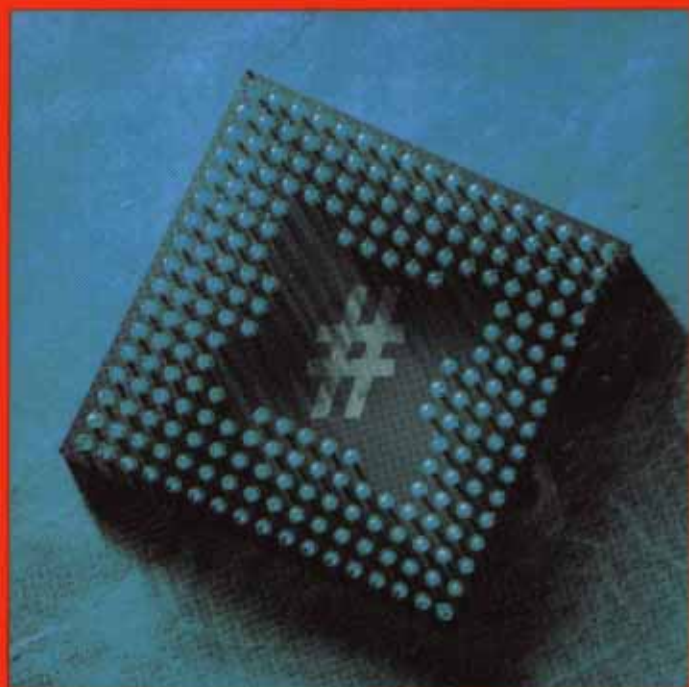


L A B I B L I A



Jeff Ferguson
Brian Patterson
Jason Beres

C#

ANAYA
MULTIMEDIA

La biblia de

C#

**Jeff Ferguson, Brian Patterson,
Jason Beres, Pierre Boutquin y
Meeta Gupta**

ANAYA
COMERCIAL

Todos los nombres propios de programas, sistemas operativos, equipos hardware, etc. que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Reservados todos los derechos. El contenido de esta obra está protegido por la ley, que establece penas de prisión y o multas, además de las correspondientes indemnizaciones por daños y perjuicios, para quienes reprodujeren, plagiaren, distribuyeren o comunicasen públicamente, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución artística fijada en cualquier tipo de soporte o comunicada a través de cualquier medio, sin la preceptiva autorización.

Copyright © 2003 by Anaya Multimedia.
Original English language edition copyright © 2002 by Hungry Minds, Inc.

All rights reserved including the right of reproduction in whole or in part in any form. This edition published by arrangement with the original publisher, Hungry Minds, Inc.

Edición española:

© EDICIONES ANAYA MULTIMEDIA (GRUPO ANAYA, S.A.), 2003
Juan Ignacio Luca de Tena, 15, 28027 Madrid
Depósito legal: M. 3.033 - 2003
ISBN: 84-415-1484-4
Printed in Spain
Imprime: Imprime Artes Gráficas Guemo, S.L.
Febrero, 32, 28022 Madrid.

Para mi familia y amigos.
Jeff Ferguson

Este libro está dedicado a mi tío, Brian Weston, al que no pareció importarle cuando fui de visita y pasé todo el día con su TRS-80 Model II.
Brian Patterson

A Nitin, que fue la motivación.
Meeta Gupta

Agradecimientos

Jeff Ferguson: Pocos libros de este tamaño y extensión son el fruto de un solo individuo y éste no es una excepción. Estoy en deuda con mucha gente por su ayuda y apoyo mientras escribía este libro. En primer lugar, debo dar las gracias a mis padres por la educación que recibí. Sin sus paternos consejos no me habría convertido en la persona que soy y no habría podido completar ninguno de mis trabajos. Siempre os estaré agradecido, no sólo a vosotros, sino a toda la familia por el amor y apoyo que siempre he recibido.

Me gustaría dar las gracias a todo el mundo de Wiley por su dirección en la elaboración de este material. Gracias, Andrea Boucher, Sharon Cox, Eric Newman y Chris Webb, por guiarme por el intimidador mundo de la publicación de libros técnicos. Gracias también a Rolf Crozier, que discutió conmigo este proyecto en primer lugar en sus primeros días. Debo dar las gracias especialmente a mi colega Bob Knutson, que revisó los borradores del material de este libro.

