked)
{
    MessageBox.Show("RadioButton 1 seleccionado");
}

if (this.radioButton2.Checked)
{
    MessageBox.Show("RadioButton 2 seleccionado");
}
```

```
if (this.radioButton3.Checked)
{
    MessageBox.Show("RadioButton 3 seleccionado");
}
```

En el código, verificamos cuál es el control que está seleccionado. Como solo puede haber uno seleccionado cada vez dentro de un contenedor, estaremos seguros de que, al estar seleccionado cualquiera de los controles, los demás no lo estarán (**Figura 24**).

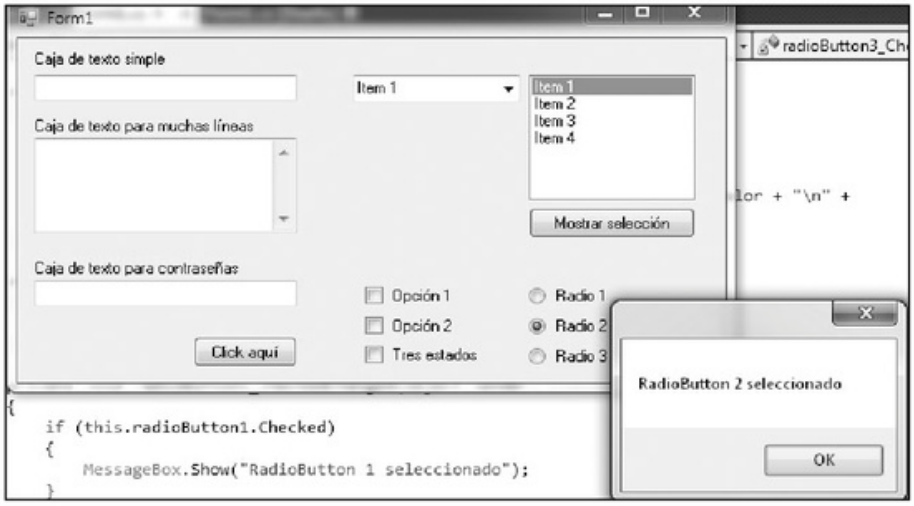


***Figura 24.** Cuando uno de los controles es seleccionado, se verifica cuál de ellos es mediante un mensaje personalizado para destacar la acción.*

Controles personalizados

Ya hemos visto, en este capítulo, los principales controles provistos por Microsoft .Net para el desarrollo de aplicaciones para escritorio. De cualquier manera, en muchos casos nos veremos con la necesidad de crear nuestros propios controles, con funcionalidad personalizada; esto sucede cuando los controles estándares no nos provean de lo que necesitemos y tengamos que extender alguna funcionalidad a un control ya existente, o bien porque no exista un control que realice aquello que estemos buscando.
Existen tres tipos de controles que podemos crear: **controles de usuario**, **controles compuestos** y **controles personalizados avanzados**; estos últimos resultan ser los más complejos y requieren mayor conocimiento para su confección.

Controles de usuario

Los **controles de usuario** son controles de fácil creación. Estos, además de requerir código de nuestra parte, poseen una interfaz visual donde poder diagramar el control y conocer de antemano cómo se visualizará. Para crear un control de usuario, utilizamos otros controles ya existentes para acomodarlos dentro del control

que estamos creando. Estos controles son utilizados preferentemente en situaciones en las que un conjunto de funcionalidades visuales dentro de nuestra aplicación tiende a repetirse en diferentes lugares.

Pensemos en una funcionalidad para filtrar elementos de una grilla. Estas grillas podrían estar en diferentes partes de la aplicación y requerirían ser filtradas por este conjunto de opciones representadas por controles. Podríamos encapsular todos estos controles en un único control de usuario, y así poder copiarlo y pegarlo en cada uno de nuestros formularios.

En la **Figura 25**, podemos ver cómo agregar un nuevo control de usuario a nuestra aplicación. Una vez adicionado el control de usuario, nos encontraremos con un lienzo donde poder agregar otros controles, como el que vemos en la **Figura 26**. El control de usuario, además, contiene una serie de propiedades y eventos que es posible utilizar para definir su comportamiento visual. Si bien podemos utilizar estas propiedades desde el código que creemos en el nuevo control mediante la sintáxis **this.[Propiedad]**, también será posible sobre escribir las propiedades por defecto para adicionar funcionalidad a las mismas.

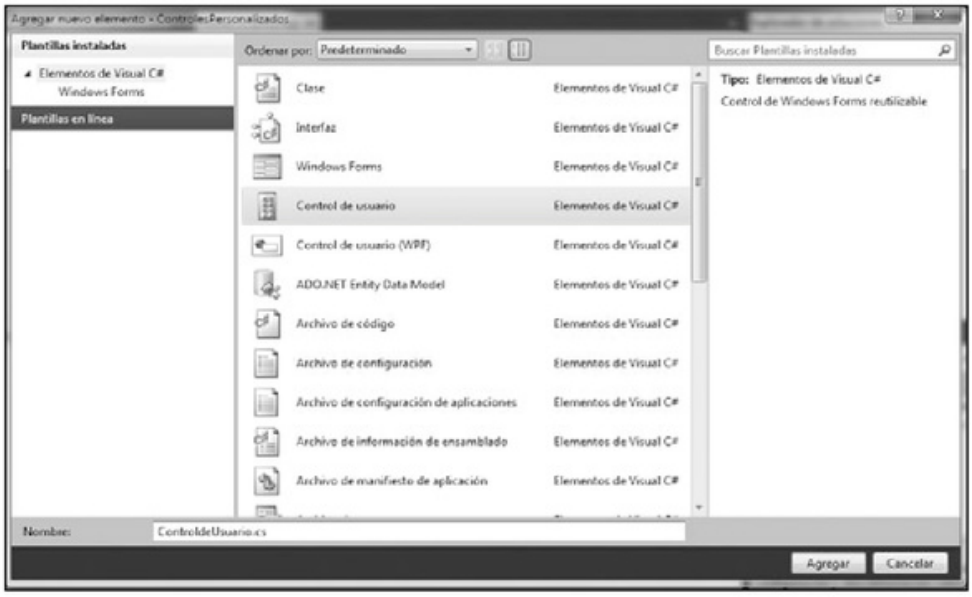


***Figura 25.** Al agregar un nuevo archivo a la solución, podemos elegir el tipo Control de usuario. Este nos servirá como contenedor para diagramar nuestro control.*

CONTROLES DE TERCEROS

Entre los distintos proveedores de controles y componentes para el desarrollo de aplicaciones, podemos encontrar a la empresa **Component One**. Esta empresa provee una gran variedad de productos para potenciar nuestro desarrollo. Podemos visitar su sitio web en la siguiente dirección: **www.componentone.com/**.

PROPIEDAD	DESCRIPCIÓN
AllowDrop	En acciones de arrastrar y soltar otros objetos del formulario o de Windows, especifica si puede aceptar este tipo de acciones.
AutoScroll	Si el contenido del control de usuario supera sus dimensiones, se mostrarán barras de desplazamiento para que el usuario pueda visualizar todo el contenido.
BackColor	Especifica el color de fondo del control de usuario.
BackgroundImage	Especifica una imagen para ser usada en el fondo del control de usuario.
BorderStyle	Representa el tipo de borde que mostrará el control de usuario, si es que hay alguno.
Enabled	Especifica si el control de usuario se encuentra activo o no y si el usuario podrá interactuar con este o no.

***Tabla 4.** Las propiedades de los controles de usuario son comunes a las de otros controles, pero poseen algunas particulares útiles.*

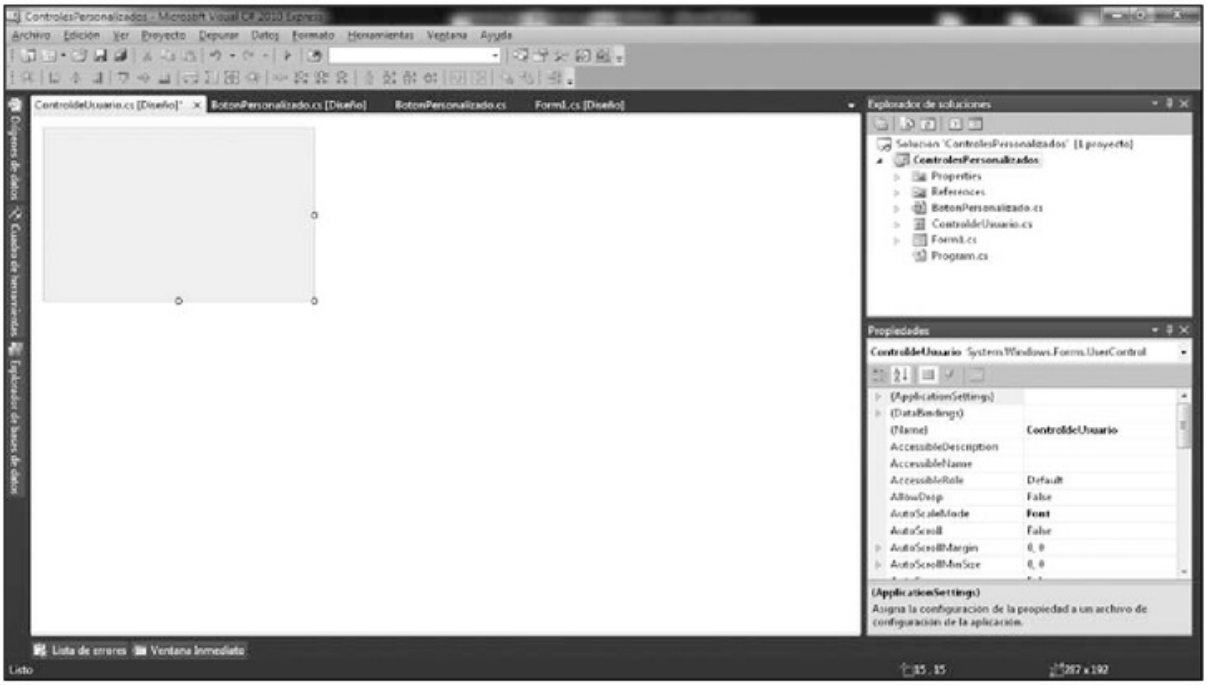


***Figura 26.** El lienzo de un control de usuario es similar al de un formulario, aunque este necesitará del primero para poder ser visualizado y contenido. Un control de usuario no puede ser mostrado por sí solo en el escritorio.*

Cuando agregamos este tipo de controles a nuestro proyecto, notaremos que, en el cuadro de herramientas, la barra lateral de la izquierda donde son contenidos los demás controles como los botones, cajas de textos y otros. Ahora aparece el control para poder ser adicionado a los diferentes formularios (**Figura 27**) y que este se cree en el momento de la ejecución de la aplicación. Además de esto, es posible adicionar el control mediante líneas de código de la siguiente forma.

```
public partial class Form1 : Form
{
```

```
    public Form1()
    {
        InitializeComponent();
    }

     //Evento Load del formulario principal
    private void Form1_Load(object sender, EventArgs e)
    {
        ControldeUsuario miControl = new ControldeUsuario();
        this.Controls.Add(miControl);
    }
}
```

Figura 27. *En el Cuadro de herramientas, además de los controles y componentes provistos por Microsoft .Net, ahora también aparece el control de usuario que hemos adicionado al proyecto. Podremos agregarlo a nuestro formulario.*

En el lienzo del control de usuario adicionado al proyecto, agregaremos dos controles de **etiquetas**, una **caja de texto**, una **lista desplegable** y un **botón**. Con estos, formaremos una representación de filtros para nuestra grilla (**Figura 28**).

Figura 28. *Si agregamos los distintos controles al lienzo del control de usuario, creamos su diseño de forma visual. Este control de usuario será utilizado como filtro para datos.*

En el código que maneja este control de usuario, agregaremos algunas líneas para cargar inicialmente la lista desplegable, así como la creación de un evento que le avisará al usuario que el botón de filtro ha sido presionado; de esta forma, se podrá realizar el filtro desde el formulario que contiene los datos.

```
public delegate void ManejadorDeFiltro(string texto, int
     elementoSeleccionado);
public event ManejadorDeFiltro FiltroPresionado;

private void BotonFiltro_Click(object sender, EventArgs e)
{
     if (FiltroPresionado != null)
     {
          FiltroPresionado(this.textoFiltro1.Text,
               this.combo1.SelectedIndex);
     }
}
```

En el código anterior, hemos creado un evento que servirá para que el programador sepa en qué momento se ha presionado el botón de filtro. Este evento le enviará como parámetros al desarrollador los valores contenidos en la caja de texto y el valor seleccionado de la lista desplegable. Por último, agregaremos algunos valores a la lista desplegable de la siguiente forma.

```
private void ControldeUsuario_Load(object sender, EventArgs e)
{
     this.combo1.Items.Add(new
     {
          Nombre = "Igual que",
          Valor = "0"
     });

     this.combo1.Items.Add(new
     {
          Nombre = "Distinto que",
          Valor = "1"
     });

     this.combo1.Items.Add(new
```

```
    {
        Nombre = "Contiene",
        Valor = "2"
    });

    this.combo1.DisplayMember = "Nombre";
    this.combo1.ValueMember = "Valor";
}
```

Como vemos, para llenar la lista desplegable utilizamos un **tipo anónimo**. Este tipo no posee una representación física dentro de una clase creada por nosotros, sino que es delimitado en el momento de su creación. Como podemos ver en la **Figura 29**, la lista desplegable contiene los valores establecidos en el código dándonos la seguridad de que el **tipo anónimo** no ha alterado el resultado esperado.

***Figura 29.** Los tipos anónimos pueden ser útiles para llenar con rapidez un control con datos. Estos tipos no requieren estar especificados en una clase para poder ser utilizados.*

El siguiente paso es adicionar al formulario un control tipo **grilla** para mostrar los datos que luego podremos filtrar. Junto con este control, asociaremos el evento del **control de usuario** al código de ejecución del formulario que lo contiene (**Figura 30**).

III ENLAZADO DE DATOS

La mayoría de los controles pueden ser enlazados a fuentes de datos. Esta fuente de datos debe ser una colección de registros en el formato de filas y columnas. Si interactuamos con el control y obtenemos la fuente de datos enlazada podremos recuperar la misma colección que hemos pasado originalmente, incluso aquellos valores no visibles en el control.

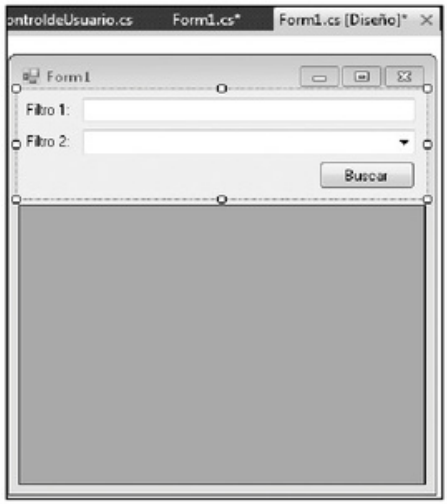


Figura 30. *El evento creado dentro del* ***control de usuario*** *se muestra junto a la lista de eventos estándares del* ***control de usuario.*** *Este evento forma parte del control y puede ser manejado de igual forma que los demás eventos.*

```
private void controldeUsuario1_FiltroPresionado(string texto, int
     elementoSeleccionado)
{
     List<Usuario> usuarios = this.dataGridView1.DataSource as
      List<Usuario>;

     switch (elementoSeleccionado)
     {
          case 0:
               var igual = (from c in usuarios
                            where c.Nombre.Equals(texto)
                            select c).ToList<Usuario>();
               this.dataGridView1.DataSource = igual;
               break;
          case 1:
               var noIgual = (from c in usuarios
                            where c.Nombre != texto
                            select c).ToList<Usuario>();
               this.dataGridView1.DataSource = noIgual;
               break;
          case 2:
               var contiene = (from c in usuarios
                            where c.Nombre.Contains(texto)
                            select c).ToList<Usuario>();
               this.dataGridView1.DataSource = contiene;
```

```
                break;
            default:
                break;
        }
    }
```

En el código anterior, implementamos un filtro rudimentario sobre la base de la selección del usuario. Cuando el control de usuario dispara su evento, este es capturado y procesado por el código del formulario. Mediante el uso de **LinQ**, tomamos los valores almacenados en la grilla y los filtramos. Por último, al aplicar uno de los filtros, los datos son regenerados y solo se muestran aquellos que cumplen con la condición dada (**Figura 31**).

***Figura 31.** Los registros en la grilla son filtrados sobre la base de la selección del usuario. La capacidad de un filtro de datos dependerá de la complejidad con la que lo tratemos; aquí solo vemos una forma rudimentaria de filtrar datos.*

El **control de usuario** que hemos creado puede ser utilizado en tantos formularios como necesitemos. Su funcionalidad será replicada en todas las copias que tengamos, por lo tanto, podemos encapsular este funcionamiento y transportarlo a los distintos formularios en los cuales lo utilicemos. De esta forma, podríamos tener otras grillas que muestren otro tipo de datos, y el filtro se aplicará a otras columnas de la grilla. Esto dependerá en forma directa de nuestras necesidades dentro de la aplicación.

Controles compuestos

Los **controles compuestos** a diferencia de los controles de usuario no cuentan con una interfaz visual para diseñar el control. Otra diferencia fundamental es que estos controles se crean **a partir de otro control ya existente**. Esto quiere decir que, para poder crear un control compuesto, es necesario heredar la funcionalidad de un control ya existente extendiendo sus funcionalidades por medio del nuevo control.

```
class Controlcompuesto
    : Button
{
    protected override void OnMouseHover(EventArgs e)
    {
        base.OnMouseHover(e);
        this.Font = new Font(FontFamily.GenericSerif,
            15, FontStyle.Regular);
    }
    protected override void OnMouseLeave(EventArgs e)
    {
        base.OnMouseLeave(e);
        this.Font = new Font(FontFamily.GenericSerif,
            12, FontStyle.Italic);
    }
}
```

Lo primero que debemos notar en el código anterior es que nuestra clase **Controlcompuesto** hereda de la clase **Button** (botón). La clase **Button** es la que representa la funcionalidad de los controles de tipo botón que utilizamos normalmente en nuestras aplicaciones, por lo tanto, la clase **Controlcompuesto** se transformará en un botón que nos da la posibilidad de sobrescribir o adicionar funcionalidad al tipo botón predeterminado. Este tipo de control también es mostrado en el **Cuadro de herramientas** como un control de usuario, así como es dibujado como un botón en el formulario debido a que su funcionalidad es la de este tipo de control (**Figura 32**).

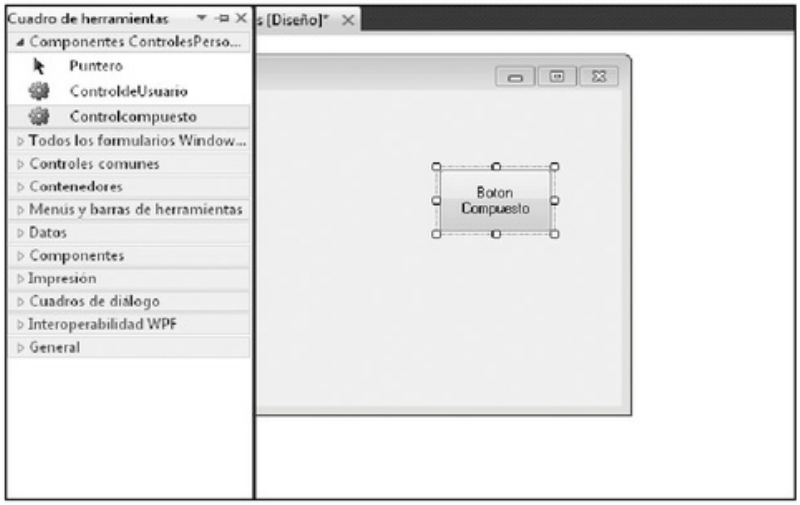


Figura 32. *Un control compuesto se mostrará como el tipo del cual hereda su funcionalidad. Si realizamos modificaciones significativas en este, podríamos conseguir que cambiara por completo su apariencia visual.*

En el ejemplo que ya hemos visto, además de la herencia, vemos que hemos sobrescrito dos funcionalidades del botón. La primera se ejecuta cuando el mouse pasa por **arriba del botón**, y la segunda, cuando **sale del botón**. En cada caso, el nuevo botón aparece con cambios en el tipo, el tamaño y el estilo de la fuente (**Figura 33**).

Figura 33. *Al colocar el mouse sobre el botón, el evento OnMouseHover se dispara. En su interior, el botón implementa su lógica para este evento.*

Las modificaciones que podemos realizar no se remiten solo a cambios de color cuando ocurren determinados eventos. Este tipo de controles nos permite realizar cualquier tipo de modificación y así personalizar nuestros controles de acuerdo con las necesidades. Veamos el siguiente código.

```
protected override void OnPaint(PaintEventArgs pevent)
{
    base.OnPaint(pevent);
    Image imagen = Bitmap.FromFile("big-smile-icon.png");
    pevent.Graphics.DrawImage(imagen, new RectangleF(1, 1, 100, 100));
}
```

DIBUJANDO CONTROLES

Las empresas que crean y venden de controles personalizados utilizan controles como botones o cajas de texto como base de sus controles y modifican algunas funciones para presentar un nuevo control. Por otro lado, controles más avanzados o vistosos deben ser codificados, su funcionalidad, por completo. Incluso dibujando cada pieza del mismo en el momento de su ejecución.

La función **OnPaint** (al pintarse) sobrescrita en el código anterior nos permite especificar cómo debe dibujarse el control que estamos creando, por lo tanto, podemos tomar los valores por defecto y agregar mayor funcionalidad. En el caso del código, leemos una imagen desde el disco duro de la computadora y la utilizamos para pintarla en el margen izquierdo de este (**Figura 34**).

***Figura 34.** La versatilidad de los controles compuestos permite modificar el comportamiento tradicional del control base y agregar tantas modificaciones como creamos necesarias. En este caso, incorporamos una imagen dentro del botón.*

Los **controles compuestos** como estos nos llevan un poco más cerca de la personalización e implementación de código sobre la base de nuestras necesidades, ya que podremos agregar o modificar funcionalidad a los controles ya existentes. De cualquier manera, siempre seremos dependientes de un control creado con anterioridad para poder heredar su funcionalidad y sobrescribirla. El siguiente paso consiste en crear un control propio desde cero, y esto lo conseguiremos con los **controles personalizados avanzados**.

Controles personalizados avanzados

Los **controles personalizados** entran en la categoría más compleja de controles que podemos crear. En estos, no contamos con un lienzo para el diseño visual, tampoco tenemos un molde de otro control del cual podamos heredar, sino que debemos crear todo por nuestra cuenta. Aunque comparte ciertas similitudes con los **controles compuestos**, en especial en las sobrecargas de funciones, estaremos tocando los elementos más internos del desarrollo de este tipo de objetos.