Gracias a Greg Frankenfield y a Paul Fridman por crear una excelente organización consultora basada en Microsoft que me permite trabajar en los proyectos de mis clientes junto en los míos. El crecimiento técnico que he experimentado durante mi estancia en Magenic ha sido incalculable. Esto es para que continúe el éxito de Magenic. Gracias a todo el mundo de las listas de correo y grupos de noticias de DOTNET en Internet. Estoy aprendiendo mucho sobre .NET Framework y C# simplemente leyendo vuestros correos. Los envíos de acá para allá del banter me han dado una mayor comprensión de cómo encajan todas estas nuevas piezas.

Brian Patterson: Me gustaría dar las gracias a mi esposa, Aimee, por perdonarme todas esas horas que pasé escondido en el ordenador para que pudiera completar este libro. Un agradecimiento especial a Steve Cisco por su duro trabajo en este libro, que abrió camino para el resto de nosotros; a Sharon Cox, la editora de adquisiciones, que siempre me mantuvo en el buen camino; al editor de proyecto, Eric Newman, por aguantar todos mis regates; y al editor de la serie, Michael Lane Thomas, que revisó todos y cada uno de los capítulos, haciendo algunas sugerencias muy buenas y proporcionando una apreciable comprensión de Microsoft y .NET Framework.

Pierre Boutquin: Se necesitó mucho trabajo para crear este libro y no sólo de la gente que aparece en la portada. Debo dar las gracias especialmente al equipo de Wiley por su tremendo esmero por producir un libro de calidad. Los revisores se merecen casi todo el crédito por hacerme parecer un escritor competente. Por último, este trabajo no habría sido posible sin el apoyo de mi familia y amigos: Sandra, Andrea, Jennifer y Paul, Tindy y Doel, Marcel y Diana Ban, Margaret Fekete, y John y Nadine Marshall.

Meeta Gupta: Agradezco a Anita que me diera la oportunidad. Pero mi mayor agradecimiento es para Nitin por, bueno, por todo.

Sobre los autores

Jeff Ferguson es consejero superior de Magenic Technologies, una compañía consultora de software dedicada a resolver problemas empresariales usando exclusivamente herramientas y tecnología de Microsoft. Ha sido programador de software profesional desde 1989 y ha desarrollado software para Unix, DOS y Windows empleando C, C++ y C#. Puede enviar un e-mail a Jeff en JeffF@magenic.com (no olvide incluir las tres "F" en el nombre de la dirección).

Brian Patterson actualmente trabaja para Affina, Inc., como jefe del equipo técnico, donde suele trabajar con C++ en HP-UX o en el desarrollo de Windows con cualquier versión de los lenguajes de Visual Studio. Brian ha estado escribiendo para varias publicaciones sobre Visual Basic desde 1994 y ha co-escrito varios libros relacionados con .NET, incluyendo *Migrating to Visual Basic .NET* y *.NET Enterprise Development with VB.NET*. Puede encontrarse generalmente contribuyendo en los grupos de noticias de MSDN o puede ponerse en contacto con él por e-mail en BrianDPatterson@msn.com.

Jason Beres ha sido programador de software durante 10 años. Actualmente es asesor en Florida del Sur y trabaja exclusivamente con tecnología de Microsoft. Jason tiene los certificados MCT, MCSD y MCDBA de Microsoft. Cuando no está enseñando, asesorando o escribiendo, está formateando su disco duro, instalando los últimos productos beta de Microsoft y poniéndose al día de los últimos episodios de "*Star Trek*".

Pierre Boutquin es arquitecto superior de software en la tesorería de uno de los principales bancos canadienses, donde ayuda a desarrollar software puntero para la prevención de riesgos de mercado. Tiene más de una década de experiencia introduciendo sistemas computerizados basados en el PC con un exhaustivo conocimiento del diseño de sistemas distribuidos, almacenamiento de datos, Visual Basic, Visual C++ y SQL. Ha co-escrito muchos libros sobre programación y ha contribuido con material sobre VB, COM+, XML y SQL a otros libros. Koshka y Sasha, sus dos adorables gatos de Birmania, ocupan casi todo el tiempo libre de Pierre. Mientras los acaricia, suele pensar en lo hermoso que sería encontrar más tiempo para volver al ajedrez o mantenerse informado sobre Bélgica, su país natal. Puede contactar con él en boutquin@hotmail.com.

Meeta Gupta tiene una licenciatura en ingeniería informática. Los sistemas de redes son lo que más le gusta. Actualmente trabaja en NIIT Ltd., donde diseña, desarrolla y escribe libros sobre temas muy diversos. Ha co-escrito libros sobre TCP/IP, A+ Certification, ASP.NET y PHP. También tiene una amplia experiencia diseñando y desarrollando varias ILT. Aparte de escribir, Meeta ha realizado cursos sobre C++, Sybase, Windows NT, Unix y HTML para una audiencia diversa, desde estudiantes hasta clientes corporativos.

NIIT es una compañía de soluciones globales TI que produce productos de enseñanza multimedia personalizados y tiene más de 2.000 centros de enseñanza por todo el mundo. NIIT tiene más de 4.000 empleados en 37 países y tiene acuerdos estratégicos con varias de las principales corporaciones, incluidos Microsoft y AT&T.

Sobre el editor de la serie

Michael Lane Thomas es un activo programador de comunidades y un analista de la industria informática que actualmente pasa la mayor parte de su tiempo difundiendo el evangelio de Microsoft .NET para Microsoft. Mientras trabajaba con más de media docena de editoriales, Michael ha escrito numerosos artículos técnicos y ha escrito o participado en casi 20 libros sobre numerosos temas técnicos, incluyendo Visual Basic, Visual C++ y tecnologías .NET. Es un prolífico defensor de la certificación de programas de Microsoft y ya ha conseguido su MCSD, MCSE+I, MCT, MCP+SB y MCDBA.

Además de sus escritos técnicos, también puede escuchar a Michael en las ondas de vez en cuando, incluidos dos programas de radio semanales en las cadenas Entercom (<http://www.entercom.com/>) y más a menudo en la ciudad de Kansas en News Radio 980KMBZ (<http://www.kmbz.com/>). También puede encontrarse con él en Internet haciendo un MSDN Webcast (<http://www.microsoft.com/usa/webcasts/>) debatiendo sobre .NET, la nueva generación de tecnologías aplicadas a la Red.

Michael empezó su trayectoria técnica en su época universitaria en la University of Kansas, donde ganó sus galones y un par de títulos. Tras un breve trabajo como técnico y asesor comercial para Global Online Japan, con base en Tokio, regresó a los Estados Unidos para ascender por la escalera corporativa. Ha ocupado puestos variados, incluyendo el de encargado de la IT, ingeniero de campo, instructor, consultor independiente e incluso un breve trabajo como CTO interino de una exitosa punto-com, aunque él cree que su actual papel como evangelista de .NET para Microsoft es el mejor del lote. Puede contactar con él vía e-mail en mlthomas@microsoft.com.

Índice

Agradecimientos	6
Sobre los autores	7
Sobre el editor de la serie	8
Introducción	29
Quién debería leer este libro	30
Cómo está organizado este libro	30
Parte I: Fundamentos del lenguaje C#	30
Parte II: Programación orientada a objetos con C#	31
Parte III: C# avanzado	31
Parte IV: Desarrollando soluciones .NET usando C#	31
Parte V: C# y .NET Framework	31
Parte VI: Apéndices	31
Cómo usar este libro	32
Normas usadas en este libro	32
Parte I. Fundamentos del lenguaje C#	35
1. Introducción a C#	37
.NET Framework	38
Desarrollo Web	38
Desarrollo de aplicaciones	39
Entorno común de ejecución	40
Bibliotecas de clase .NET	41

Lenguajes de programación .NET	42
Entorno ASP.NET	43
Historia de C, C++ y C#	43
Introducción a C#	45
Características del lenguaje	45
Clases	45
Tipos de datos	46
Funciones	47
Variables	47
Interfaces	48
Atributos	49
Cómo compilar C#	49
Lenguaje intermedio de Microsoft (MSIL)	49
Metadatos	51
Ensamblados	51
Resumen	52
2. Escribir su primer programa en C#	55
Cómo escoger un editor	55
La aplicación Hello World	56
Cómo construir una clase	56
El método Main()	57
Cómo escribir en la consola	57
Compilación y ejecución del programa	58
Las palabras clave y los identificadores	59
Uso de espacios en blanco	61
Cómo iniciar programas con la función Main()	62
Cómo comentar el código	64
Cómo usar comentarios de una línea	64
Usar comentarios regulares	64
Cómo generar documentación XML a partir de comentarios	65
<c>	67
<code>	68
<example>	68
<exception>	68
<list>	69
<param>	70
<paramref>	70
<permission>	71
<remarks>	71
<returns>	71
<see>	71
<seealso>	72