```
public class ControlPersonalizado
    : Control
{
    ...
    ...
}
```

Lo primero que podemos notar del código anterior es que nuestro control personalizado hereda del tipo **Control**. Todos los controles heredan de este tipo en algún punto. En el caso de los controles compuestos, el botón que usamos como molde en una de sus capas también heredaba del tipo **Control**. Incluso, el control de usuario creado al inicio de esta sección llegaba a heredar del tipo **Control**. En este tipo de controles, también podemos hacer uso de los controles existentes, pero no heredando de ellos, sino utilizándolos como objetos que formarán parte de nuestro control.

```
private Timer temporizador;
private int ultimoCaracter;
private Label texto;
private bool mayusculas = true;

public ControlPersonalizado()
{
    ultimoCaracter = 1;
    temporizador = new Timer();
    temporizador.Enabled = false;
    temporizador.Interval = 200;
    temporizador.Tick += new EventHandler(temporizador_Tick);

    texto = new Label()
    {
        Text = this.Text,
        Left = 1,
        Top = 1,
        Width = this.Width
    };

    this.Controls.Add(texto);
}
```

EJECUCIÓN DE CONTROLES

Cuando diseñamos controles personalizados y agregamos funcionalidad como animaciones o código, que se ejecute de forma automática, este se ejecutará incluso en tiempo de diseño. Esto quiere decir que podremos ver cómo las animaciones o los cálculos resultantes de la automatización del código se ejecutan, aunque la aplicación no esté en funcionamiento.

El código anterior presenta varios conceptos importantes. Por un lado, un conjunto de variables globales referidas a nuestro control, por el otro, un constructor de clase que nos sirve para configurar algunas de las variables, así como el poner en funcionamiento un objeto del tipo **Timer** (temporizador) el cual usaremos para agregar funcionalidad a nuestro control, y, por último, un control de tipo **Label** (etiqueta) que será el elemento principal de nuestro control personalizado. Como en este tipo de controles no contamos con un lienzo para dibujar otros controles, es necesario que todos estos los agreguemos por código a la lista de controles que nuestro control personalizado puede mantener.

```
this.Controls.Add(texto);
```

La línea anterior es la que se encarga de adicionar la etiqueta y motiva que esta sea visible. El objeto de este control es simular uno de tipo **etiqueta**, pero con una funcionalidad diferente. Al asignarle un texto, cada **200** milisegundos, el temporizador irá convirtiendo cada una de las letras a mayúsculas y luego a minúsculas, creando una animación del texto que hayamos colocado en el control. Esta funcionalidad se consigue con las siguientes líneas de código.

```
private void temporizador_Tick(object sender, EventArgs e)
{
    if (mayusculas)
    {
        texto.Text = texto.Text.Substring(0, ultimoCaracter) +
            texto.Text.Substring(ultimoCaracter, 1).ToUpper() +
            texto.Text.Substring(ultimoCaracter == texto.Text.Length ?
     ultimoCaracter : ultimoCaracter + 1);
    }

    else
    {
        texto.Text = texto.Text.Substring(0, ultimoCaracter) +
            texto.Text.Substring(ultimoCaracter, 1).ToLower() +
            texto.Text.Substring(ultimoCaracter == texto.Text.Length ?
     ultimoCaracter : ultimoCaracter + 1);
    }

    Application.DoEvents();
```

```
     ultimoCaracter++;
    if (ultimoCaracter >= texto.Text.Length)

     {
          mayusculas = !mayusculas;
          ultimoCaracter = 1;
     }
}
```

Podemos ver que, en este simple fragmento de código, hemos aplicado todo lo que hemos visto hasta el momento en este libro. Liberamos la interfaz gráfica para que pueda mostrarnos los cambios mediante **Application.DoEvents**, tomamos parte de una cadena de texto y la convertimos de mayúsculas a minúsculas, y viceversa, entre otras cosas. Esto es característico, como ya dijimos, de este tipo de controles ya que, a mayor funcionalidad requerida, deberemos realizar mayor cantidad de líneas de código aplicando todo lo aprendido hasta ahora. En la **Figura 35**, podemos ver cómo este control modifica el texto en tiempo de ejecución.

***Figura 35.** Al ejecutarse la aplicación, vemos cómo, cada 200 milisegundos, cada una de las letras de la frase es convertida a mayúsculas y luego a minúsculas.*

SOPORTE DE CULTURAS

Existe una gran cantidad de culturas así como combinaciones de cultura-país incluidas en el Microsoft .Net Framework. Esto nos soluciona el problema de conocer el comportamiento de los números y fechas de culturas poco comunes para nosotros y así ahorrarnos tiempo en la producción de reglas de conversión de los datos.

Globalización de aplicaciones

Cuando hablamos de globalización, nos referimos al concepto que indica que una aplicación podría ser ejecutada en ambientes distintos de aquellos en los que fueron desarrolladas. Esto quiere decir que una aplicación escrita para un país hispanohablante podría requerir comportarse de forma diferente en un país de habla inglesa. Esto resulta más claro en el caso del manejo de fechas: aunque es común el uso del formato **dd/mm/yyyy** (día, mes, año), para nuestro caso, en países como Estados Unidos el formato de fechas se escribe de forma **mm/dd/yyyy** (mes, día, año). Por lo tanto, es necesario poseer algún mecanismo para que, si la aplicación así lo requiere, exista esta adaptación en textos, manejo de monedas, números y fechas, que corresponda a la cultura en la cual está siendo ejecutada nuestra aplicación. En la **Figura 36**, podemos ver lo que sucede cuando mostramos una fecha con un formato de cultura difrenete a la nuestra, como sería Estados Unidos.

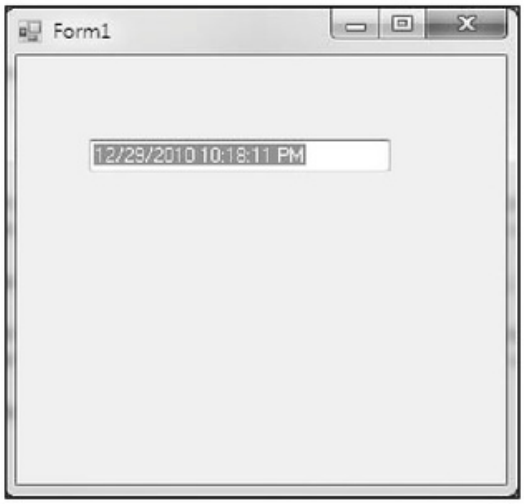


Figura 36. *El formato de fecha relacionada con una cultura distinta de la hispanohablante causa que la representación del orden de días y meses difiera de la conocida.*

A continuación, podemos ver el código que consigue la escritura de la fecha que observamos en la **Figura 36**.

```
this.txtFecha.Text = DateTime.Now.ToString();
```

Como vemos, el código simplemente toma el valor de la fecha y retorna una cadena de texto con esta. Debido a que el sistema está configurado para una cultura diferente de la hispana, la aplicación considera que debe mostrar la fecha en el formato del sistema y, por lo tanto, diferente del que conocemos. Si bien la aplicación toma las configuraciones regionales de la computadora donde se ejecuta, podemos forzar a que esta trabaje con otra configuración.

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
this.txtFecha.Text = DateTime.Now.ToString();
Thread.CurrentThread.CurrentUICulture = new CultureInfo("es-AR");
this.txtFecha2.Text = DateTime.Now.ToString();
```

Es posible modificar la cultura asociada a la interfaz visual del hilo de ejecución actual mediante el uso de **CurrentUICulture** (cultura actual para interfaz de usuario) pasándole el tipo de cultura que queremos utilizar. Esto se logra mediante la declaración de un objeto **CultureInfo** (información de cultura) y el par cultura-país. En el código anterior, vemos dos pares, uno que representa la cultura inglesa para Estados Unidos (en-US) y otro para el español de la Argentina (es-AR). Antes de tomar la fecha del sistema, cambiamos la cultura. Esto hace que su formato se vea modificado sobre la base de la cultura seleccionada. Junto al uso de **CurrentUICulture**, encontramos la función **CurrentCulture** (cultura actual) de igual uso que la anterior, pero con la diferencia que solo cambiará la cultura en relación con nuestro código y no de la interfaz visual. Es común utilizar las dos funciones en conjunto para que la interfaz visual tome la cultura que queremos utilizar, y nuestro código también lo haga.
Esto último es de vital importancia cuando estemos creando sitios web debido a que los valores ingresados por un usuario remoto puede definir completamente en las configuraciones de nuestros equipos.

... RESUMEN

En este capítulo, hemos aprendido lo primordial en el desarrollo de aplicaciones de escritorio para Windows. Hemos podido aprender el funcionamiento de los principales controles y, al mismo tiempo, entender cómo es posible hacer controles por nuestra cuenta. Además extendimos las posibilidades en el desarrollo de este tipo de aplicaciones al aprender el concepto de localización y globalización, y logramos crear aplicaciones que se adapten a los distintos idiomas en los que puedan ser ejecutadas, así como el manejo de fechas y números correspondientes a cada región. En el siguiente capítulo, nos enfocaremos en el desarrollo web, donde ampliaremos lo aprendido aquí gracias a que los distintos controles, componentes y funcionalidades poseen puntos de contacto en común. Esto nos servirá para completar el recorrido sobre el desarrollo bajo Microsoft .Net y así entender cada una de las facetas en las cuales se ve involucrado.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuántos botones de funcionalidad provee el sistema operativo para los formularios en aplicaciones de escritorio?

2 ¿Podemos crear controles sin interfaz visual?

3 ¿Cuál es la función de la línea de código Application.DoEvents?

4 ¿Cuál es el formato que debemos utilizar para definir la cultura chino simplificado?

5 ¿Para qué se utiliza el operador AS?

6 ¿Existen controles y componentes no provistos por Microsoft .Net?

7 ¿Qué es un archivo de recursos?

8 ¿Los archivos de recursos para manejo de idiomas requieren de algún formato en particular?

9 ¿Qué línea de código debemos ejecutar si queremos agregar un control a un formulario?

10 ¿Cómo podemos deshabilitar un control?

EJERCICIOS PRÁCTICOS

1 Habiendo creado el control botón personalizado, intente agregarle un color de fondo distinto al que trae por defecto.

2 Cree un nuevo archivo de recursos para soportar italiano y despliegue un texto modificando la cultura de la aplicación.

3 Asocie a una caja de texto una función que se ejecute cada vez que una tecla es presionada sobre ella.

4 Usando un nuevo control contenedor dentro de un formulario, agregue un nuevo grupo de botones circulares de selección que trabajen de forma independiente a los existentes en el ejemplo.

5 Ordene los controles en un formulario de tal forma que, al presionar la tecla TAB, el foco se realice en orden.

Capítulo 8

Programación de sitios web

Los sitios web se han convertido, en los últimos años, en el principal objetivo de las empresas para promocionar y exponer sus productos, para interconectar usuarios o para compartir información, y, poco a poco, el desarrollo se ha movido de las plataformas tradicionales a esta última. En este capítulo, aprenderemos sobre este mundo desde la perspectiva brindada por Microsoft .Net mediante el uso de la tecnología llamada ASP.net.

SERVICIO DE ATENCIÓN AL LECTOR: usershop@redusers.com

CREAR UN SITIO WEB

En los capítulos previos, hemos estado utilizando Visual C# 2010 Express como herramienta de desarrollo. Esta herramienta es ideal para el desarrollo de aplicaciones en ambientes Windows, tanto de escritorio, como aplicaciones de consola o servicios de Windows, pero no nos será de utilidad para el desarrollo de sitios web con **ASP.net**. Esto es debido a que la forma de trabajo, las plantillas de desarrollo y los controles y componentes presentados por ella no son compatibles con los utilizados en el desarrollo de aplicaciones web con ASP.net.
Si no contamos con una versión de Microsoft Visual Studio 2010 instalada en nuestra computadora, podremos hacer uso de **Visual Web Developer 2010 Express**. Esta herramienta es similar a la que hemos estado utilizando en el transcurso de este libro, pero con la diferencia de que se enfoca directamente en el desarrollo web, lo que nos facilita la creación de sitios para Internet.

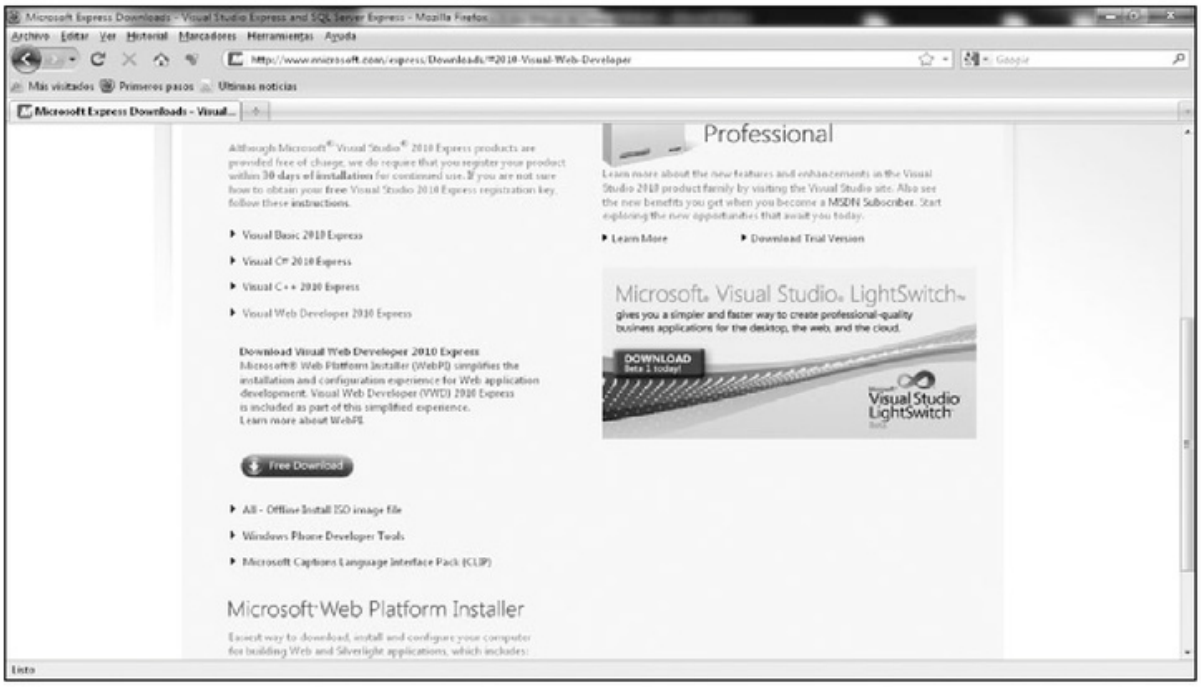


***Figura 1.** Desde el sitio web de Microsoft, podemos descargar todas las versiones Express de las herramientas de desarrollo. **Visual Web Developer 2010 Express** es ideal para la confección de páginas y sitios web de ASP.net.*

Antes de proseguir, si no contamos con Microsoft Visual Studio 2010, será necesario que descarguemos e instalemos Visual Web Developer 2010 Express. Podemos descargar la herramienta de desarrollo desde la página oficial: **http://www.microsoft.com/express/Downloads/#2010-Visual-Web-Developer**. Si bien, y al igual que en el desarrollo para escritorio con Microsoft .Net, podemos crear nuestros sitios web sin herramienta alguna, pero la cantidad de componentes, controles y funcionalidades necesarias para desarrollar una aplicación son tantas que consumiría

mucho más tiempo y esfuerzo que si usáramos, al menos, una que nos brindara los servicios de desarrollo mínimos, para la construcción del sitio.

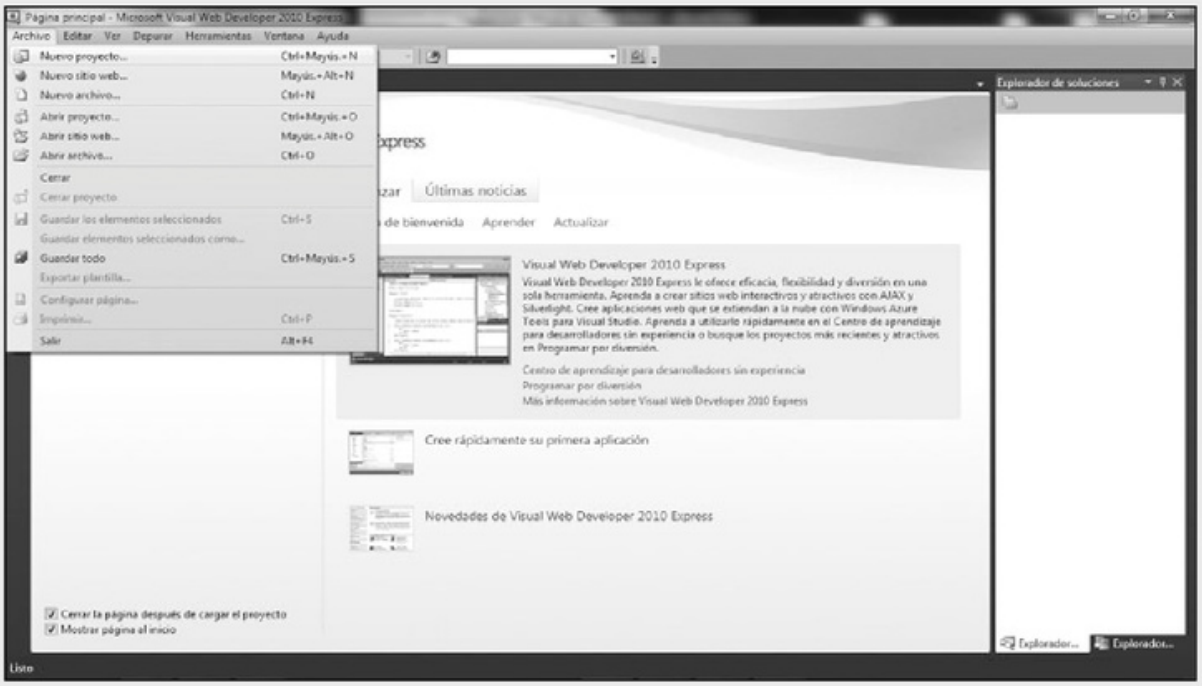
Nuevo proyecto web

Con Visual Web Developer 2010 Express instalado, podemos iniciar la construcción de nuestro primer sitio web. Al abrir la herramienta de desarrollo, notaremos que resulta similar a la que ya hemos estado utilizando. Esto nos dará mayor facilidad para entender su funcionamiento.

Al igual que una aplicación de consola o una de escritorio para Windows, podemos crear nuevas aplicaciones web realizando las instrucciones del siguiente **Paso a paso**.

Creación de una aplicación web — PASO A PASO

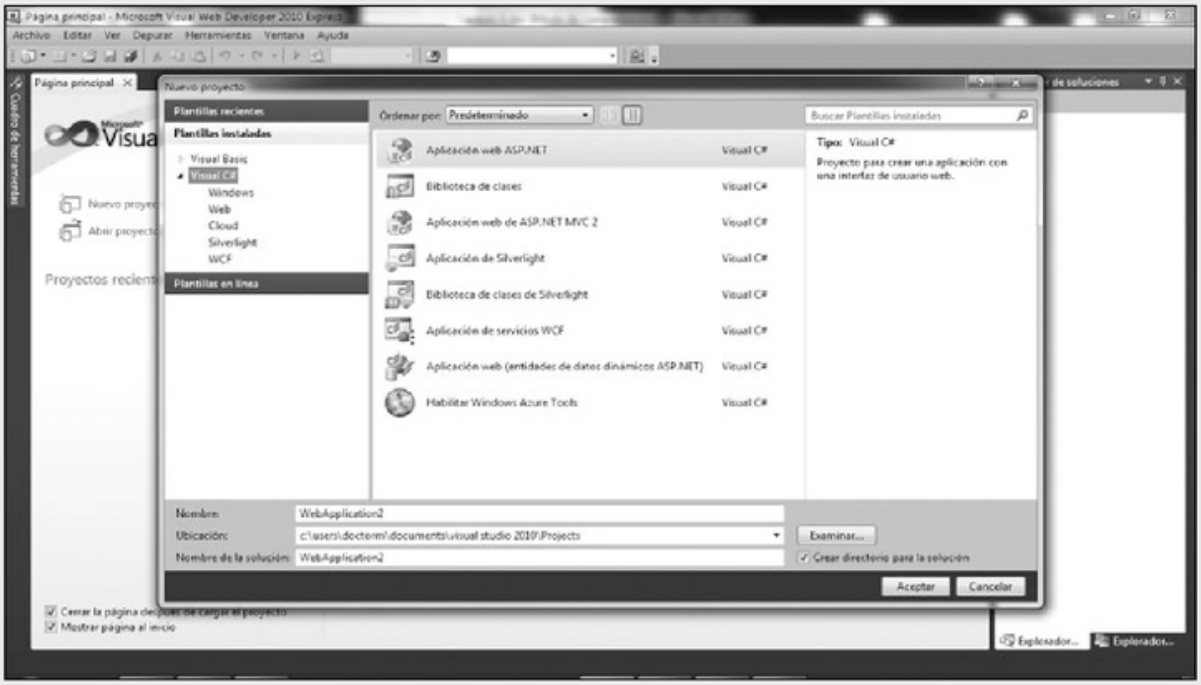
1 Desde el menú **Archivo**, seleccione **Nuevo proyecto**.



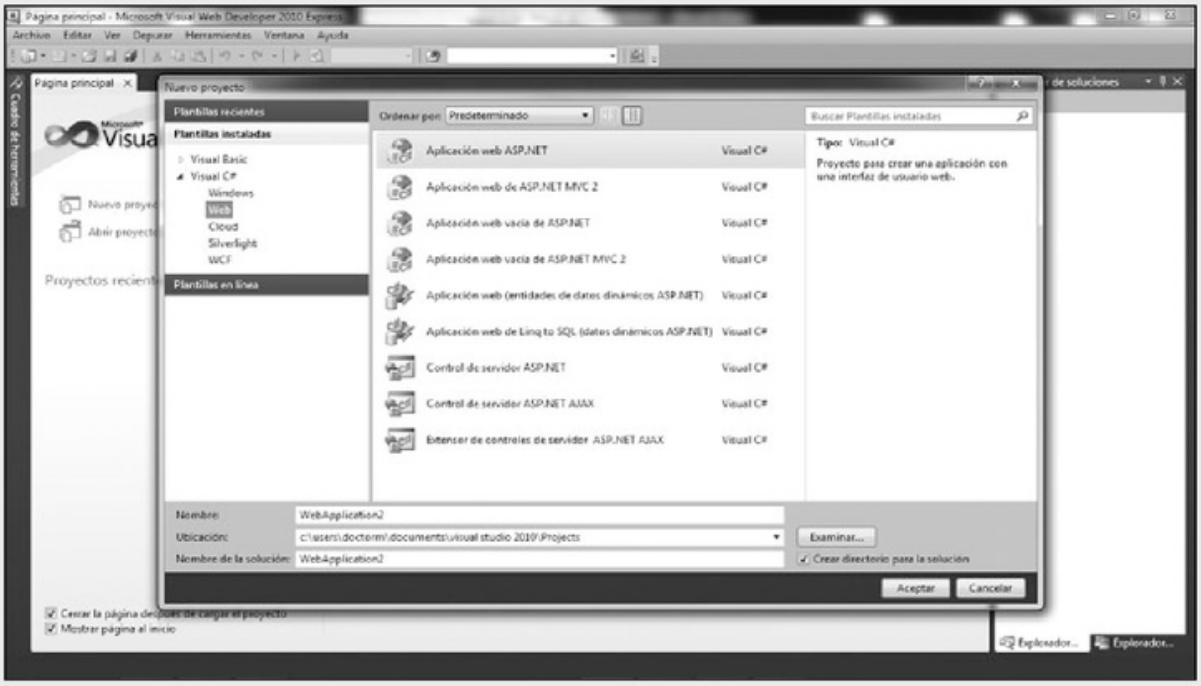
MICROSOFT VISUAL STUDIO 2010

Si bien la versión de Visual Studio 2010 es la herramienta más completa para el desarrollo, incluido el desarrollo web, podemos utilizar Visual Web Developer 2010 Express para el desarrollo de aplicaciones web con ASP.net. Esta herramienta nos proveerá de todos los elementos esenciales para construir aplicaciones web completas.

2 En la ventana de **Nuevo proyecto**, seleccione C# como lenguaje de programación.



3 En la lista de proyectos disponibles, seleccione web y, luego, **Aplicación Web ASP.NET**.



4 Presione el botón **Aceptar**. El nuevo sitio web será creado con archivos por defecto en el **Explorador de soluciones**.

Una vez que el sitio web es creado, en el **Explorador de soluciones** podremos ver que se ha adicionado un conjunto de archivos al proyecto. Estos archivos son equivalentes a los archivos que son adicionados cuando creamos aplicaciones de escritorio para Windows en el capítulo anterior. En este caso, vemos un archivo llamado **Default.aspx** utilizado como punto inicial de nuestro sitio web, así como un archivo llamado **Web.Config** el que contendrá diferentes configuraciones para nuestro sitio web. Otros archivos serán adicionados durante la construcción del sitio.

***Figura 2.** Un sitio web recién creado desde Visual Web Developer 2010 Express adiciona un conjunto de archivos iniciales de vital importancia para el funcionamiento de un sitio web.*

Para ejecutar el sitio web recientemente creado, deberemos presionar la tecla **F5** o, en su defecto, el icono de **Inicio** (flecha de color verde en la barra de herramientas). Esta acción hará que el sitio se ejecute para que podamos probarlo a medida que vayamos creándolo. Es fácil darse cuenta que el modelo de ejecución es similar al de una aplicación de consola o de escritorio para Windows.

SERVIDOR WEB PORTABLE

Visual Web Developer 2010 Express trae consigo un servidor web para realizar pruebas rápidas de nuestro avance en la construcción de un sitio web si es que no contamos con **Internet Information Service** como plataforma para hospedar nuestro desarrollo. Este servidor web de pruebas inicialmente recibía el nombre código de **Cassini**.

***Figura 3.** El sitio web inicial solo provee una página web en blanco. Al presionar la tecla F5 para ejecutarlo, Visual Web Developer 2010 Express abrirá el navegador web que tengamos configurado por defecto para mostrar los resultados.*

Al ejecutar la aplicación web, el navegador que tengamos configurado como el navegador por defecto será abierto para visualizar el sitio web. Junto con la ejecución, podremos notar que un icono de una página web es creado en la barra de tareas, al lado de la fecha y hora del sistema; esto se debe a que un sitio web no funciona igual que una aplicación de escritorio para Windows.

Un sitio web necesita de una infraestructura para trabajar, alguien que ejecute y administre las páginas web, las conexiones y la generación de código. Para poder ejecutar las aplicaciones que vayamos creando con Visual Web Developer 2010 Express y tengamos estos requerimientos de infraestructura, la herramienta de desarrollo provee un pequeño servidor que será utilizado para gestionar nuestras páginas.

Este servidor solo debe ser utilizado para realizar nuestras pruebas, a medida que vamos desarrollando la aplicación web. Como tal, solo pueda soportar una mínima cantidad de conexiones, viene configurado para que todo el modelo ASP. NET funcione en él.

III SERVIDOR WEB DE MICROSOFT

Una vez finalizado el desarrollo de una aplicación web, es necesario hospedarla en un servidor que la gestione. Para tal fin, se necesita contar con un servidor especializado. Microsoft propone Internet Information Services como la plataforma de administración de sitios web. Esta plataforma es útil para hospedar sitios ASP.net y también para PHP, Phyton y otros lenguajes web.

***Figura 4.** ASP.Net Development Server es un pequeño servidor provisto por la herramienta de desarrollo para poder realizar pruebas de nuestro código.*

Si presionamos dos veces con el mouse sobre el icono del servidor de desarrollo, aparecerá una ventana descriptiva con datos sobre este. Estos datos especifican características de ejecución de este servidor, como el **puerto** por el cual recibirá las peticiones del navegador web, la ruta o **directorio físico** de la aplicación que está administrando, la versión y la ruta para poder acceder al sitio web.

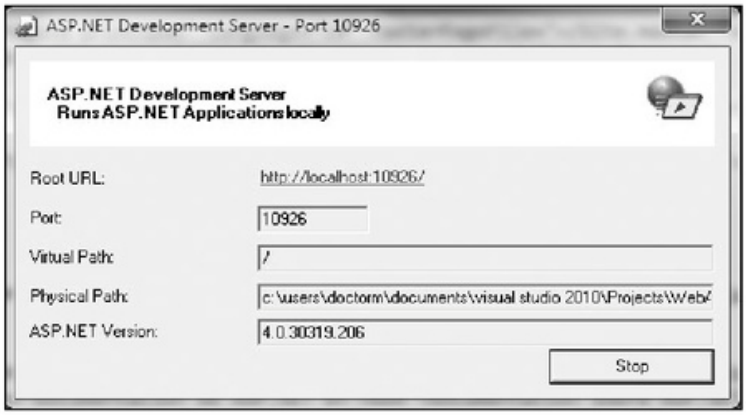


***Figura 5.** Si presionamos dos veces sobre el icono de ASP.Net Development Server, este nos mostrará las propiedades para poder administrar nuestro sitio web para pruebas.*

En la **Figura 6** y en la **Figura 3**, podemos ver que nuestro sitio web así como el servidor provisto por la herramienta de desarrollo muestran que la dirección para acceder al sitio web comienza con la palabra **localhost** (hospedaje local). Este nombre es utilizado para representar la dirección virtual por defecto cuando los servidores de administración de páginas web hospedan sitios web en forma local; es una manera de apuntar todas las

llamadas del navegador web a nuestra misma computadora y que nuestra PC sepa cómo resolver el acceso a esta mediante el nombre. Otra forma de acceder sería por medio de la **dirección IP** que hace referencia a nuestra computadora, esto es, escribiendo **127.0.0.1** como dirección en la barra de navegación del navegador.

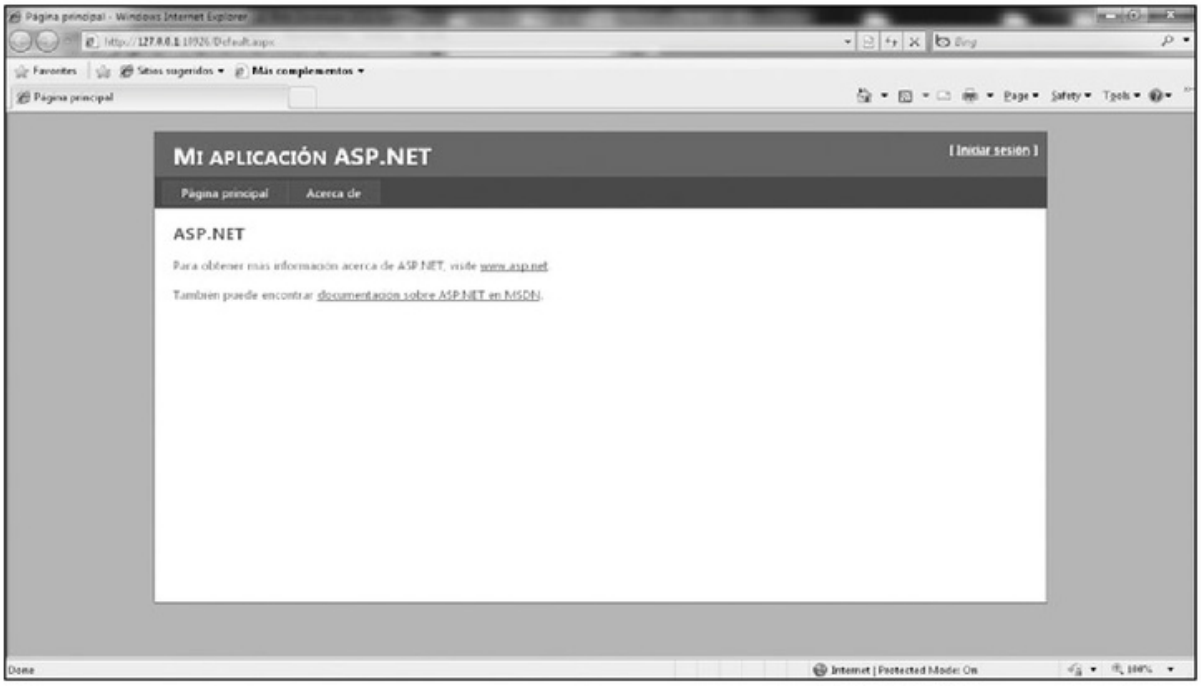


***Figura 6.** Mediante el uso de la dirección IP que referencia a nuestra propia computadora, es posible acceder al sitio web administrado por el servidor provisto por Visual Web Developer 2010 Express.*

El último elemento que aparece en la **Figura 4** es el **número de puerto** que sigue al nombre **localhost** o **dirección IP** de nuestra computadora. Este número representa el puerto o canal por el cual el servidor web está escuchando o esperando peticiones desde los distintos navegadores web con los cuales queramos conectarnos al sitio web y ver su contenido. Esto quiere decir que nuestra computadora, a través del servidor web que se está ejecutando, abre un canal por el cual esperará peticiones y responde a ellas con los resultados de las distintas páginas web. Por defecto, los servidores web utilizan el **puerto 80** para comunicarse con los navegadores web.

PUERTO 80

Los navegadores web utilizan distintos puertos de comunicación para poder acceder a los sitios web. Por defecto el **puerto 80** es aquel que todos los navegadores utilizan para ingresar en las páginas web en Internet. Si a cualquier dirección web en Internet le agregamos el sufijo **:80**, seguiremos accediendo al mismo sitio web.

El hecho de que, por lo general, no coloquemos el puerto se debe a que se considera estándar para los navegadores y servidores web el uso de este puerto, y, por lo tanto, el navegador web en forma automática enviará las peticiones utilizando este número para el puerto de comunicaciones.

COMPONENTES DE ASP.NET

Para poder realizar un sitio web, necesitaremos conocer diferentes tecnologías, desde lenguajes de programación como **JavaScript**, o el conjunto de etiquetas de **HTML** (*Hyper Text Markup Language*, o en castellano, lenguaje de etiquetas de hipertexto) que representan visualmente la página, hasta la manipulación de la apariencia visual por medio de **CSS** (*Cascade Style Sheet,* o en castellano, hojas de estilo en cascada). Estos lenguajes, definiciones y tecnologías son solo algunos de los que necesitaremos conocer para poder crear sitios web profesionales además de la tecnología ASP.net en sí misma, aunque esta nos provea de herramientas de fácil uso.
De cualquier manera, el soporte brindado por ASP.net y las tecnologías que la rodean permiten suplantar el aprendizaje de todos estos conceptos por otros más cercanos al desarrollo de software tradicional. En todo caso, no tendremos que preocuparnos por el código resultante ya que todas las tecnologías provistas por Microsoft .Net para el desarrollo web traducirán nuestro código a aquel que pueda ser entendido e interpretado por los distintos navegadores web.

Páginas ASP.net

Cuando aprendimos sobre el desarrollo de aplicaciones de escritorio, vimos que un formulario era utilizado como el lienzo donde los distintos controles y componentes eran dibujados y, con los cuales, el usuario podía interactuar. En el caso del desarrollo web con ASP.net, las páginas web cumplen una función similar.
Las páginas web en ASP.net son los archivos con la extensión ASPX seguida del nombre del archivo y se separan en dos grandes áreas. Por un lado, la página **ASPX** en sí, la cual contiene fundamentalmente componentes y controles de ASP.net junto a código HTML que, en conjunto, generan código que los navegadores web pueden entender. Por otro lado, una clase con código C# que administra y maneja los distintos componentes de la página así como las peticiones que se realicen desde el usuario.
Esta separación entre una página con código HTML y otro archivo con código en C# se debe a la forma en cómo el mundo de Internet, navegadores web y servidores web, funciona. Esto sucede a diferencia de las aplicaciones tradicionales que hemos estado construyendo, y que siempre son ejecutadas y administradas por la PC en donde estas están trabajando.

En el caso del desarrollo web, los navegadores web solo interpretan código HTML que es enviado desde el servidor web, pero con la diferencia de que la interacción entre el navegador web y el servidor que contiene la lógica de nuestro programa se limita al único momento en que el primero realiza una petición al segundo. Estas peticiones se limitan al envío de información simple que destaca datos del navegador, la página a la cual se quiere acceder y los parámetros de navegación hacia dicha página, si fueran necesarios, como se muestra en el código siguiente.

```
GET /default.aspx HTTP/1.1
Accept: */*
Accept-Language: ...
User-Agent: Mozilla/4.0...
Host: www.miSitioWeb.com
```

La petición anterior hará que el servidor busque la página **default.aspx** en el sitio web **miSitioWeb.com**, el cual debería estar administrado por el servidor web al cual se le está realizando esta petición. En respuesta, si la página existiera, el servidor tomará los valores que hayamos escrito en el archivo con extensión ASPX, ejecutará el código de C# que maneja los controles y componentes de esta, y su resultado, el cual será código HTML, se enviará al navegador web, que finalmente leerá este HTML y lo mostrará al usuario. En la **Figura 7**, podemos ver este concepto.

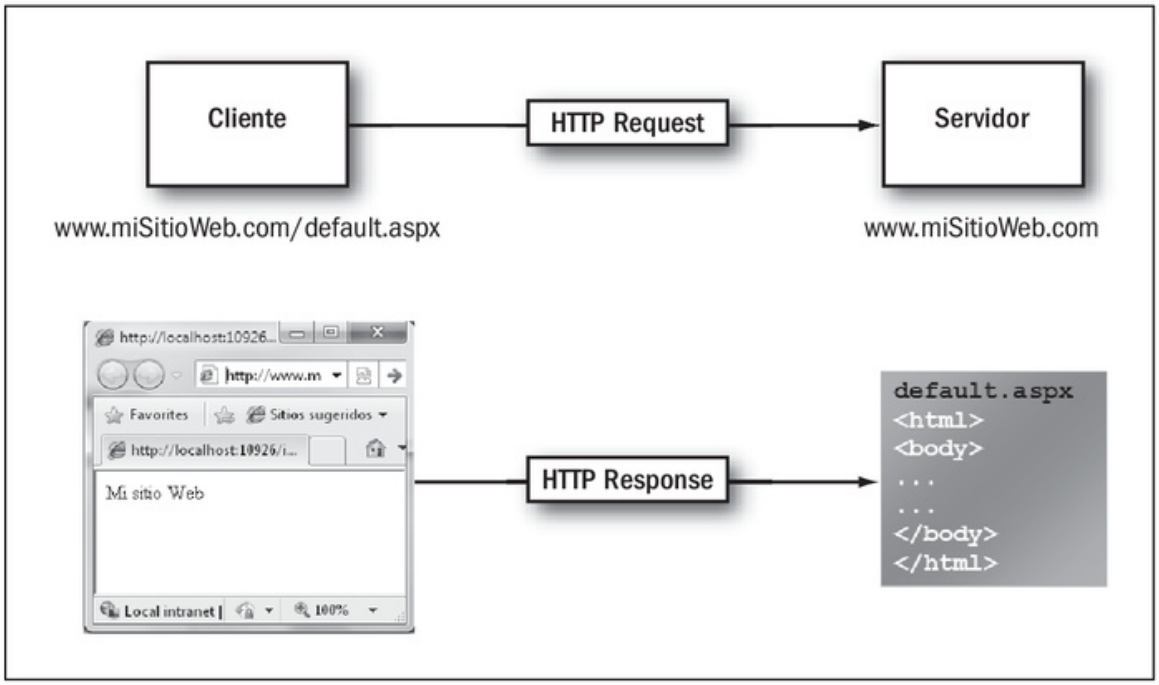
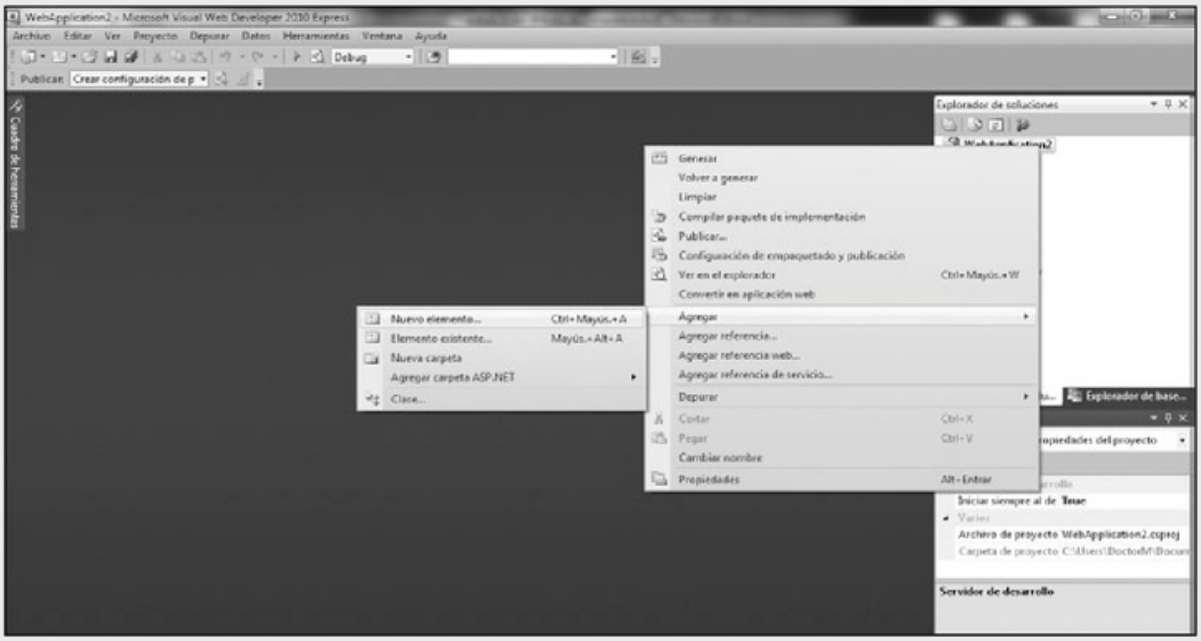


Figura 7. *El navegador solo representa el código HTML enviado desde el servidor una vez que se ha procesado la petición web. En este punto, el servidor ya no mantiene ninguna relación con el navegador del cliente.*

Por lo tanto, el servidor y el cliente (servidor web) no mantienen ningún tipo de contacto entre ellos. Esto quiere decir que, una vez que la página HTML es mostrada por el navegador, el servidor pierde contacto con el cliente web y solo lo retomará cuando este último vuelva a realizar una petición de una nueva página web al servidor web. Esto puede resultar un cambio completo de paradigma si lo comparamos con el de las aplicaciones de escritorio ya que, a diferencia de ellas, una página web posee un tiempo de vida muy corto. Esto quiere decir que solo está activa en el servidor en el momento en que se realiza una petición por parte del cliente, inicializando todas las variables que necesitemos, creando los objetos que hayamos designado, ejecutando sus funciones para, luego, obtener el HTML resultante y finalmente **destruir todos estos elementos** para liberar la memoria del servidor. Agreguemos una nueva página web a nuestro proyecto.

Nueva página ASP.net — PASO A PASO

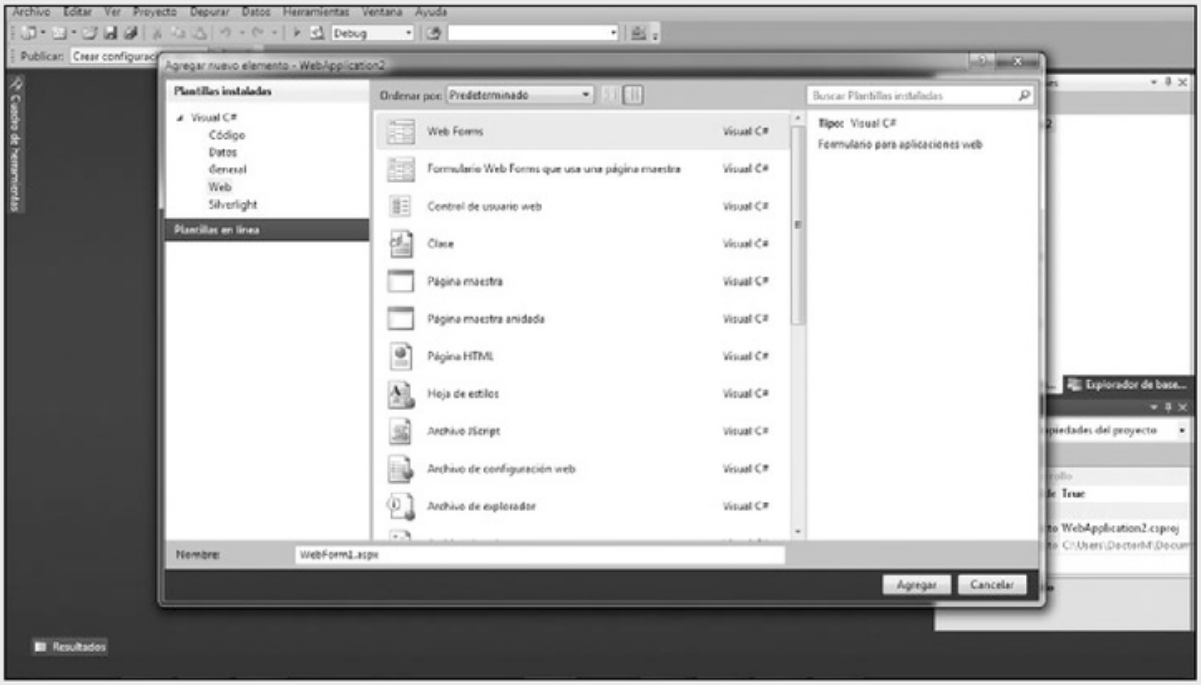
1 Presione sobre el proyecto **Agregar** y pulse sobre **Nuevo elemento**.