<summary>	73
<value>	73
Resumen	73
3. Trabajar con variables	75
Cómo dar nombre a sus variables	75
Asignación de un tipo a una variable	76
Cómo aplicar tamaño a sus variables	78
Cómo declarar sus variables	78
Uso de valores por defecto en las variables	79
Asignación de valores a variables	81
Uso de matrices de variables	81
Declaración de matrices unidimensionales	82
Cómo trabajar con los valores de las matrices unidimensionales	83
Inicialización de valores de elementos de matriz	84
Declaración de matrices multidimensionales	85
Uso de matrices rectangulares	85
Definición de matrices escalonadas	87
Tipos de valor y de referencia	88
Cómo convertir tipos de variable	89
Conversiones implícitas	89
Conversiones explícitas	90
Cómo trabajar con cadenas	92
Uso de caracteres especiales en cadenas	92
Desactivación de los caracteres especiales en cadenas	94
Cómo acceder a caracteres individuales en la cadena	95
Declaración de enumeraciones	95
Resumen	96
4. Expresiones	99
Cómo usar los operadores	99
Uso de expresiones primarias	100
Cómo usar los literales	100
Literales booleanos	101
Cómo usar los literales enteros en notaciones decimales y hexadecimales	101
Cómo usar los literales reales para valores de coma flotante	103
Cómo usar los literales de carácter para asignar valores de carácter	104
Cómo usar los literales de cadena para incrustar cadenas	104
Cómo usar los literales null	104
Uso de identificadores	105
Expresiones entre paréntesis	105
Cómo llamar a métodos con expresiones de acceso a miembros	106

Cómo llamar a métodos con expresiones de invocación	106
Cómo especificar elementos de matriz con expresiones de acceso a elementos	107
Cómo acceder a objetos con la palabra clave <code>this</code>	108
Cómo acceder a objetos con la palabra clave <code>base</code>	109
Cómo usar los operadores postfijo de incremento y de decremento	109
Creación de nuevos tipos de referencia con el operador <code>new</code>	110
Cómo devolver información sobre el tipo con el operador <code>typeof</code>	110
Cómo usar operadores <code>checked</code> y <code>unchecked</code>	110
Las expresiones unarias	113
Cómo devolver valores de operando con el operador unario más	113
Cómo devolver valores de operando con el operador unario menos	113
Expresiones negativas booleanas con el operador de negación lógica	113
El operador de complemento bit a bit	114
Cómo prefijar operadores de incremento y decremento	114
Los operadores aritméticos	115
Cómo asignar nuevos valores con el operador de asignación	116
Uso del operador multiplicación	116
Uso del operador división	117
Uso del operador resto	118
Uso del operador suma	118
Uso del operador resta	119
Los operadores de desplazamiento	120
Cómo mover bits con el operador de desplazamiento a la izquierda	120
Cómo mover bits con el operador de desplazamiento a la derecha	121
Cómo comparar expresiones con operadores relacionales	122
Cómo comprobar la igualdad con el operador de igualdad	122
Cómo comprobar la desigualdad con el operador de desigualdad	122
Cómo comprobar valores con el operador menor que	123
Cómo comprobar valores con el operador mayor que	123
Cómo comprobar valores con el operador menor o igual que	124
Cómo comprobar valores con el operador mayor o igual que	124
Operadores lógicos enteros	124
Cómo calcular valores booleanos con el operador AND	125
Cómo calcular valores booleanos con el operador exclusivo OR	125
Cómo calcular valores booleanos con el operador OR	126
Operadores condicionales lógicos	126
Comparación de valores booleanos con el operador AND condicional	126
Comparación de valores booleanos con el operador OR condicional	127
Comparación de valores booleanos con el operador lógico condicional	127
El orden de las operaciones	127
Resumen	129

5. Cómo controlar el flujo del código	131
Instrucciones de C#	131
Instrucciones para declarar variables locales	132
Cómo usar instrucciones de selección para seleccionar la ruta del código ..	133
La instrucción if	134
La instrucción switch	135
Cómo usar instrucciones de iteración para ejecutar instrucciones	
incrustadas	137
La instrucción while	138
La instrucción do	138
La instrucción for	139
La instrucción foreach	142
Instrucciones de salto para moverse por el código	142
La instrucción break	143
La instrucción continue	143
La instrucción goto	144
Cómo usar instrucciones para realizar cálculos matemáticos	
con seguridad	145
Resumen	145
6. Cómo trabajar con métodos.....	149
La estructura de un método	149
Tipo devuelto	150
Nombre del método	150
Lista de parámetros	150
Cuerpo del método	151
Cómo llamar a un método	151
Tipos de parámetros	155
Parámetros de entrada	155
Parámetros de salida	156
Parámetros de referencia	159
Matrices de parámetros	160
Sobrecarga de métodos	162
Métodos virtuales	163
Métodos sobrecargados	164
Resumen	166
7. Agrupación de datos usando estructuras.....	169
Cómo declarar una estructura	170
Cómo usar estructuras en el código	171
Cómo definir métodos en estructuras	173
Cómo usar métodos constructores	174
Cómo llamar a métodos desde estructuras	177

Cómo definir propiedades en estructuras	178
Cómo definir indizadores en estructuras	179
Cómo definir interfaces en estructuras	181
Cómo usar los tipos simples de C# como estructuras	182
Resumen	185
Parte II. Programación orientada a objetos con C#	187
8. Escribir código orientado a objetos	189
Clases y objetos	192
Terminología del diseño de software orientado a objetos	192
Abstracción	193
Tipos de datos abstractos	193
Encapsulación	195
Herencia	197
Herencia simple	197
Herencia múltiple	198
Polimorfismo	199
Resumen	202
9. Clases de C#	205
Cómo declarar una clase	205
El método Main	206
Cómo usar argumentos de línea de comandos	207
Cómo devolver valores	207
El cuerpo de la clase	209
Cómo usar constantes	209
Cómo usar campos	210
Cómo usar métodos	212
Cómo usar propiedades	212
Descriptores de acceso get	215
Descriptores de acceso set	216
Propiedades de sólo lectura y de sólo escritura	216
Cómo usar eventos	217
Cómo usar indizadores	217
Cómo usar operadores	222
Cómo usar constructores	222
Cómo usar destructores	223
Cómo usar los tipos de clase	226
Cómo usar la palabra clave this como identificador	226
El modificador static	228
Cómo usar campos estáticos	228
Cómo usar constantes estáticas	230

Cómo usar métodos estáticos	230
Resumen	232
10. Cómo sobrecargar operadores	237
Operadores unarios sobrecargables	238
Cómo sobrecargar el unario más	238
Cómo sobrecargar el unario menos	240
Cómo sobrecargar complementos bit a bit	242
Cómo sobrecargar el incremento prefijo	244
Cómo sobrecargar el decremento prefijo	245
Cómo sobrecargar los operadores true y false	246
Operadores binarios sobrecargables	248
Operadores de conversión sobrecargables	251
Operadores que no pueden sobrecargarse	253
Resumen	253
11. Herencia de clase	257
Cómo compilar con clases múltiples	258
Cómo especificar una clase base en C#	259
Ámbito	260
Cómo reutilizar identificadores de miembros en las clases derivadas	261
Cómo trabajar con métodos heredados	263
Métodos virtuales y de reemplazo	263
Polimorfismo	265
Métodos abstractos	267
Clases base: Cómo trabajar con propiedades e indizadores heredados	268
Cómo usar la palabra clave base	269
Cómo acceder a campos de clase base con la palabra clave base	270
Clases selladas	270
Contención y delegación	270
La clase de objeto .NET	277
Cómo usar boxing y unboxing para convertir a tipo object y desde el tipo object	279
Resumen	281
Parte III. C# avanzado	283
12. Cómo trabajar con espacios de nombre	285
Cómo declarar un espacio de nombre	286
Cómo declarar un espacio de nombre en varios archivos fuente	287
Cómo usar clases en un espacio de nombre	288
Cómo ayudar a los espacios de nombre mediante la palabra clave using	290
Cómo crear alias de nombres de clase con la palabra clave using	290