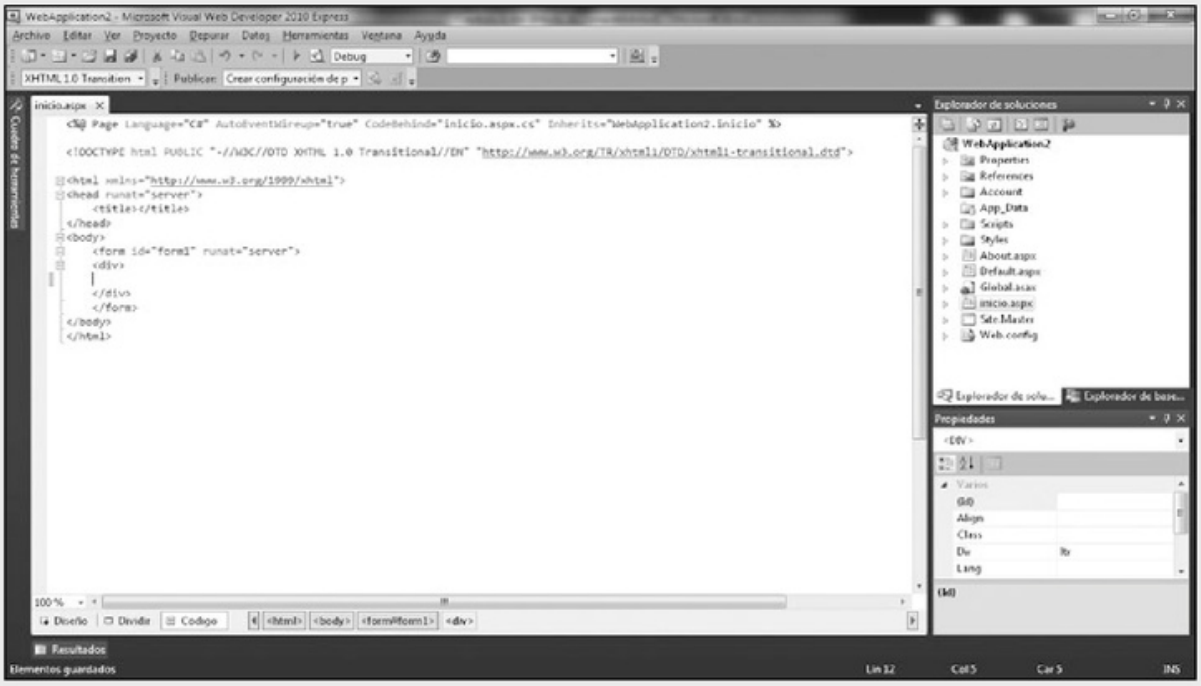
III PÁGINAS ASP.NET

Las páginas ASP.net son parte fundamental de nuestros sitios Web, de cualquier manera es posible capturar las peticiones realizadas por los navegadores Web sin necesidad de tener una página física. Mediante la implementación de un HTTP Handler podremos capturar las peticiones y retornar diferentes valores para ser interpretados por el navegador Web.

2 En la ventana **Agregar nuevo elemento** seleccione **Web Forms**.



3 Escriba un nuevo nombre para la página web y presione **Agregar**.



Esta muestra una implementación simple de código HTML; en él, crearemos nuestros controles y la parte visual de la página. Desde el **Cuadro de herramientas** y pulsando dos clics sobre él, adicionaremos un control del tipo **botón**.

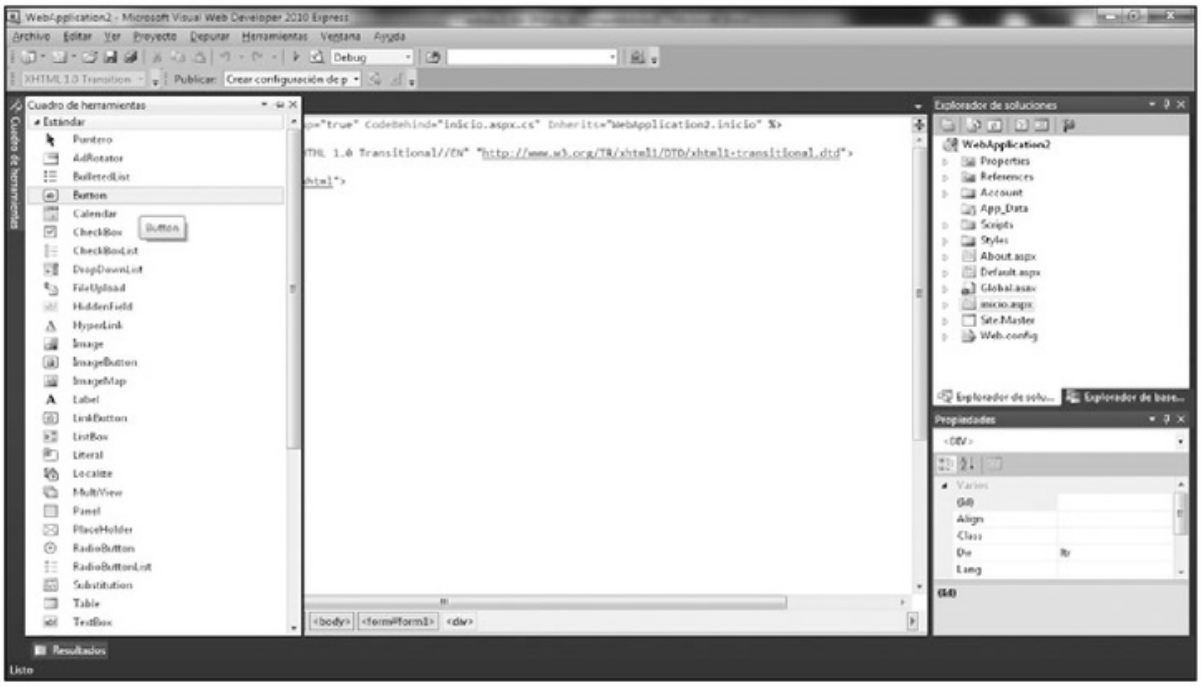


Figura 8. *El Cuadro de herramientas, al igual que en las aplicaciones de escritorio, muestra la lista de controles y componentes disponibles para el desarrollo de aplicaciones con ASP.net.*

```
<form id="form1" runat="server">
<div>
    <asp:Button ID="Button1" runat="server" Text="Button" />
</div>
</form>
```

El código anterior nos muestra el resultado final de adicionar el nuevo botón a la página de ASP.net. Este botón, si bien se muestra en formato HTML, no posee características estándares de HTML; esto quiere decir que el marcador usado para describir el control no es soportado por los distintos navegadores al no ser un marcador estándar. De cualquier manera, este se mostrará de forma correcta en todos los navegadores una vez que ejecutemos el sitio web. Nos ocuparemos de esta idea

III EL NOMBRE TAG

Cuando nos referimos a elementos HTML o marcador, en realidad, estamos haciendo referencia a **TAGs** HTML. Este es el nombre que reciben normalmente, y representan los nombres descriptivos que son utilizados por los navegadores para dibujar el código HTML. Un **TAG** THML comienza con el nombre de este dentro de los símbolos **<** y **>**, finalizando con el nombre dentro de **</** y **>**.

más adelante en este mismo capítulo cuando aprendamos sobre controles ASP.net. Una de las utilidades de Visual Web Developer 2010 Express es la posibilidad de poder ver el código HTML y, al mismo tiempo, una previsualización de los cambios que estemos realizando. Para esto, deberemos presionar en la pestaña **Dividir**.

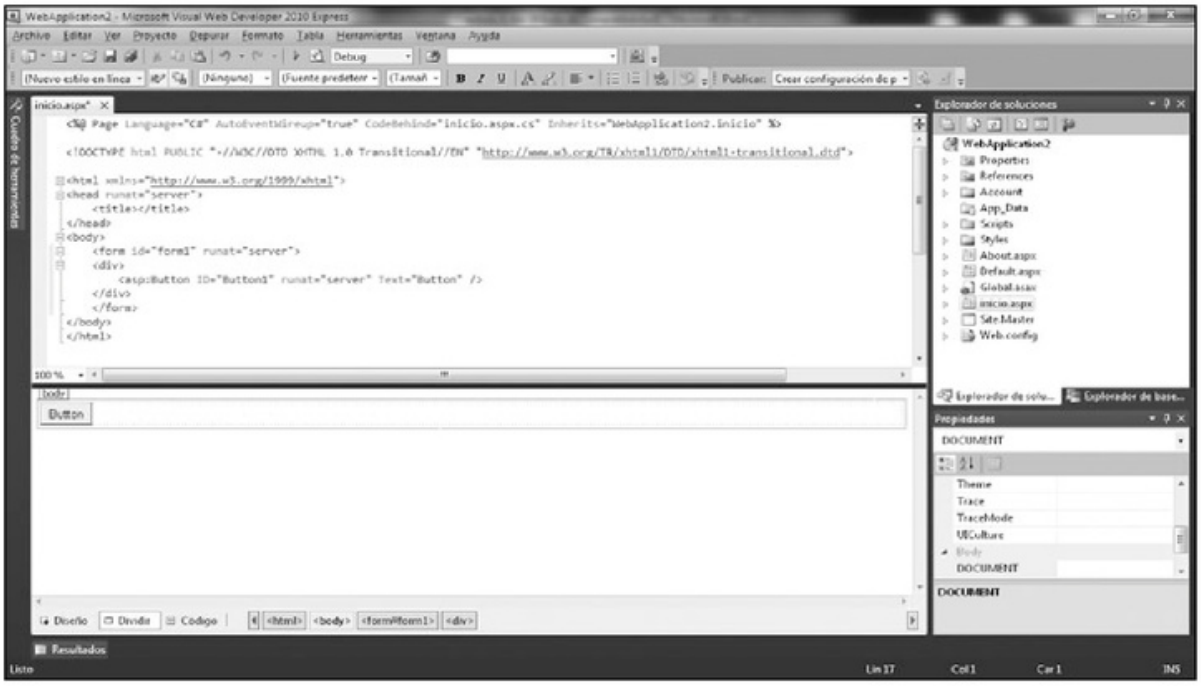


Figura 9. *Podemos trabajar en tres vistas diferentes. Visualizando solo el código HTML, desde la inspección visual o desde una pantalla dividida que muestra tanto HTML como el resultado visual. Cualquier cambio realizado en una de las ventanas es reflejado en la otra.*

Una vez que tenemos la pantalla divida en dos, es posible seleccionar los distintos controles e interactuar con sus propiedades y eventos. Si presionamos dos veces sobre el botón que hemos agregado, se adicionará una nueva función para el manejo del evento **Click** de este, de igual forma como sucedía en las aplicaciones de escritorio. Al hacer esto, veremos que la clase que administra nuestra página es abierta, y el código que capturará el evento es escrito en C#. Depende cómo configuremos el manejo de los controles del servidor también es posible que el código de dichos controles sea adicionado en la misma página HTML, aunque seguirá manejado por el servidor.

III EL OBJETO RESPONSE

El objeto **Response** es utilizado para manipular la salida de código C# hacia el navegador web. Cuando utilizamos **Response.Write**, estamos escribiendo HTML directamente en el resultado de la página que se mostrará en el navegador. Esto es equivalente a utilizar **Console.WriteLine** en aplicaciones de consola.

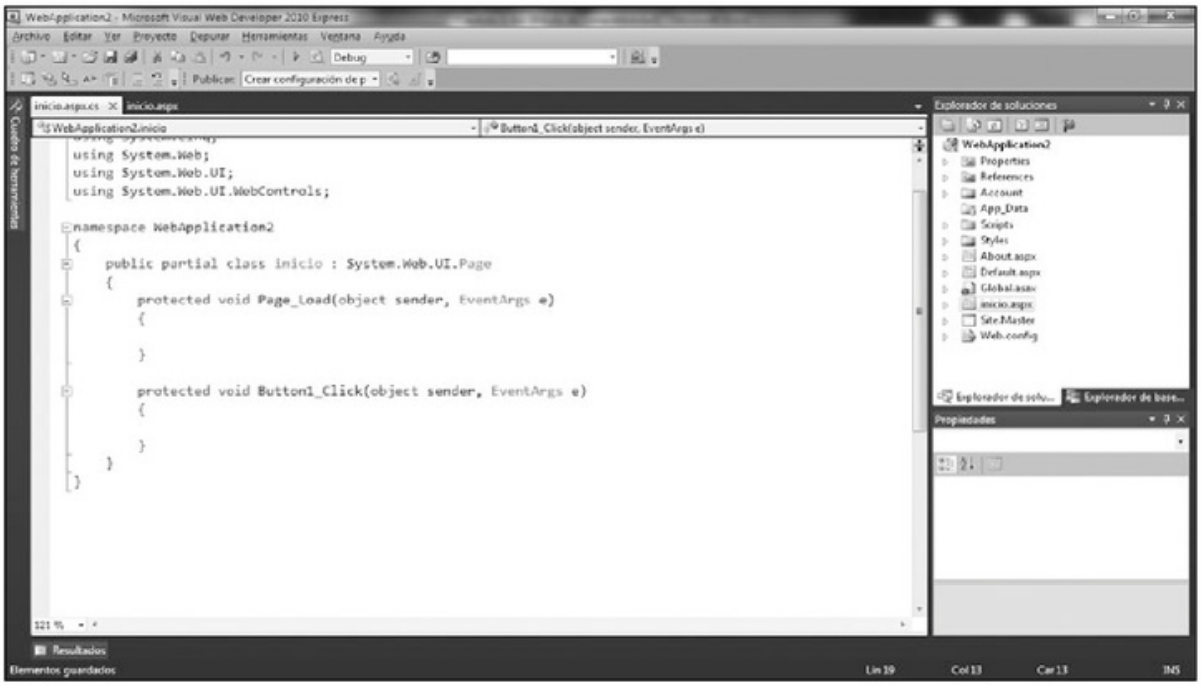


Figura 10. *Los controles de servidor ASP.net presentan eventos con los cuales el usuario puede interactuar desde la página Web. Cada evento en el cliente posee una correspondencia en el servidor, el cual será manejado por código C#.*

El que tengamos código en C# para el manejo del evento se debe a que este será manejado desde el servidor. Para conseguir esto, cada vez que el usuario presione sobre el botón, el contenido de la página será enviado al servidor en una nueva petición como veíamos en la **Figura 7**. Esto causará que el código de C# que maneja la página se inicialice, ejecute el evento, genere código HTML y, luego, se destruyan todos los objetos para liberar, así, la memoria.

```
private int sumador = 0;

protected void Button1_Click(object sender, EventArgs e)
{
    sumador++;
```

III STATELESS

El desarrollo de aplicaciones web, la forma en cómo el protocolo de Internet funciona, hace que los servidores web no mantengan una conexión constante con los diferentes clientes que consumen las páginas web. Este concepto recibe el nombre de **Stateless** (sin estado), lo que permite poder administrar miles de peticiones de manera rápida y con la menor cantidad de consumo de recursos.

```
        Response.Write("Valor de sumador: " + sumador.ToString());
    }
```

En el código anterior, podemos ver que hemos declarado una variable **sumador** de tipo **int**, que es inicializada con el valor de **0**. Luego, en el evento que se disparará cuando el usuario presione el botón de la página, sumaremos **1** al valor almacenado en dicha variable y lo escribiremos en la página como HTML.
Si ejecutamos la página por primera vez y presionamos sobre el botón, veremos que el valor de la variable es escrito en forma correcta.

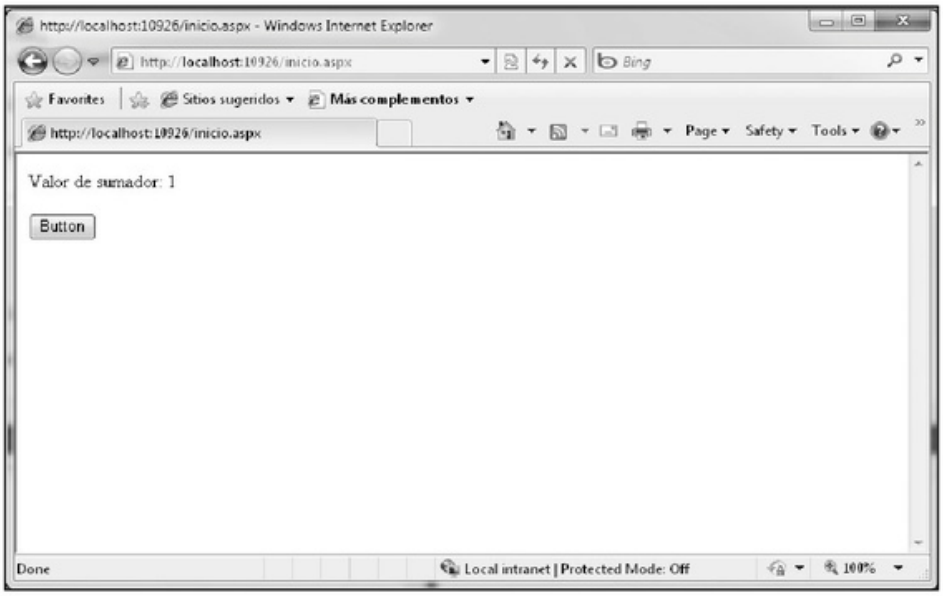


Figura 11. *Todo el código de manejo de eventos de los controles ASP.net es tratado en el servidor. El control será mostrado en formato HTML en el navegador web, pero, en el momento en que el usuario lo presione, esta acción será manejada desde el código C#.*

Esto se debe a que, inicialmente, la variable **sumador** contiene el valor **0** y, en el momento en que se presiona sobre el botón, se ejecuta el código que suma **1** a esta variable; así se obtiene como resultado el valor que vemos en la página. Si presionamos otra vez sobre el botón, podremos comprobar que el resultado será nuevamente **1** en vez de **2**. Este comportamiento es debido a la forma en cómo trabajan las páginas web que hemos nombrado con anterioridad. Esto puede ocasionarnos un problema si no estamos acostumbrados a desarrollar código que no dependa de los datos almacenados en la memoria, o intentamos crear aplicaciones con múltiples pasos, los que deben ser recolectados y retenidos por la aplicación hasta poder utilizarlos al final.
A pesar de esto, existen mecanismos para poder almacenar, de manera temporal, esta información, así como técnicas de desarrollo de código para no tener la necesidad de almacenar tanta información en variables volátiles.

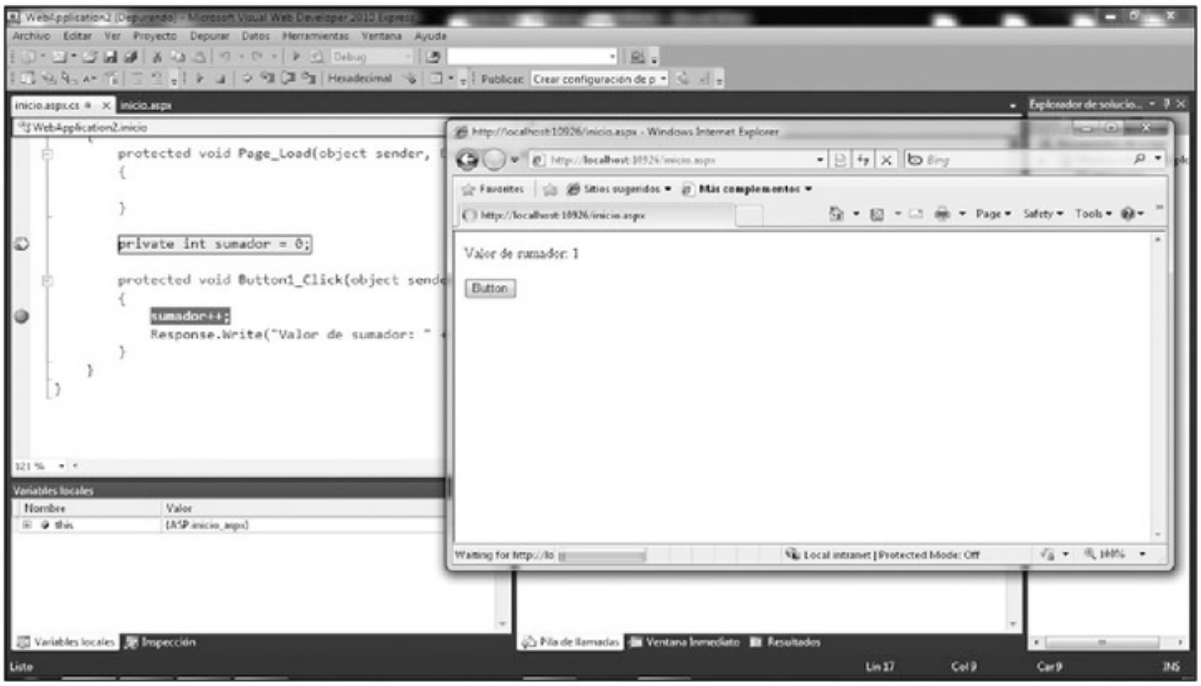


Figura 12. *Al presionar otra vez sobre el botón en la página web, todo el código es ejecutado nuevamente. Las variables deben ser inicializadas tomando el valor por defecto inicial y no manteniendo el estado que tenían en la anterior ejecución.*

Etiqueta Formulario

ASP.net no modifica la forma en cómo funcionan las **páginas web**, **Internet** y sus **protocolos**; esto quiere decir que, a pesar de tener etiquetas propias para describir controles como los botones, el resultado final será uno compatible con los parámetros utilizados por todos los navegadores web que interpretan código HTML.
De esta forma, ASP.net necesita de los mismos elementos utilizados en la creación de HTML para enviar información al servidor. Esto lo logra mediante el uso de la etiqueta HTML **Form** (formulario), la cual engloba a todos los controles ASP.net que se encuentren dentro de una página ASP.net.

```
<form id="form1" runat="server">
...
...
...
</form>
```

En el código HTML anterior, podemos ver cómo esta etiqueta **form** contiene a todos los elementos de la página; además podemos notar un atributo de esta etiqueta que no es estándar del contexto HTML. El atributo **runat** (trabajar en) especifica quién administrará el elemento y dónde lo hará. En este caso, se especifica que será administrado por el servidor web mediante el valor **server** (servidor).

Además de ser administrado por el servidor, lo que quiere decir que todos los procesos de traer la información que venga contenida en el formulario tras cada petición realizada por parte del usuario, esta función hará que la etiqueta administrada retorne el código HTML de forma que sea entendido por los distintos navegadores web, por lo que no debemos preocuparnos por problemas de compatibilidad.

```
<form method="post" action="inicio.aspx" id="form1">
...
...
...
</form>
```

Una vez ejecutada y procesada la página web, el formulario es transformado en el código HTML anterior. Como vemos, la declaración inicial del formulario es modificada para que el código resultante sea compatible con los distintos navegadores.
Otro comportamiento importante es la adición del atributo **action** (acción), el cual especifica el nombre de la página que procesará este formulario una vez que el usuario envíe la información al servidor. Como este formulario está contenido en la página **inicio.aspx**, el atributo es configurado para que la petición se realice sobre sí misma. Este proceso recibe el nombre de **postback**.
La etiqueta **form** en ASP.net resulta de vital importancia no solo para el envío de información hacia el servidor, sino también para mostrar los controles de una página.

```
<form id="form1" runat="server">
...
...
...
</form>
<asp:Button ID="Button1" runat="server" Text="Button"
     onclick="Button1_Click" />
```

III POSTBACK

Tras cada petición realizada por el cliente hacia un servidor web, existe un envío de información por parte del primero hacia el segundo, que provoca una respuesta por parte del segundo hacia el primero. Esta respuesta suele ser código HTML con información actualizada sobre la base de la petición realizada. Este proceso recibe el nombre de **Postback** cuando trabajamos con ASP.net.

En el código anterior, el control botón se encuentra fuera de la etiqueta **form** manejada por el servidor. Al hacer esto, los controles no pueden ser manejados en forma correcta, por lo que la página nos mostrará un error que deberemos corregir.

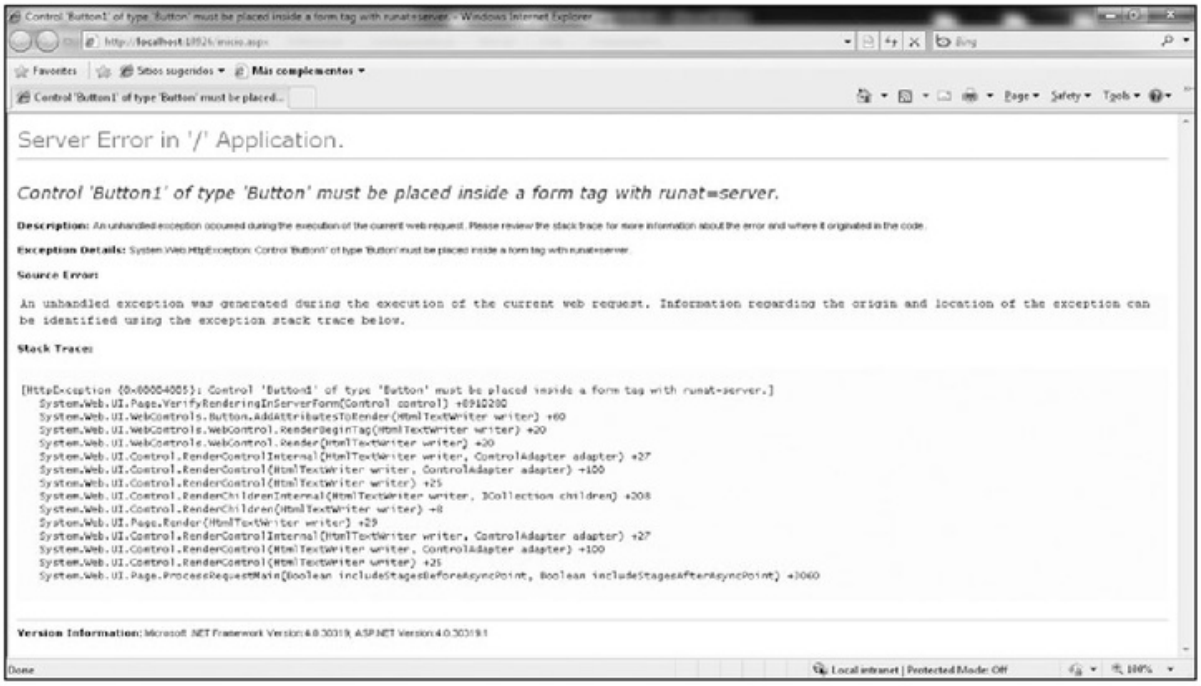


Figura 13. *Al no poder manejar los controles que están por fuera de la etiqueta de formulario, la página ASP.net nos arrojará un error que deberemos corregir para que pueda funcionar correctamente.*

El archivo Web.config

Así como en un sitio web de ASP.net existen las páginas de ASP.net, también podremos encontrar otros archivos adicionales que sirven para la ejecución del sitio que estemos construyendo. El archivo **Web.config** es utilizado para configurar un sitio web de ASP.net; en este, podremos escribir configuraciones como cadenas de conexión a bases de datos, permisos de acceso al sitio web, configurar servicios o definir comportamientos de ejecución del sitio web.

```
<configuration>
  <connectionStrings>
    <add name="BaseDatos" connectionString="data
     source=.\SQLEXPRESS;Integrated
     Security=SSPI;AttachDBFilename=|DataDirectory|\Database.mdf;User
     Instance=true"
       providerName="System.Data.SqlClient" />
  </connectionStrings>
```

```
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
...
...
```

En las líneas XML anteriores, podemos ver un fragmento del archivo **Web.config**. Las primeras líneas definen una cadena de conexión a una base de datos local de **SQL Server Express**; así como el nodo **<system.web>** muestra algunas opciones de compilación de la aplicación web, y especifica que se usará Microsoft .Net Framework versión **4.0** y que esta será realizada en modo de depuración.
El tener la conexión a la base de datos declarada en un archivo externo editable puede ser una ventaja debido a que podemos modificarla en cualquier momento sin tener la necesidad de recompilar la aplicación web además de simplificar el acceso a la cadena de conexión desde un único lugar y no tenerlo dispersado en todo el código.

```
protected void Page_Load(object sender, EventArgs e)
{
    SqlConnection con = new
SqlConnection(ConfigurationManager.ConnectionStrings["BaseDatos"].ToString());
    SqlCommand com = new SqlCommand("select * from usuarios", con);
    con.Open();

    SqlDataReader dr = com.ExecuteReader();
    while (dr.Read())
    {
        Response.Write(dr["Nombre"] + "<br>");
    }

    dr.Close();
    con.Close();
}
```

El código anterior hace uso de la cadena de conexión declarada en el archivo **Web.config** para conectarse a la base de datos y leer registros de una tabla. Este código no varía de lo que ya hemos aprendido en el **Capítulo 4**, solo agregamos una línea para poder leer esta configuración mediante el uso del objeto **ConfigurationManager** (administración de configuraciones).
Este objeto contiene una propiedad específica para leer las cadenas de conexiones declaradas en el archivo **Web.config** y, colocando el nombre que le asignamos a dicha

cadena, este nos la retornará. Gracias a esto, no se requerirá replicar la cadena de conexión en cada lugar del código que la necesitemos ya que, por medio de este objeto, podríamos asegurarnos de obtener aquella que hemos declarado en el archivo de configuraciones de la aplicación web.

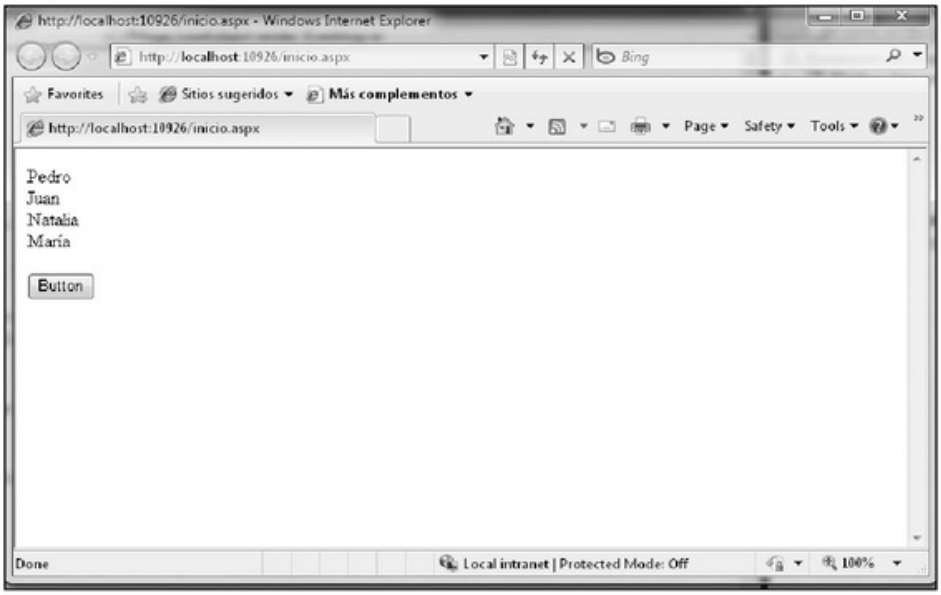


Figura 14. *La conexión y la lectura de datos de una base datos es idéntica a la realizada en aplicaciones de consola y de escritorio. Esto nos facilita la creación de código en distintos ambientes con solo aprender las particularidades de cada uno.*

El archivo **Web.config** puede resultar muy amplio en la cantidad de configuraciones que puede admitir. Como ya dijimos, para poder ver todas sus variantes, es recomendable que visitemos el sitio web oficial de Microsoft **http://msdn.microsoft.com/es-es/library/ff400235%28VS.100%29.aspx**, donde encontraremos una referencia completa de este archivo.

El archivo Global.asax

El último archivo vital que compone los sitios web de ASP.net es el archivo **Global.asax**. Este archivo no posee una interfaz gráfica como las páginas ASP.net, pero sí cuenta con código C# que afectará a toda la aplicación web que estemos desarrollando.

```
public class Global : System.Web.HttpApplication
{
    void Application_Start(object sender, EventArgs e)
    {
     ...
     ...
    }
```

```
    void Application_End(object sender, EventArgs e)
    {
     ...
     ...
    }

    void Application_Error(object sender, EventArgs e)
    {
     ...
     ...
    }

    void Session_Start(object sender, EventArgs e)
    {
     ...
     ...
    }

    void Session_End(object sender, EventArgs e)
    {
     ...
     ...
    }
}
```

El código anterior muestra la estructura del archivo **Global.asax**. En este, encontramos una serie de eventos que se ejecutan a nivel de aplicación; esto quiere decir que se ejecutarán sin importar si el usuario está visitando una página dentro de nuestro sitio web. Por ejemplo, el evento **Application_Start** se ejecutará cuando el servidor web donde esté alojado nuestro sitio web dé inicio a nuestra aplicación. En la **Tabla 1**, podemos ver la descripción y el uso de cada uno de estos eventos.

III REDIRECCIONAR LAS LLAMADAS

Para redireccionar un usuario desde una página a otra, podemos utilizar **Response.Redirect**. Este no es el único mecanismo disponible. Para realizar una redirección similar, podemos utilizar **Server.Transfer**. En este caso, la página actual dejará de escribir el código HTML para comenzar a escribir el código HTML de la página a la cual estamos redirigiendo al usuario.

EVENTO	DESCRIPCIÓN
Application_Start	El código contenido en este evento se ejecuta cuando la aplicación es iniciada por el servidor que la administra.
Application_End	Este evento será disparado cuando la aplicación finaliza, por ejemplo, si es detenida por el servidor que la administra.
Session_Start	Cuando un usuario ingresa por primera vez a nuestro sitio web, se le asigna una nueva sesión. Al crearse esta sesión, este evento es disparado.
Session_End	Este evento se ejecuta cuando una sesión caduca por tiempo de inactividad.
Application_Error	Si durante la ejecución de código no capturamos algún error que pudiera haber ocurrido, este evento se ejecutará antes de mostrar el mensaje del error.

***Tabla 1.** Lista de eventos globales de una aplicación ASP.net.*

Vimos qué ocurría cuando un control se encontraba fuera de una etiqueta de formulario en una página web. Este error era mostrado al usuario final reemplazando la página web del control por otra con el mensaje de error. Si bien esto puede ser útil, cuando estamos desarrollando para el usuario final puede resultar incomprensible; incluso, podríamos estar mostrando información crítica a personas que no deberían enterarse de esta. Una solución a este tipo de problemas sería agregar código en el archivo **Global.asax** para el evento **Application_Error**, que se ejecutará cuando ocurra un error en la aplicación web con la que estamos trabajando.

***Figura 15.** Al colocar un punto de interrupción en el evento **Application_Error**, vemos que la ejecución del código se detiene en el momento en que se produce un error en la ejecución de la aplicación.*

Sin escribir código, el error seguirá su curso una vez que haya pasado por nuestro punto de interrupción, por lo que será necesario agregar algunas líneas de código para realizar alguna acción que alerte al usuario de que ha ocurrido este error. Para este

caso, podríamos adicionar una nueva página **HTML**, que no ejecute código ASP.net. Esta página será tratada como simple código **HTML** estándard y no como una página ASP.net, por lo tanto, no ejecutará código de este tipo si este fuera escrito en ella.

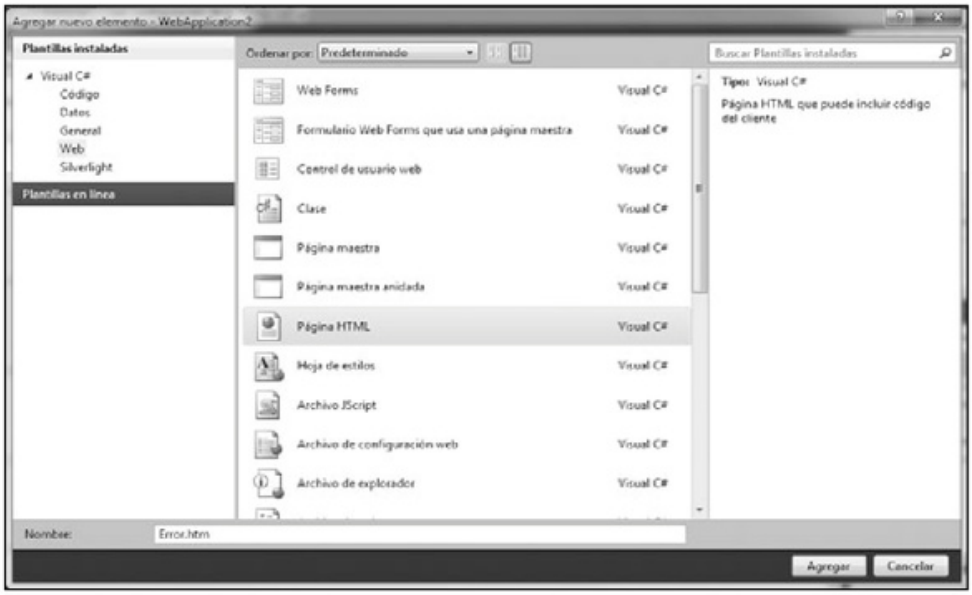


Figura 16. *Podemos agregar todos los tipos de archivos utilizados en la creación de páginas web a nuestra solución de ASP.net.*

```
<html>
<head>
    <title></title>
</head>
<body>
    <h1>
    Ocurrió un error
    </h1>
</body>
</html>
```

El código HTML anterior representará el mensaje de error que mostraremos al usuario cuando, en la aplicación, se encuentre un problema que no hayamos previsto. Finalmente, agregaremos una línea de código dentro del evento **Application_Error** que mueva al usuario desde la página que quería visualizar y produjo el error hasta la página HTML con el mensaje.

```
void Application_Error(object sender, EventArgs e)
{
```

```
    Response.Redirect("Error.htm");
}
```

Mediante el uso de **Response.Redirect** (redireccionar la respuesta), podemos especificar la página a la cual queremos dirigirnos; en este caso, el usuario será llevado desde la página que ocasionó la ejecución de **Application_Error** a la página **Error.htm**.

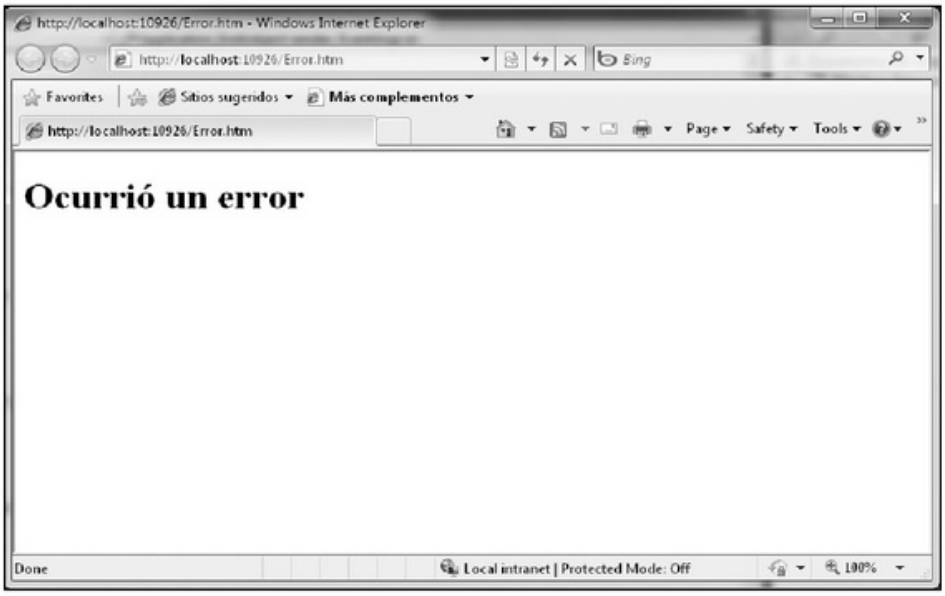


Figura 17. *En vez de mostrarse la página con el detalle del error producido por no haber colocado el control ASP.net dentro de la etiqueta de formulario, el usuario es redirigido a la página HTML que muestra un mensaje más adecuado.*

MÁS ALLÁ DE C#

Durante este capítulo, hemos visto que la mayoría del código, como por ejemplo las conexiones a las bases de datos o la declaración de variables, es igual al código que hemos realizado hasta el momento en otro tipo de aplicaciones. A pesar de esto, ASP.net posee algunas particularidades para manejar los estados de las variables y mantenerlas, poder saber si se ha realizado algún envío de información desde el cliente hacia el código C# que mantiene la página ASP.net asociada, o poder manipular el flujo de las páginas web con las que el usuario está interactuando.

Postback

El proceso de enviar una petición con información desde el cliente hacia el servidor, que este la procese y, luego, retorne un resultado típicamente en formato

HTML recibe el nombre de **postback**. Debido a que en ASP.net las páginas son manejadas por código C# asociada a ellas, el envío de esta información se realiza sobre esta página, esto es, configurando el atributo **action** (acción) del formulario de la página con el nombre de esta página. Cada vez que se realiza un **postback**, una serie de eventos se disparan en secuencia en el código C# independientemente de qué control hubiera causado el envío de la información.

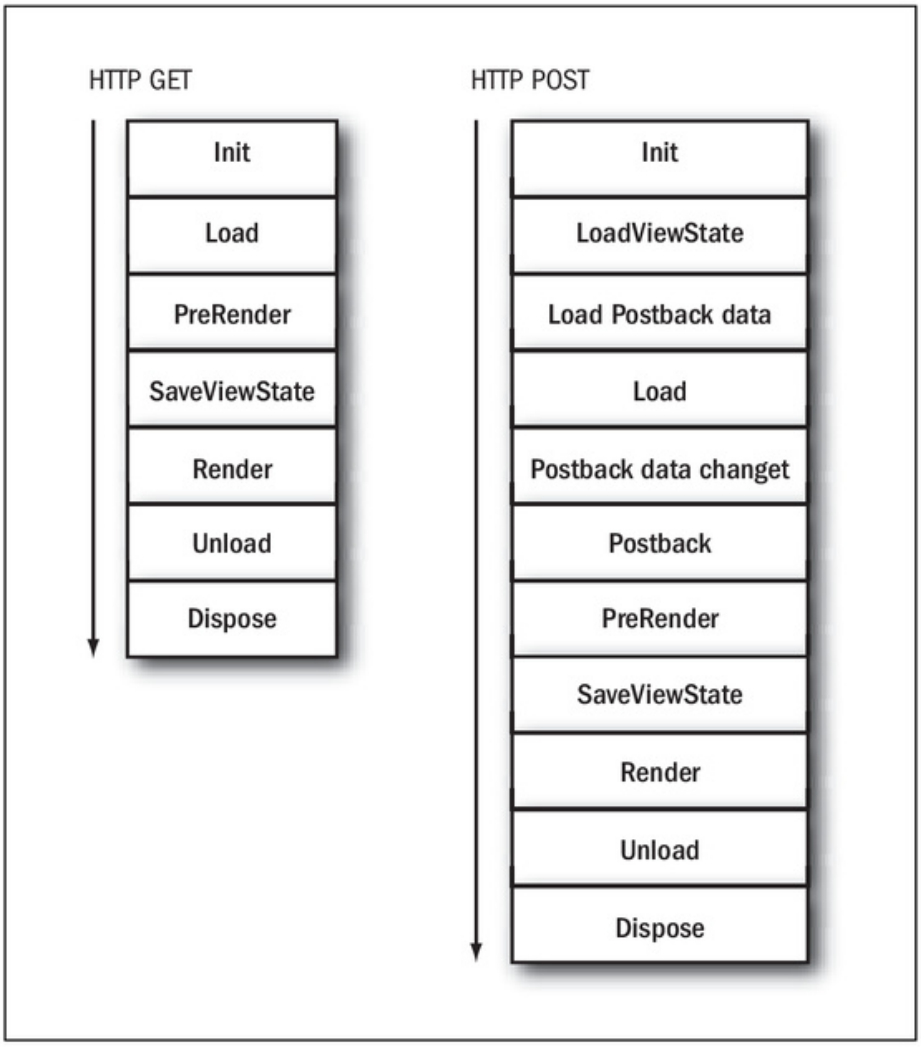


Figura 18. *Dependiendo del método de envío de información que tengamos, una página ASP.net ejecutará una serie de eventos en secuencia sin importar quién hubiera provocado este envío de información.*

Por lo tanto, si un botón provoca un **postback**, primero se ejecutarán los eventos de la **Figura 18** hasta el evento **Load**, luego el evento del botón y, por último se seguirá con la ejecución de los demás eventos. Esto puede traernos un problema notorio si tuviéramos que ejecutar alguna funcionalidad la primera vez que la página es cargada. Al principio, podríamos colocar nuestro código dentro del evento **Load** de la página, pero, cada vez que el usuario envíe información desde la página hacia nuestro código, este evento se ejecutaría volviendo a ejecutar nuestra funcionalidad.