Cómo declarar directivas de espacio de nombre con la palabra clave using	293
Un rápido recorrido por los espacios de nombre de .NET	295
Resumen	298
13. Interfaces	301
Cómo definir una interfaz	303
Cómo definir métodos de interfaz	303
Cómo definir propiedades de interfaz	304
Cómo definir indizadores de interfaz	304
Cómo definir eventos de interfaz	305
Cómo derivar a partir de interfaces base	305
Cómo usar la palabra clave new para reutilizar identificadores	307
Cómo implementar interfaces en clases y estructuras	308
Cómo implementar métodos de interfaz con el mismo nombre	310
Cómo acceder a miembros de interfaz	311
Consultar a un objeto por una interfaz	311
Cómo acceder a una interfaz en un objeto	314
Declaraciones de interfaz y palabras clave de ámbito	316
Cómo implementar interfaces definidas por .NET Framework	317
Cómo implementar foreach mediante IEnumerable IEnumerator	317
Cómo implementar limpieza mediante IDisposable	322
Resumen	325
14. Enumeraciones	329
Cómo declarar una enumeración	331
Cómo usar una enumeración	333
Cómo usar operadores en valores de enumeración	335
Cómo usar la clase .NET System.Enum	337
Cómo recuperar nombres de enumeración	337
Cómo comparar valores de enumeración	339
Cómo descubrir el tipo subyacente en tiempo de ejecución	341
Cómo recuperar todos los valores de enumeración	341
Análisis de cadenas para recuperar valores de enumeración	342
Resumen	343
15. Eventos y delegados	345
Cómo definir delegados	346
Cómo definir eventos	346
Cómo instalar eventos	347
Cómo desencadenar eventos	348
Cómo unirlo todo	348
Cómo estandarizar un diseño de evento	350

Cómo usar descriptores de acceso de eventos	353
Cómo usar modificadores de eventos	354
Eventos estáticos	354
Eventos virtuales	355
Eventos de reemplazo	355
Eventos abstractos	355
Resumen	355
16. Control de excepciones	359
Cómo especificar el procesamiento de excepciones	361
Cómo capturar excepciones	362
Cómo usar la palabra clave try	362
Cómo atrapar clases específicas de excepciones	362
Cómo liberar recursos después de una excepción	364
La clase exception	365
Introducción a las excepciones definidas por .NET Framework	365
OutOfMemoryException	366
StackOverflowException	366
NullReferenceException	367
TypeInitializationException	368
InvalidCastException	368
ArrayTypeMismatchException	369
IndexOutOfRangeException	369
DivideByZeroException	370
OverflowException	370
Cómo trabajar con sus propias excepciones	371
Cómo definir sus propias excepciones	372
Cómo iniciar sus excepciones	373
Cómo usar excepciones en constructores y propiedades	374
Resumen	376
17. Cómo trabajar con atributos	379
Atributos	380
Cómo trabajar con atributos de .NET Framework	383
System.Diagnostics.ConditionalAttribute	384
System.SerializableAttribute class	386
System.ObsoleteAttribute class	388
Cómo escribir sus propias clases de atributo	390
Cómo restringir el uso de atributos	390
Cómo permitir múltiples valores de atributo	391
Cómo asignar parámetros de atributo	392
Ejemplo explicativo de las clases de atributo	394
Resumen	396

18. Cómo utilizar versiones en sus clases	399
El problema de las versiones	399
Cómo solucionar el problema de las versiones	402
Mediante el modificador new	402
Mediante el modificador override	404
Resumen	406
19. Cómo trabajar con código no seguro	409
Conceptos básicos de los punteros	410
Tipos de puntero	411
Cómo compilar código no seguro	412
Cómo especificar punteros en modo no seguro	413
Cómo acceder a los valores de los miembros mediante punteros	414
Cómo usar punteros para fijar variables a una dirección específica	415
Sintaxis del elemento de matriz puntero	416
Cómo comparar punteros	417
Cálculo con punteros	417
Cómo usar el operador sizeof	418
Cómo asignar espacio de la pila para la memoria	419
Resumen	419
20. Constructores avanzados de C#	423
Operadores implícitos y conversiones no válidas	424
Inicialización de estructuras	425
Cómo inicializar estructuras	426
Cómo resolver los problemas con la inicialización	427
Clases derivadas	429
Cómo pasar clases derivadas	429
Cómo resolver problemas que surgen cuando se pasan clases derivadas ..	430
Cómo usar no enteros como elementos de matriz	431
Resumen	434
Parte IV. Cómo desarrollar soluciones .NET usando C#	435
21. Cómo construir aplicaciones WindowsForms	437
Arquitectura de WindowsForms	438
La clase Form	438
La clase Application	438
Cómo crear la primera aplicación WindowsForms	439
Cómo compilar una aplicación WindowsForms	440
Ensamblados: cómo añadir información de versión a las aplicaciones	
WindowsForms	441
AssemblyTitle	442