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("Hola, bienvenido por primera vez");
}
```

Este código simplemente escribe un mensaje en la página cada vez que esta se ejecuta.

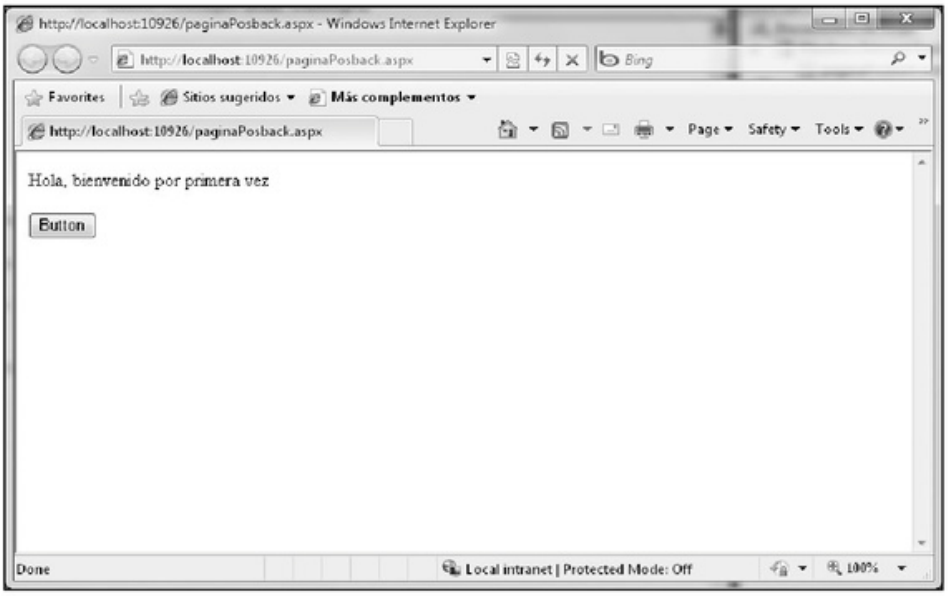


Figura 19. *Al ejecutar la página, se mostrará el mensaje. Si el usuario presiona sobre el botón de la página, el evento* ***Load*** *volverá a ejecutarse.*

Si ejecutamos en forma reiterada el código, siempre recibiremos el mismo mensaje. Es posible evitar esto y controlar el flujo de ejecución de una página haciendo uso de variables globales provistas por la página con la que estamos trabajando para saber el estado de las peticiones.

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        Response.Write("Hola, bienvenido por primera vez");
    }
    else
    {
        Response.Write("Mensaje después del postback");
    }
}
```

En el código anterior, usamos la variable **IsPostBack** (es una petición postback) para saber el estado en el cual nos encontramos. Si la variable contiene el valor **false** (falso), eso querrá decir que es la primera vez que el usuario está accediendo a esta página, por el contrario, si el valor es **true** (verdadero) significará que la llamada al código C# se está realizando por medio de una nueva petición por parte del usuario mientras que se encontraba en la misma página.

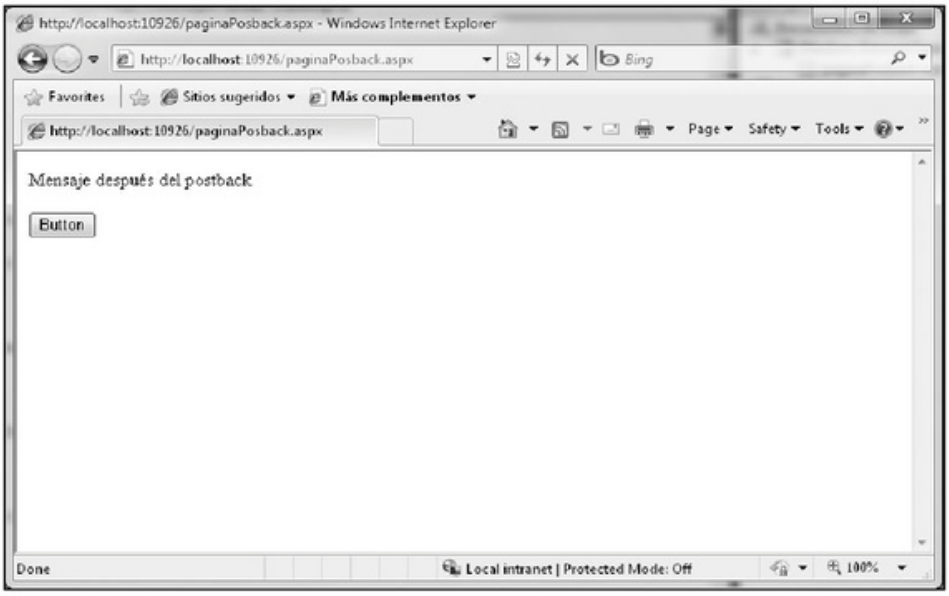


***Figura 20.** La segunda vez que el código C# que administra la página es llamado, la variable IsPostBack es igual a true y muestra el segundo mensaje. Todas las demás veces que este código sea ejecutado por medio de una petición, esta variable será igual a true.*

La variable **IsPostBack** volverá a contener el valor **false** cuando el usuario navegue hacia otra página y vuelva a esta. Esto se considerará como una llamada inicial a la página, por lo que se ejecutará el código dentro del fragmento verdadero de la sentencia **if**.

Manipular el flujo de ejecución

En el entorno de desarrollo de ASP.net, existen dos objetos que brindan soporte para la manipulación de las peticiones con envío de parámetros realizados por los

APP.CONFIG

Cuando estamos realizando aplicaciones web con ASP.net, podemos utilizar el archivo **Web.config** para configurar la aplicación que estamos construyendo. De igual forma, las aplicaciones de escritorio realizadas con Microsoft .Net cuentan con el archivo **app.config**. Este archivo puede ser utilizado para configurar la aplicación sin necesidad de recompilarla.

usuarios, así como la escritura de texto o el redireccionamiento de solicitudes a otras páginas web. A pesar de que ASP.net se encarga de administrar estas peticiones por medio de los controles que tengamos en nuestras páginas, muchas veces necesitaremos poder manejar estos valores por nuestra cuenta a un nivel más detallado.

Request

El objeto **Request** (solicitud) sirve para el manejo de las peticiones realizadas desde el cliente hacia el servidor web. En los ejemplos anteriores, cuando el usuario presionaba sobre un botón, o la página se cargaba por primera vez, esta era considerada una solicitud. En estos casos, el objeto **Request** contendría toda la información que es enviada desde el usuario hacia nuestro código.

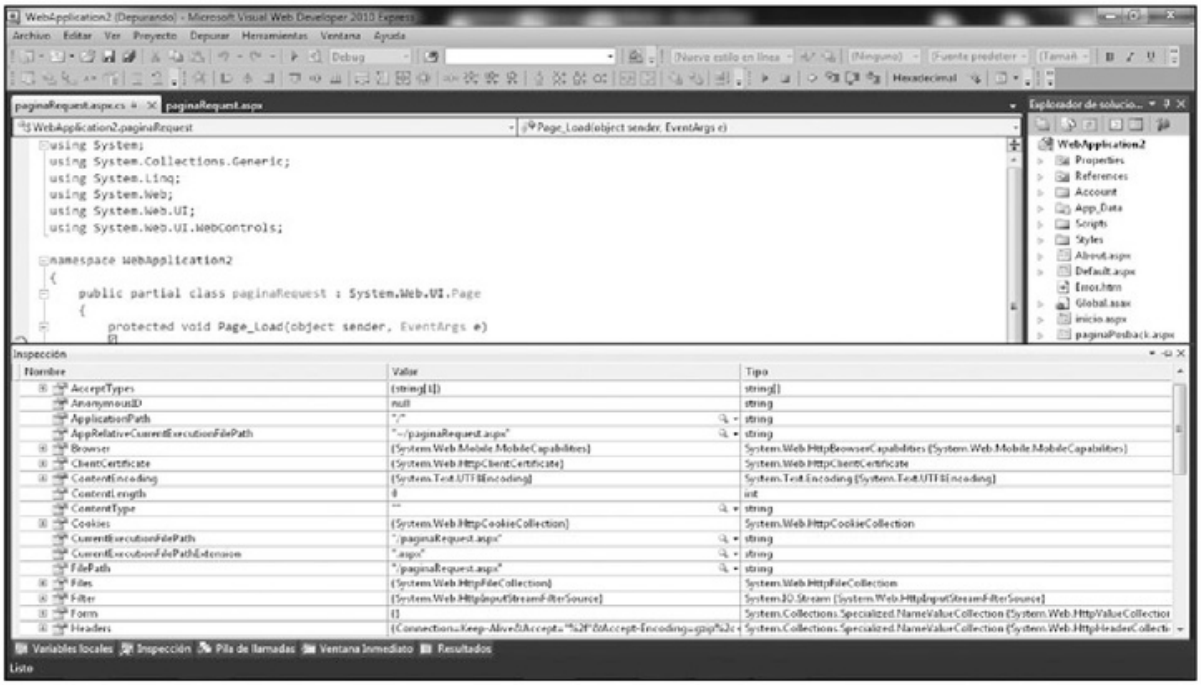


Figura 21. *Al inspeccionar el objeto* ***Request*** *en una petición realizada hacia nuestra página, este contiene información relacionada al usuario que está realizando la petición, además de información que es incluida en esta petición.*

```
Response.Write(Request.UserAgent + "<br>");
Response.Write(Request.UserHostAddress + "<br>");
Response.Write(Request.UserLanguages[0] + "<br>");
Response.Write(Request.Browser.Browser + "<br>");
Response.Write(Request.Browser.IsMobileDevice + "<br>");
Response.Write(Request.Browser.Version + "<br>");
```

En el código, podemos ver una lista de algunas de las propiedades a las cuales podemos acceder desde el objeto **Request**. En este código, obtenemos el nombre

completo y las capacidades del navegador web con el que se está conectando a nuestro sitio web, la dirección **IP** desde donde accede a él, la lista de idiomas que soporta o que tiene configurada, si es un dispositivo móvil el que intenta acceder y la versión de navegador que está usando.
Toda esta información de gran utilidad para poder personalizar el sitio web, en base al usuario que lo visita o dispositivo que use.

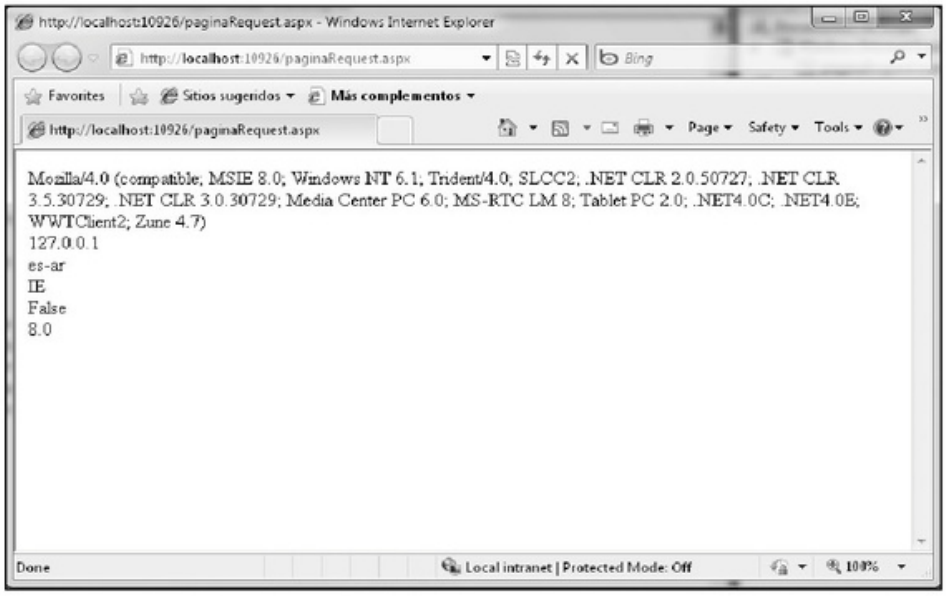


***Figura 22.** El uso del objeto **Request** de gran utilidad para conocer las preferencias y las capacidades del navegador. Al saber los idiomas configurados en el navegador, podemos hacer que el sitio se muestre automáticamente en el idioma de su preferencia.*

Response

En sentido inverso, el objeto **Response** (respuesta) es utilizado para generar salidas desde el código C# que administra la página web hacia el cliente. En el transcurso de este capítulo, hemos visto dos de las principales funcionalidades de este objeto. Por un lado, el uso de **Redirect** (redirigir) para mover al usuario de una página a la otra, y, por otro, el uso de **Write** (escribir) para escribir texto en el HTML resultante de la página; este último se comporta de forma similar a **Console.WriteLine** de las aplicaciones de consola.

III NOMBRES DE PÁGINAS WEB

Cuando creamos sitios, debemos tener especial cuidado con los nombres que utilicemos para los archivos y para las páginas web. Debido a convencionalismos de Internet, no podremos utilizar caracteres latinos como la letra eñe o la tilde, así como no es recomendable el uso de espacios, de lo contrario, nuestro sitio web podría no funcionar de manera correcta.

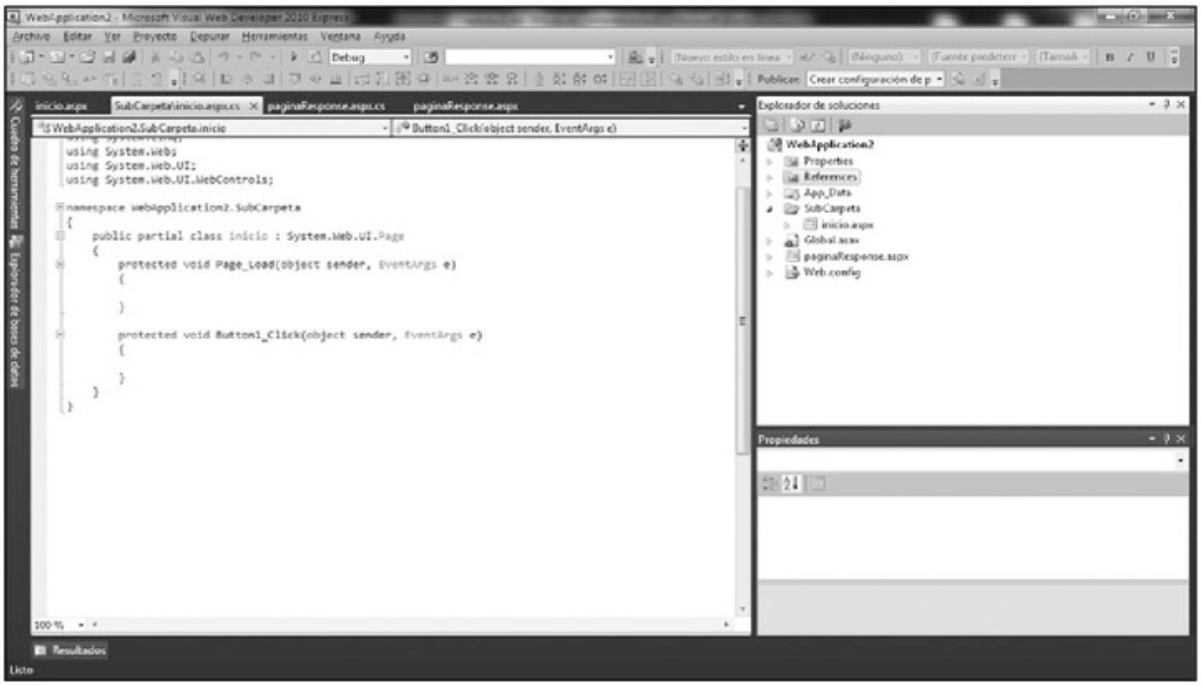


Figura 23. *Las páginas ASP.net pueden estar organizadas en diferentes carpetas. Para poder acceder a ellas, será necesario escribir la ruta relativa a la página desde el punto en el que nos encontremos.*

En la **Figura 23**, vemos una estructura de carpetas dentro de nuestra aplicación; dentro de la carpeta **SubCarpeta** tenemos una página, y otra, a nivel de la raíz del proyecto. Para poder navegar desde una página hacia otra, es necesario colocar la ruta relativa de acceso desde la página en la cual estemos posicionados actualmente.

```
protected void Button1_Click(object sender, EventArgs e)
{
    Response.Redirect("SubCarpeta/inicio.aspx");
}
```

Desde la página que se encuentra en la raíz del sitio web, es necesario colocar la ruta completa de acceso a la página que se encuentra dentro de la carpeta **SubCarpeta**.

III USO DEL CARÁCTER ~

Para navegar de una página a otra o acceder a distintos recursos como imágenes o archivos, es posible hacer referencia a estos mediante una ruta lógica. En ASP.net, podremos utilizar el carácter ~ para que, desde cualquier parte del árbol del sitio, se haga referencia a su raíz. Esto nos facilitará la resolución de rutas dinámicas a estos recursos.

En este código, cuando el usuario presione el botón, será redirigido hacia la página **inicio.aspx** de la carpeta **SubCarpeta**.

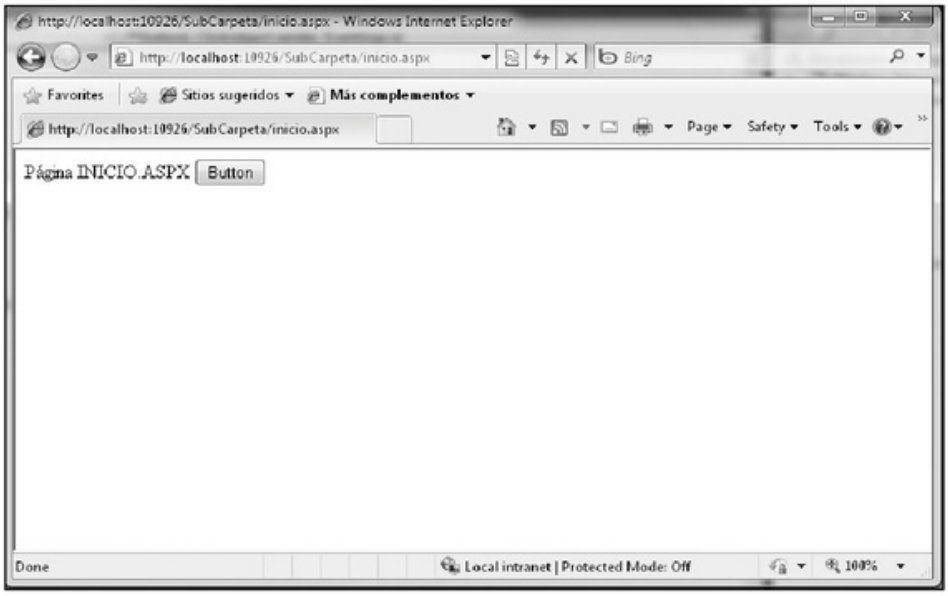


Figura 24. *Al presionar el botón, el usuario es redirigido a la página que se encuentra dentro de la subcarpeta en el árbol del sitio web.*

Para poder volver hacia la página que se encuentra en la raíz del sitio, desde la página de la subcarpeta podremos volver a usar la función **Redirect**, pero con cambios.

```
protected void Button1_Click(object sender, EventArgs e)
{
    Response.Redirect("~/paginaResponse.aspx");
}
```

La línea de código anterior hace referencia a la raíz del sitio web donde se encuentra la página de internet en cuestión. Notemos que usamos el caracter especial ~ para especificar la raíz del sitio Web. ASP.net traducirá este caracter tomándolo como la base del sitio.

Conservar información

Ya hemos comprobado que las variables y los distintos objetos se destruyen después de que la página completa su ejecución. Esto hace que debamos cambiar la forma en la que realizamos nuestro código y tratar de no depender de los datos almacenados en las variables. De cualquier manera, muchas veces necesitaremos preservar algunos datos mientras el usuario realiza diferentes acciones sobre nuestro sitio web y, para esto, ASP.net nos provee de tres objetos con diferentes alcances.

ViewState

El primer objeto que podemos usar para almacenar información es el llamado **ViewState** (estado de la vista). Este objeto nos permite almacenar distintos objetos sobre la base de una llave o clave de texto en el momento en que el código que maneja nuestra página es ejecutado. Luego pueden ser recuperados la siguiente vez que se realiza una petición a la página.

```
private int sumador = 0;

protected void Button1_Click(object sender, EventArgs e)
{
     sumador++;
     ViewState.Add("Dato", sumador);
}
```

La ejecución de **ViewState.Add** creará una nueva entrada en la colección de datos llamada **Dato**, que contiene el valor de la variable **sumador**. El nombre **Dato** es utilizado como una referencia al objeto contenido, por lo que podemos utilizar cualquier nombre para este valor.

```
protected void Button2_Click(object sender, EventArgs e)
{
     sumador = (int)ViewState["Dato"];
     sumador++;
     ViewState["Dato"] = sumador;
}
```

En las siguientes iteraciones con el código, es posible recuperar el objeto previamente almacenado y volver a asignarlo a la variable **sumador**. Tengamos en cuenta que la variable **sumador** sigue perdiendo su información tras cada petición, por

III LUGAR DE ALMACENAMIENTO DE VIEWSTATE

Para preservar información entre peticiones al servidor, podemos utilizar el **ViewState**. De cualquier manera, deberemos tener cuidado cuando hagamos uso de este recurso ya que toda la información almacenada es encriptada y enviada junto con el HTML de la página web. Esto puede provocar que el tamaño de esta crezca y haga que su transferencia sea más lenta.

lo tanto, es necesario asignarle el último valor que se haya almacenado en el contenedor **ViewState**. Una vez asignado este, podemos seguir realizando operaciones para luego volver a guardar, sobrescribiendo el objeto del **ViewState**, el valor de la variable hasta que vuelva a ser consumido.

***Figura 25.** Tras algunas iteraciones con la página, podemos ver que la variable **sumador** contiene un valor superior a 1 ya que ha estado sumando 1 al valor previamente almacenado en ViewState.*

Los objetos almacenados en **ViewState** solo existirán mientras el usuario interactúe con esta página. Si el usuario es redirigido a otra página y luego vuelve a la inicial, todos los valores de **ViewState** se habrán destruido. Esto se debe a que los objetos del **ViewState** son escritos por ASP.net en el código HTML de la página, por lo que, cuando el usuario se va de dicha página y vuelve a esta, ASP.net considerará que es una petición completamente nueva y no una llamada más sobre esta página.

Session

El objeto **Session** (sesión) tiene un funcionamiento sintáctico idéntico a **ViewState**. En este objeto, podremos almacenar información y objetos con datos, para luego poder recuperarlos. La principal diferencia es que los objetos almacenados en **Session** perdurarán a través de todas las páginas por las cuales navegue el usuario. Estas variables serán únicas a cada usuario, por lo que, si tenemos más de un usuario viendo nuestro sitio web, los datos de una variable de sesión creada para el usuario **A** no se compartirán con el usuario **B** por más que estas variables tengan este nombre.
Este tipo de variables son las comúnmente usadas en los sitios web para almacenar la identidad de un usuario en un momento que este se autentifica.

```
//Código de página principal
protected void Page_Load(object sender, EventArgs e)
{
    Session.Add("IDUsuario", 1234);
    Response.Redirect("Session2.aspx");
}

//Código de página Session2.aspx
protected void Page_Load(object sender, EventArgs e)
{
    int idUsuario = (int)Session["IDUsuario"];
}
```

El código crea una variable de sesión llamada **IDUsuario** con el valor **1234** y, luego, redirige al usuario a una página secundaria la cual toma el valor de la variable de sesión para su uso posterior, como vemos en la figura siguiente.

Figura 26. *El valor de la sesión es recuperada correctamente desde la página secundaria.*

Debemos tener especial cuidado cuando trabajemos con variables de sesión, ya que su contenido es manejado por el servidor y ocupa memoria. Almacenar grandes objetos en este tipo de variables podría causar muchos problemas de desempeño del servidor y de nuestro sitio web. Por este motivo, estas variables tienen un tiempo de vida corto marcado por la actividad del usuario sobre el sitio web. Esto quiere decir que el servidor destruirá su valor cuando pase determinado tiempo de inactividad por parte del

usuario. Por defecto, el tiempo de vida de una variable de sesión es de **20** minutos, pero podemos configurar este valor según lo requieran nuestras necesidades.

```
//El tiempo de vida de la variable de sesión
//es de 50 minutos
Session.Timeout = 50;
```

El código anterior configura el tiempo de vida de la variable de sesión a **50** minutos. Esto hará que el usuario pueda estar 50 minutos sin enviar ningún tipo de petición al servidor donde reside nuestra aplicación, y sus datos se mantendrán activos. Debido al consumo de recursos ocasionados por este tipo de variables, es posible eliminarlas por completo sin tener que esperar que se destruyan por inactividad.

```
//Eliminamos las variables para este usuario
Session.Abandon();
```

Application

El último objeto que nos servirá para almacenar información es **Application** (aplicación). Se emplea de igual forma que **ViewState** y **Session**, con la diferencia que posee un rango de acceso mucho mayor. Los objetos almacenados dentro de **Application** serán compartidos por todos los usuarios en toda la aplicación. Estas variables estarán activas en todo el curso de la aplicación sin tener un tiempo prefijado para su destrucción. Debido a que estas variables se comparten con todos los usuarios de la aplicación web, sería muy sencillo realizar un contador de visitas simple.

```
void Application_Start(object sender, EventArgs e)
{
    Application.Add("Contador", 0);
}
void Session_Start(object sender, EventArgs e)
{
    int contador = (int)Application["Contador"];
    contador++;
    Application["Contador"] = contador;
}
```

Si hacemos uso del archivo **Global.asax**, creamos una nueva variable de aplicación en el evento que se disparará cuando nuestro sitio web sea inicializado por primera

vez. Luego, cada vez que un usuario se conecte a nuestras páginas y una variable de sesión sea creada, adicionamos **1** a la variable de aplicación creada con anterioridad. Desde cualquier página, podremos leer el valor de la variable de aplicación y consultar su valor. Debido a que este valor es compartido por todos los usuarios, este contador se lo mostrará a todos ellos, viendo en todo momento el último valor, incrementado por la última interacción realizada por el usuario que ingrese al final.

```
protected void Page_Load(object sender, EventArgs e)
{
    int contador = (int)Application["Contador"];
    Response.Write("Total de visitas: " + contador.ToString());
}
```

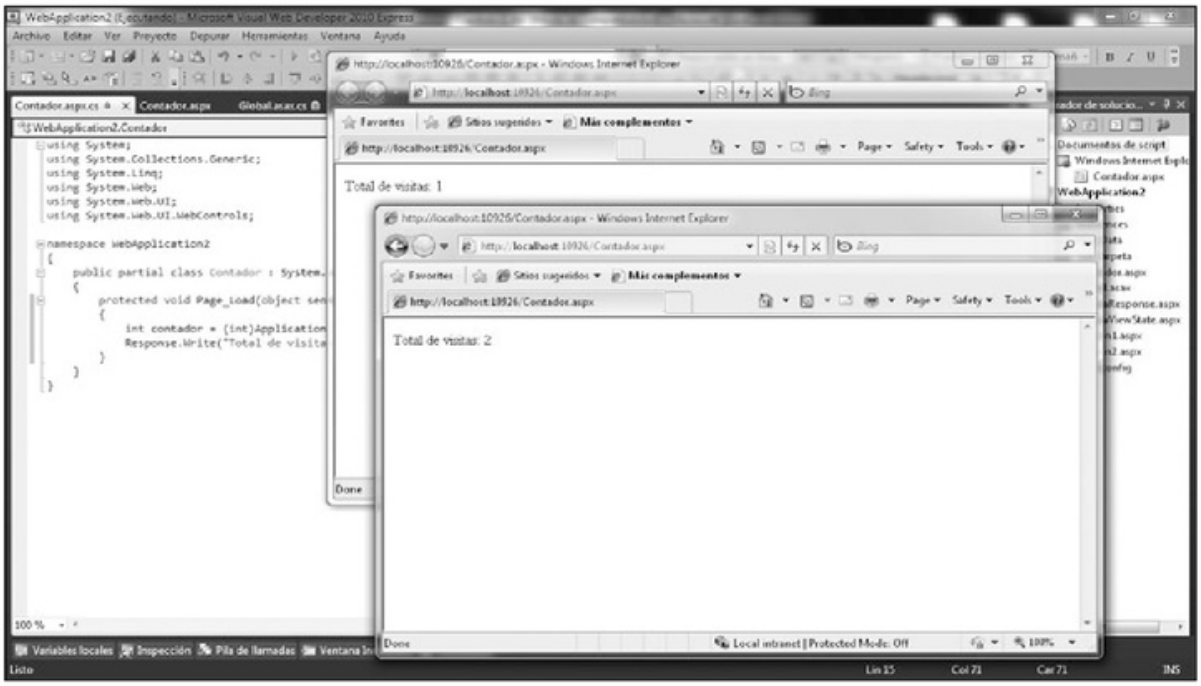


Figura 27. *La primera ventana representa al primer usuario y muestra el valor al momento que ingresó al sitio web. En la segunda ventana, vemos que, como ya ingresó un usuario previamente, se le suma 1 a la variable de aplicación.*

III TIEMPO DE VIDA

Cuando definimos el tiempo de vida de las variables de sesión, debemos tener especial cuidado ya que un tiempo de vida muy corto podría hacer que el usuario perdiera información muy rápido, si no realiza peticiones al servidor de manera constante. Por el contrario, un tiempo de vida muy largo causaría que las variables no se destruyeran.

EL ENTORNO DE ASP.NET

Los controles ASP.net poseen una nomenclatura específica, por lo que no podría ser interpretada por los distintos navegadores web. Por lo tanto, ASP.net toma estas declaraciones y las transforma a etiquetas HTML estándares.

Controles genéricos

ASP.net nos provee de controles para el desarrollo de páginas web de igual forma que lo hace para el desarrollo de aplicaciones de escritorio. Si estamos diseñando una página, podremos ver el **Cuadro de herramientas** con una lista extensa de controles.

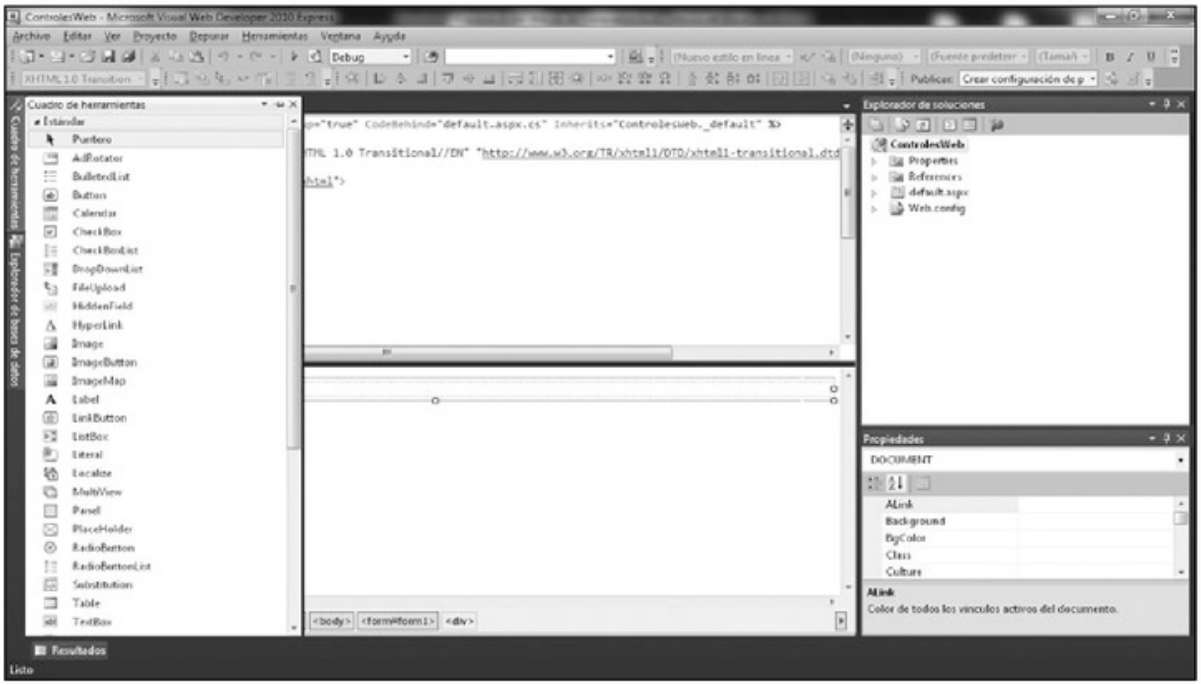


***Figura 28.** El Cuadro de herramientas nos provee todos los controles ASP.net necesarios para el desarrollo de aplicaciones web. Estos controles serán transformados, en tiempo de ejecución, a código HTML estándar.*

```
<form id="form1" runat="server">
<div>
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    <asp:Button ID="Button1" runat="server" Text="Button" />
</div>
</form>
```

Hemos agregado dos controles ASP.net, uno que representará una caja de texto y otro, un botón. Ambos controles poseen ciertas coincidencias que los definen como

controles de ASP.net y que son obligatorias para su correcto funcionamiento. En la **Tabla 2**, podemos ver las propiedades necesarias para definir un control ASP.net se listan las principales propiedades, necesarias, para la definición de un control ASP.net. La ausencia de las mismas hará que el código arroje errores.

PROPIEDAD	DESCRIPCIÓN
<asp:[Tipo de Control] ...>	Todos los controles ASP.net se inician con la declaración <asp: seguida del nombre o tipo de control que queremos declarar.
ID	Representa el nombre del control dentro de la página ASP.net y del código C#. Por medio de este identificador, podremos acceder a las propiedades del control.
RunAt	La única propiedad permitida es Server y especifica que el control será tratado por el servidor.

***Tabla 2.** Lista de características y atributos obligatorios de los controles ASP.net.*

Al ejecutar la página web del código anterior, podremos ver que los dos controles se muestran de manera correcta en el navegador web: tanto la caja de texto como el botón se muestran sin problemas. Esto se debe a que, en el momento de su ejecución, fueron transformados a código HTML estándar.

***Figura 29.** Una vez procesado el control ASP.net, es transformado a código HTML estándar.*

HTTPHANDLER Y HTTPMODULE

Los HTTP Handlers y HTTP Modules son de gran utilidad cuando necesitamos separar funcionalidad de interfaces visuales (Páginas ASP.net). De esta forma el código C# se ejecutará retornándole información al navegador sin contener código HTML. Visite el siguiente sitio Web para conocer más sobre estos manejadores: **http://msdn.microsoft.com/en-us/library/5c67a8bd%28v=VS.85%29.aspx**.

Los controles poseen otra particularidad: como son administrados por el servidor a diferencia del desarrollo de aplicaciones web con otros lenguajes, ASP.net es quien mantiene el estado de los controles; esto quiere decir que, entre cada **postback**, el estado de los controles seguirá siendo el mismo. Si un usuario hubiese escrito un valor dentro de una caja de texto y luego produce un **postback**, cuando la página fuera devuelta al usuario, los valores introducidos serán mantenidos de forma automática.

Enlace de datos

Los controles utilizados para mostrar datos como grillas o listas desplegables funcionan de la misma forma que en las aplicaciones de escritorio. Estos controles pueden ser enlazados a colecciones de datos mediante la propiedad **DataSource** (fuente de datos).

```
List<Usuario> usuarios = new List<Usuario>();
usuarios.Add(new Usuario()
{
    ID = 1,
    Edad = 30,
    Nombre = "Juan"
});
...
...
this.GridView1.DataSource = usuarios;
this.GridView1.DataBind();
```

Junto con la propiedad **DataSource**, necesitaremos decirle al control que enlace los datos mediante la función **DataBind**. A diferencia de los controles de las aplicaciones de escritorio en ASP.net, es necesario llamar a esta última función para que el enlace de datos se realice. En el caso del control **GridView**, este detectará las propiedades del objeto que le estemos pasando y generará las columnas que requiera.

III JQUERY

Existen diferentes implementaciones que brindan una solución elegante para la utilización de AJAX; una de estas es **JQuery**, la cual proporciona acceso a diferentes objetos para la manipulación de peticiones asíncronas al servidor así como la manipulación del código HTML de una página web. Si tenemos conocimientos intermedios de JavaScript, este modelo puede resultar útil.

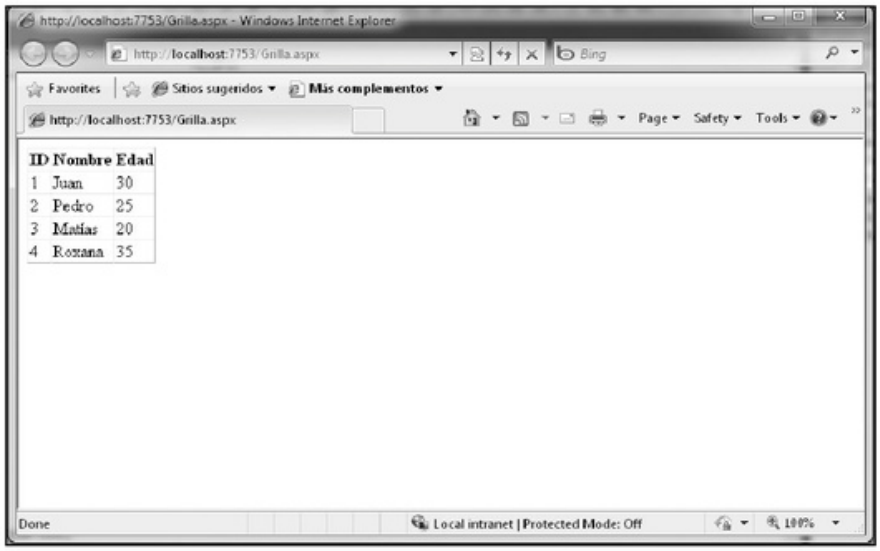


Figura 30. *Los controles ASP.net siempre son transformados a código HTML estándar. En el caso de un control GridView, este será transformado a una tabla HTML.*

Los controles **GridView** son declarados, al igual que cualquier control ASP.net, por medio de nodos no estándares HTML.

```
<asp:GridView ID="GridView1" runat="server">
</asp:GridView>
```

Pero, de igual forma que los demás controles, este control será transformado a un conjunto de elementos entendibles por los navegadores web.

```
<table cellspacing="0" rules="all" border="1" id="Table1" style="border-
     collapse:collapse;">
     <tr>
            <th scope="col">ID</th><th scope="col">Nombre</th><th
     scope="col">Edad</th>
     </tr><tr>
            <td>1</td><td>Juan</td><td>30</td>
     </tr><tr>
            <td>2</td><td>Pedro</td><td>25</td>
     </tr><tr>
            <td>3</td><td>Mat&#237;as</td><td>20</td>
     </tr><tr>
            <td>4</td><td>Roxana</td><td>35</td>
     </tr>
</table>
```

Muchos controles, además, poseen plantillas decorativas. En el caso específico del control **GridView**, este puede tomar ciertas características visuales de una lista preconfigurada. Este conjunto de plantillas visuales pre-configuradas resultan de gran ayuda para realizar un diseño rápido de nuestro sitio web. En especial, en los controles complejos como éste.

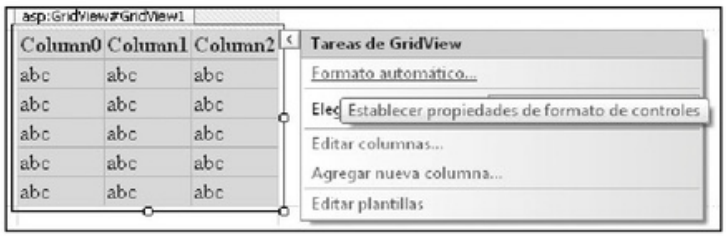


Figura 31. *Desde el control, podemos seleccionar diferentes plantillas preconfiguradas para modificar la apariencia de la grilla. Esta es una forma rápida de modificar la configuración visual de los controles.*

Desde la lista que se despliega cuando elegimos la opción **Formato automático**, podemos seleccionar cualquiera de las plantillas visuales. Esto nos ahorrará tiempo de desarrollo, ya que configurar cada valor de este control puede consumir mucho esfuerzo.

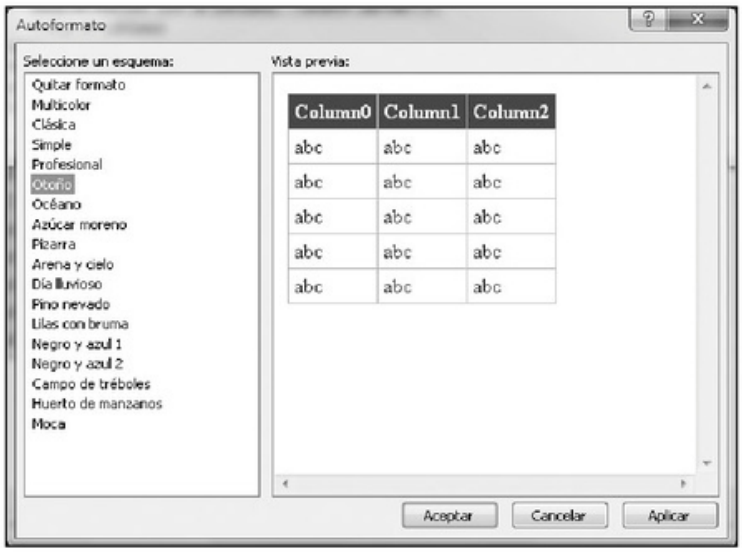


Figura 32. *La lista de configuraciones visuales preestablecidas nos resultará de utilidad cuando necesitemos crear sitios web rápidamente. Esto nos ahorrará esfuerzo de construcción de nuestro sitio.*

```
<asp:GridView ID="GridView1" runat="server" BackColor="White"
    BorderColor="#CC9966" BorderStyle="None" BorderWidth="1px" CellPadding="4">
    <FooterStyle BackColor="#FFFFCC" ForeColor="#330099" />
    <HeaderStyle BackColor="#990000" Font-Bold="True" ForeColor="#FFFFCC" />
    <PagerStyle BackColor="#FFFFCC" ForeColor="#330099"
```

```
            HorizontalAlign="Center" />
    <RowStyle BackColor="White" ForeColor="#330099" />
    <SelectedRowStyle BackColor="#FFCC66" Font-Bold="True" ForeColor="#663399" />
    <SortedAscendingCellStyle BackColor="#FEFCEB" />
    <SortedAscendingHeaderStyle BackColor="#AF0101" />
    <SortedDescendingCellStyle BackColor="#F6F0C0" />
    <SortedDescendingHeaderStyle BackColor="#7E0000" />
</asp:GridView>
```

Una vez seleccionada una de las plantillas, podemos ver, en el código anterior, todos los parámetros del control **GridView** que configura. Las configuraciones de colores y tipos de fuente para cada una de las propiedades podrían tomarnos bastante tiempo. Por supuesto, la versatilidad del control nos permite ajustar su aspecto como más nos guste.

... RESUMEN

En este capítulo, hemos realizado un recorrido por el mundo del desarrollo web con ASP.net. Una de las principales ventajas del uso de Microsoft .Net es la posibilidad de reutilizar los conocimientos adquiridos bajo otras plataformas, como el desarrollo de aplicaciones de consola o de escritorio para Windows, y llevarlos a áreas como la Web. En el caso de ASP.net, solo necesitamos aprender algunas claves sintácticas específicas para este ambiente ya que, como hemos aprendido, conceptualmente no resultará diferente de otros tipos de aplicaciones. De esta forma, hemos hecho un barrido completo por el mundo del desarrollo, y hemos aprendido tanto los conceptos iniciales como lo último en el desarrollo de aplicaciones, aprovechando la versatilidad del mundo .Net.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Cuál es la propiedad que usamos para enlazar datos a un control GridView?

2 ¿Para qué es utilizado el archivo Web.config?

3 Si necesitáramos administrar eventos relacionados con nuestra aplicación web, ¿dónde colocaríamos el código?

4 Si solo necesitáramos almacenar datos temporalmente en una única página web diferenciando los datos por usuario, ¿qué objeto deberíamos usar?

5 ¿Cuál es el nombre de la variable que nos permite saber si es la primera vez que una página es pedida por el usuario?

6 ¿En que formato son mostrados los controles ASP.net una vez ejecutada la página web?

7 ¿Cómo distinguimos una página web de otro tipo de archivos?

8 ¿Cuál es el mejor lugar para almacenar una cadena de conexión para un sitio web?

9 Si necesitáramos definir una zona en nuestra página web para que se cargue de forma parcial, ¿qué control de Microsoft AJAX deberíamos usar?

10 ¿Cuál es el nombre del componente que deberemos incluir para usar Microsoft AJAX?

EJERCICIOS PRÁCTICOS

1 Las páginas ASP.net poseen conceptos similares a las aplicaciones de escritorio. Intente crear un control de usuario para su página web imitando los pasos del capítulo anterior.

2 Utilizando variables de sesión, almacene los datos de un usuario tomado desde una base de datos.

3 En otra página ASP.net, controle que la variable de sesión se haya creado y contenga datos. De no ser así, redirija el usuario a la página en la cual la variable de sesión es cargada.

4 Con el contador de visitas que hemos hecho de ejemplo durante este capítulo, intente restar las visitas del sitio web cada vez que un usuario abandona ese sitio.

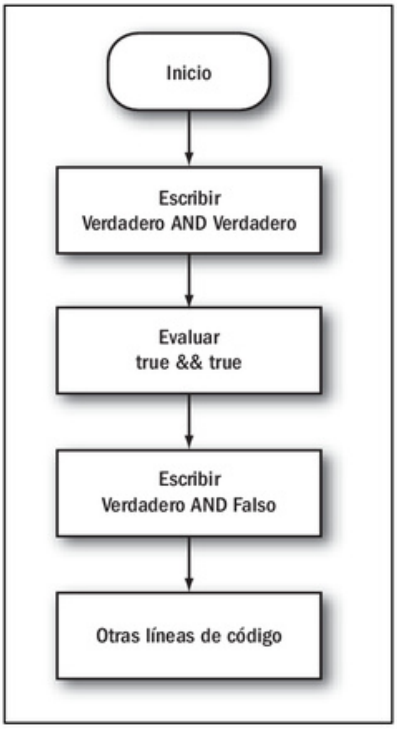
5 En ASP.net, podemos convertir controles HTML en controles manejados por el servidor solo agregando los atributos mínimos necesarios para que un control pueda ser manejado por el servidor. Agregue estos atributos a un control INPUT HTML e intente accederlo desde el servidor.

Servicios al lector

En este apartado final, incluimos el índice temático que nos permitirá acceder, en forma rápida y precisa, a los principales conceptos tratados en este libro sobre Programación.net.

ÍNDICE TEMÁTICO

E



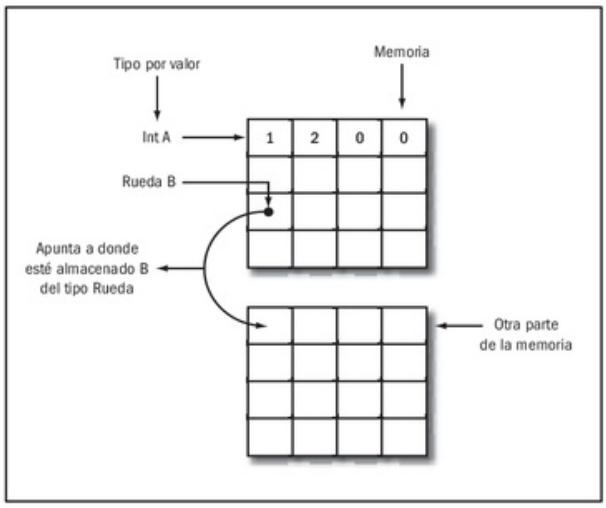
F

G

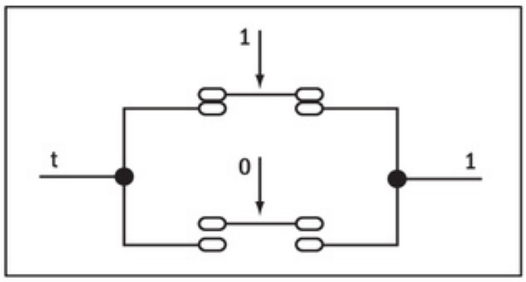
H

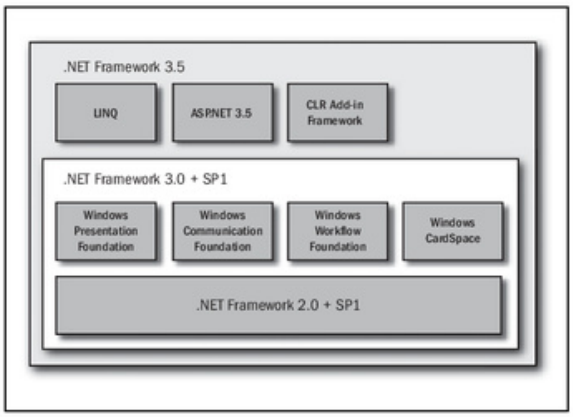
I

L



M





CLAVES PARA COMPRAR
UN LIBRO DE COMPUTACIÓN

1 SOBRE EL AUTOR Y LA EDITORIAL

Revise que haya un cuadro "sobre el autor", en el que se informe sobre su experiencia en el tema. En cuanto a la editorial, es conveniente que sea especializada en computación.

2 PRESTE ATENCIÓN AL DISEÑO

Compruebe que el libro tenga guías visuales, explicaciones paso a paso, recuadros con información adicional y gran cantidad de pantallas. Su lectura será más ágil y atractiva que la de un libro de puro texto.

3 COMPARE PRECIOS

Suele haber grandes diferencias de precio entre libros del mismo tema; si no tiene el valor en tapa, pregunte y compare.

4 ¿TIENE VALORES AGREGADOS?

Desde un sitio exclusivo en la Red, un Servicio de Atención al Lector, la posibilidad de leer el sumario en la Web para evaluar con tranquilidad la compra, y hasta la presencia de adecuados índices temáticos, todo suma al valor de un buen libro.

5 VERIFIQUE EL IDIOMA

No sólo el del texto; también revise que las pantallas incluidas en el libro estén en el mismo idioma del programa que usted utiliza.

usershop.redusers.com

Windows 7: Trucos y secretos

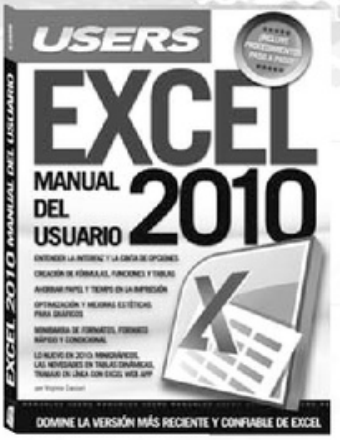
Este libro está dirigido a todos aquellos que quieran sacar el máximo provecho de Windows 7, las redes sociales y los dispositivos ultraportátiles del momento. A lo largo de sus páginas, el lector podrá adentrarse en estas tecnologías mediante trucos inéditos y consejos asombrosos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-17-6

Desarrollo PHP + MySQL

Este libro presenta la fusión de dos de las herramientas más populares para el desarrollo de aplicaciones web de la actualidad: PHP y MySQL. En sus páginas, el autor nos enseñará las funciones del lenguaje, de modo de tener un acercamiento progresivo, y aplicar lo aprendido en nuestros propios desarrollos.

→ COLECCIÓN: MANUALES USERS
→ 432 páginas / ISBN 978-987-1773-16-9



Excel 2010

Este manual resulta ideal para quienes se inician en el uso de Excel, así como también para los usuarios que quieran conocer las nuevas herramientas que ofrece la versión 2010. La autora nos enseñará desde cómo ingresar y proteger datos hasta la forma de imprimir ahorrando papel y tiempo.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-15-2

Técnico Hardware

Esta obra es fundamental para ganar autonomía al momento de reparar la PC. Aprenderemos a diagnosticar y solucionar las fallas, así como a prevenirlas a través del mantenimiento adecuado, todo explicado en un lenguaje práctico y sencillo.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-14-5

PHP Avanzado

Este libro brinda todas las herramientas necesarias para acercar al trabajo diario del desarrollador los avances más importantes incorporados en PHP 6. En sus páginas, repasaremos todas las técnicas actuales para potenciar el desarrollo de sitios web.

→ COLECCIÓN: MANUALES USERS
→ 400 páginas / ISBN 978-987-1773-07-7

AutoCAD

Este manual nos presenta un recorrido exhaustivo por el programa más difundido en dibujo asistido por computadora a nivel mundial, en su versión 2010. En sus páginas, aprenderemos desde cómo trabajar con dibujos predeterminados hasta la realización de objetos 3D.

→ COLECCIÓN: MANUALES USERS
→ 384 páginas / ISBN 978-987-1773-06-0

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.

Windows 7 Avanzado

Esta obra nos presenta un recorrido exhaustivo que nos permitirá acceder a un nuevo nivel de complejidad en el uso de Windows 7. Todas las herramientas son desarrolladas con el objetivo de acompañar al lector en el camino para ser un usuario experto.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1773-08-4

Photoshop

En este libro aprenderemos sobre las más novedosas técnicas de edición de imágenes en Photoshop. El autor nos presenta de manera clara y práctica todos los conceptos necesarios, desde la captura digital hasta las más avanzadas técnicas de retoque.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-05-3

Grabación y producción de música

En este libro repasaremos todos los aspectos del complejo mundo de la producción musical. Desde las cuestiones para tener en cuenta al momento de la composición, hasta la mezcla y el masterizado, así como la distribución final del producto.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1773-04-6

Linux

Este libro es una completa guía para migrar e iniciarse en el fascinante mundo del software libre. En su interior, el lector conocerá las características de Linux, desde su instalación hasta las opciones de entretenimiento, con todas las ventajas de seguridad que ofrece el sistema.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-8-6

Premiere + After Effects

Esta obra nos presenta un recorrido detallado por las aplicaciones audiovisuales de Adobe: Premiere Pro, After Effects y Soundbooth. Todas las técnicas de los profesionales, desde la captura de video hasta la creación de efectos, explicadas de forma teórica y práctica.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-9-3

Office 2010

En este libro aprenderemos a utilizar todas las aplicaciones de la suite, en su versión 2010. Además, su autora nos mostrará las novedades más importantes, desde los minigráficos de Excel hasta Office Web Apps, todo presentado en un libro único.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-26013-6-2

usershop.redusers.com

Excel Paso a Paso

En esta obra encontraremos una increíble selección de proyectos pensada para aprender, mediante la práctica, la forma de agilizar todas las tareas diarias. Todas las actividades son desarrolladas en procedimientos paso a paso de una manera didáctica y fácil de comprender.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-26013-4-8

C#

Este libro es un completo curso de programación con C# actualizado a la versión 4.0. Ideal tanto para quienes desean migrar a este potente lenguaje, como para quienes quieran aprender a programar desde cero en Visual Studio 2010.

→ COLECCIÓN: MANUALES USERS
→ 400 páginas / ISBN 978-987-26013-5-5

200 Respuestas: Seguridad

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos contestar para conseguir un equipo seguro. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-26013-1-7

Funciones en Excel

Este libro es una guía práctica de uso y aplicación de todas las funciones de la planilla de cálculo de Microsoft. Desde las funciones de siempre hasta las más complejas, todas presentadas a través de ejemplos prácticos y reales.

→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-26013-0-0

Proyectos con Windows 7

En esta obra aprenderemos cómo aprovechar al máximo todas las ventajas que ofrece la PC. Desde cómo participar en las redes sociales hasta las formas de montar una oficina virtual, todo presentado en 120 proyectos únicos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-036-8

PHP 6

Este libro es un completo curso de programación en PHP en su versión 6.0. Un lenguaje que se destaca tanto por su versatilidad como por el respaldo de una amplia comunidad de desarrolladores, que lo convierten en un punto de partida ideal para quienes comienzan a programar.

→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-663-039-9

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.

200 Respuestas: Blogs

Esta obra es una completa guía que responde a las preguntas más frecuentes de la gente sobre la forma de publicación más poderosa de la Web 2.0. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-037-5

Hardware paso a paso

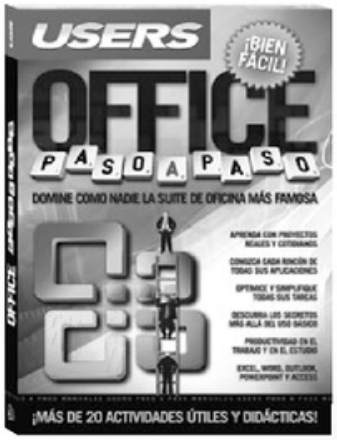
En este libro encontraremos una increíble selección de actividades que abarcan todos los aspectos del hardware. Desde la actualización de la PC hasta el overclocking de sus componentes, todo en una presentación nunca antes vista, realizada íntegramente con procedimientos paso a paso.

→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-034-4

200 Respuestas: Windows 7

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos conocer para dominar la última versión del sistema operativo de Microsoft. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-663-035-1



Office paso a paso

Este libro presenta una increíble colección de proyectos basados en la suite de oficina más usada en el mundo. Todas las actividades son desarrolladas con procedimientos paso a paso de una manera didáctica y fácil de comprender.

→ COLECCIÓN: PASO A PASO
→ 320 páginas / ISBN 978-987-663-030-6

101 Secretos de Hardware

Esta obra es la mejor guía visual y práctica sobre hardware del momento. En su interior encontraremos los consejos de los expertos sobre las nuevas tecnologías, las soluciones a los problemas más frecuentes, cómo hacer overclocking, modding, y muchos más trucos y secretos.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-029-0

Access

Este manual nos introduce de lleno en el mundo de Access para aprender a crear y administrar bases de datos de forma profesional. Todos los secretos de una de las principales aplicaciones de Office, explicados de forma didáctica y sencilla.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-025-2

Redes Cisco

Este libro permitirá al lector adquirir todos los conocimientos necesarios para planificar, instalar y administrar redes de computadoras. Todas las tecnologías y servicios Cisco, desarrollados de manera visual y práctica en una obra única.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-024-5

Proyectos con Office

Esta obra nos enseña a usar las principales herramientas de Office a través de proyectos didácticos y útiles. En cada capítulo encontraremos la mejor manera de llevar adelante todas las actividades del hogar, la escuela y el trabajo.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-023-8

Dreamweaver y Fireworks

Esta obra nos presenta las dos herramientas más poderosas para la creación de sitios web profesionales de la actualidad. A través de procedimientos paso a paso, nos muestra cómo armar un sitio real con Dreamweaver y Fireworks sin necesidad de conocimientos previos.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-022-1

Excel revelado

Este manual contiene una selección de más de 150 consultas de usuarios de Excel y todas las respuestas de Claudio Sánchez, un reconocido experto en la famosa planilla de cálculo. Todos los problemas encuentran su solución en esta obra imperdible.

→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-021-4

Robótica avanzada

Esta obra nos permitirá ingresar al fascinante mundo de la robótica. Desde el ensamblaje de las partes hasta su puesta en marcha, todo el proceso está expuesto de forma didáctica y sencilla para así crear nuestros propios robots avanzados.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-020-7

Windows 7

En este libro, encontraremos las claves y los secretos destinados a optimizar el uso de nuestra PC tanto en el trabajo como en el hogar. Aprenderemos a llevar adelante una instalación exitosa y a utilizar todas las nuevas herramientas que incluye esta versión.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-015-3

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.

De Windows a Linux

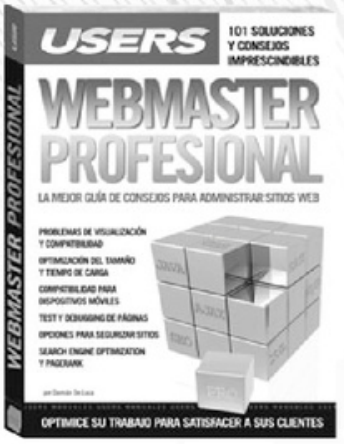
Esta obra nos introduce en el apasionante mundo del software libre a través de una completa guía de migración, que parte desde el sistema operativo más conocido: Windows. Aprenderemos cómo realizar gratuitamente aquellas tareas que antes hacíamos con software pago.

→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-013-9

Producción y edición de video

Un libro ideal para quienes deseen realizar producciones audiovisuales con bajo presupuesto. Tanto estudiantes como profesionales encontrarán cómo adquirir las habilidades necesarias para obtener una salida laboral con una creciente demanda en el mercado.

→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-012-2



Webmaster profesional

Esta obra explica cómo superar los problemas más frecuentes y complejos que enfrenta todo administrador de sitios web. Ideal para quienes necesiten conocer las tendencias actuales y las tecnologías en desarrollo que son materia obligada para dominar la Web 2.0.

→ COLECCIÓN: MANUALES USERS
→ 336 páginas / ISBN 978-987-663-011-5

Silverlight

Este manual nos introduce en un nuevo nivel en el desarrollo de aplicaciones interactivas a través de Silverlight, la opción multiplataforma de Microsoft. Quien consiga dominarlo creará aplicaciones visualmente impresionantes, acordes a los tiempos de la incipiente Web 3.0.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-010-8

Flash extremo

Este libro nos permitirá aprender a fondo Flash CS4 y ActionScript 3.0 para crear aplicaciones web y de escritorio. Una obra imperdible sobre uno de los recursos más empleados en la industria multimedia, que nos permitirá estar a la vanguardia del desarrollo.

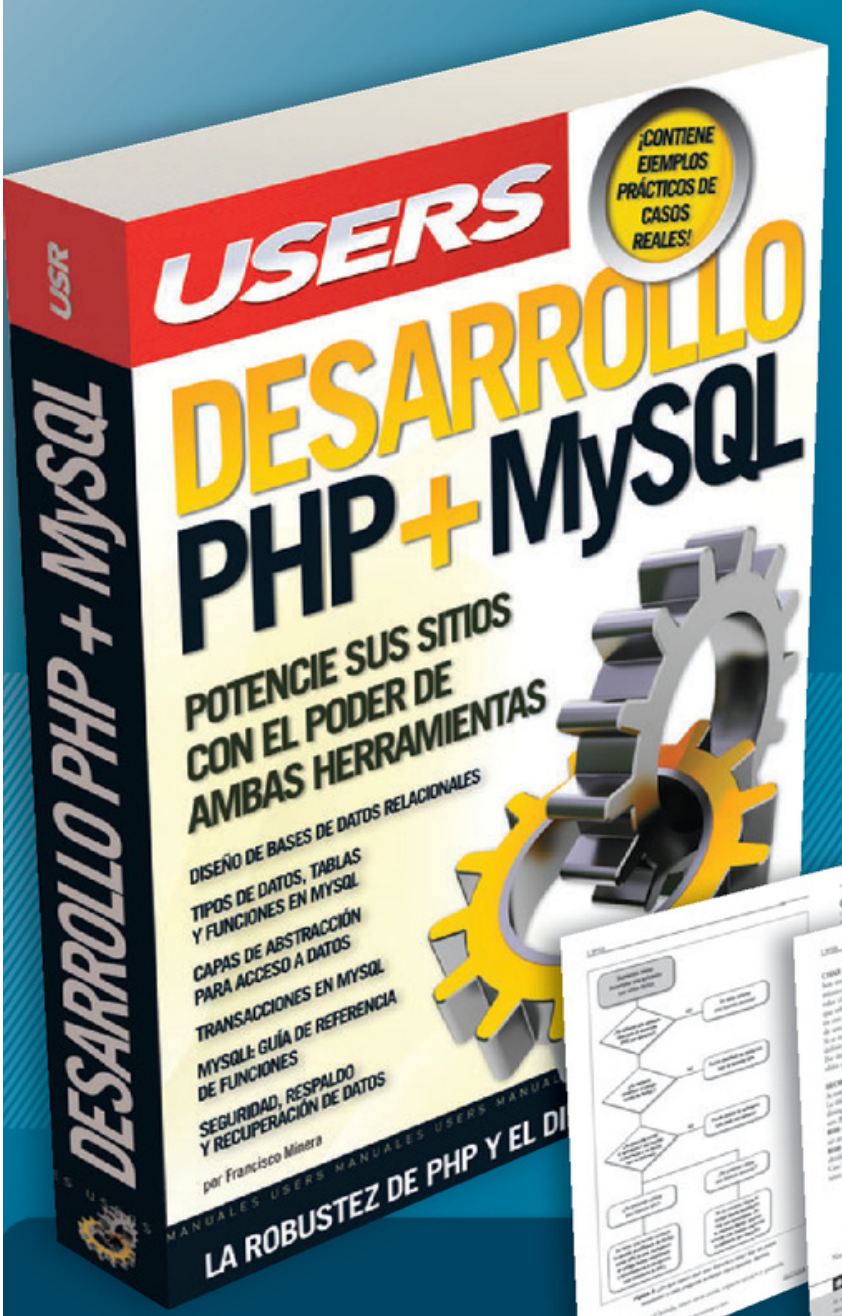
→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-663-009-2

Hackers al descubierto

Esta obra presenta un panorama de las principales técnicas y herramientas utilizadas por los hackers, y de los conceptos necesarios para entender su manera de pensar, prevenir sus ataques y estar preparados ante las amenazas más frecuentes.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-663-008-5

POTENCIE SUS SITIOS CON EL PODER DE AMBAS HERRAMIENTAS



Este libro presenta la fusión de dos de las herramientas más populares para el desarrollo de aplicaciones web de la actualidad: PHP y MySQL. En sus páginas, el autor nos enseñará las funciones del lenguaje, de modo de tener un acercamiento progresivo, y aplicar lo aprendido en nuestros propios desarrollos.

» DESARROLLO / PHP
» 432 PÁGINAS
» ISBN 978-987-1773-16-9

USERS

PROGRAMADOR .NET

Esta obra está dirigida a todos aquellos que quieran iniciarse en el desarrollo bajo lenguajes Microsoft, así como también a quienes provengan de otros lenguajes.
A través de la lectura de este manual, aprenderemos los conceptos de la programación orientada a objetos, con la tarea definitiva de crear aplicaciones para diferentes objetivos y plataformas. Además, comprenderemos la utilidad de las tecnologías .NET, su aplicación, cómo interactúan entre sí y de qué manera se desenvuelven con otras tecnologías existentes.
Todos los procedimientos son expuestos de manera práctica con el código fuente de ejemplo (disponible en Internet), diagramas conceptuales y la teoría necesaria para comprender en profundidad cada tema presentado. Al finalizar el libro, aprenderemos a desarrollar aplicaciones desde cero, tanto para escritorio como para la Web.
Matías Iacono, Ingeniero en Sistemas, docente y especialista certificado en tecnologías Microsoft, es el guía ideal para comenzar a desarrollar en uno de los lenguajes que tienen mayor salida laboral de la actualidad.

CONTENIDO

¡Capítulo Online!

NIVEL DE USUARIO

PRINCIPIANTE | INTERMEDIO | AVANZADO | EXPERTO

RedUSERS.com

En este sitio encontrará una gran variedad de recursos y software relacionado, que le servirán como complemento al contenido del libro. Además, tendrá la posibilidad de estar en contacto con los editores, y de participar del foro de lectores, en donde podrá intercambiar opiniones y experiencias.

Si desea más información sobre el libro puede comunicarse con nuestro Servicio de Atención al Lector: usershop@redusers.com

.NET DEVELOPER

This manual will help the reader to understand desktop and Web programming basics. In its pages, we will also learn new skills to become professional developers, such as lambda expressions, threading or LinQ, and WFC, among others.

ISBN 978-987-1773-26-8

MANUALES USERS MANUALES USERS MANUALES USERS MANUALES USERS MAN

MICROSOFT .NET FRAMEWORK 4.0 + SINTAXIS, PROCEDIMIENTOS Y FUNCIONES
C# PARA DESARROLLO MULTIPLATAFORMA + DESARROLLO CON XNA PARA WINDOWS PHONE 7
.NET AVANZADO: LINQ, PROGRAMACIÓN DE HILOS, GENÉRICOS