AssemblyDescription	443
AssemblyConfiguration	443
AssemblyCompany	443
AssemblyProduct	443
AssemblyCopyright	444
AssemblyTrademark	444
AssemblyCulture	444
AssemblyVersion	445
El objeto Application con más detalle	447
Eventos Application	447
Cómo trabajar con eventos en el código	448
Propiedades Application	449
AllowQuit	450
CommonAppDataRegistry	450
CommonAppDataPath	450
CompanyName	451
CurrentCulture	451
CurrentInputLanguage	451
ExecutablePath	452
LocalUserAppDataPath	452
MessageLoop	452
ProductName	453
ProductVersion	453
SafeTopLevelCaptionFormat	453
StartupPath	453
UserAppDataPath	453
UserAppDataRegistry	454
Métodos Application	454
AddMessageFilter	454
DoEvents	457
Exit	457
ExitThread	457
OleRequired	457
OnThreadException	458
RemoveMessageFilter	460
Run	461
Cómo añadir controles al formulario	461
Jerarquía de las clases de control	462
Cómo trabajar con controles en un formulario	462
Cómo trabajar con recursos	465
Cómo trabajar con recursos de cadena	465
Cómo trabajar con recursos binarios	468
Resumen	468

22. Cómo crear aplicaciones Web con WebForms	471
Fundamentos de las aplicaciones ASP.NET Web	472
Nuevas características de ASP.NET	472
Ejecución en el entorno .NET Framework	472
Presentación de WebForms	472
Integración con Visual Studio .NET	473
Presentación de los controles de servidor	473
Controles de usuario y compuestos	474
Controles más usados en WebForms	474
Control Label	474
Control TextBox	475
Controles CheckBox y CheckBoxList	475
Controles RadioButton y RadioButtonList	476
Control ListBox	477
Control DropDownList	477
Control HyperLink	477
Controles Table, TableRow y TableCell	478
Control ImageButton	479
Controles Button y LinkButton	479
Cómo crear y configurar una aplicación Web	479
Cómo crear un nuevo proyecto	480
Cómo agregar controles al WebForm	483
Cómo controlar eventos	487
Viajes de ida y vuelta	487
Controladores de eventos	489
Cómo controlar la devolución de datos	491
Cómo usar el estado de vista	491
Resumen	492
23. Programación de bases de datos con ADO.NET	495
Clases Dataset y otras clases relacionadas	496
Compatibilidad con OLE DB SQL Server	497
Operaciones de bases de datos comunes mediante ADO.NET	499
Operaciones que no devuelven filas	500
Operaciones de datos que devuelven entidades de fila única	504
Operaciones de datos que afectan a las entidades de fila única	509
Operaciones de introducción de datos que afectan a las entidades de fila única	509
Operaciones de actualización que afectan a entidades de fila única	514
Operaciones de borrado que afectan a las entidades de fila única	515
Operaciones de datos que devuelven conjuntos de filas	517
Operaciones de datos que afectan a conjuntos de filas	520

Operaciones que no devuelven datos jerárquicos	522
Resumen	527
24. Cómo trabajar con archivos y con el registro de Windows	529
Cómo acceder a archivos	529
Acceso binario	530
BinaryWriter	530
BinaryReader	531
Cómo supervisar los cambios de archivo	533
Cómo usar la supervisión de archivos	534
Cómo codificar FileSystemWatcher	538
Cómo manipular archivos	539
Cómo copiar archivos	539
Cómo eliminar archivos	540
Cómo trasladar archivos	542
Cómo acceder al registro	544
Cómo leer claves del registro	544
Cómo escribir claves de registro	546
Cómo enumerar claves del registro	549
Resumen	551
25. Cómo acceder a secuencias de datos	555
Jerarquía de clases de E/S de datos	555
Cómo usar secuencias	556
Cómo usar escritores	557
Cómo usar lectores	557
Cómo trabajar con secuencias	558
E/S sincrónica	558
E/S asincrónica	563
Cómo leer de forma asincrónica	564
Cómo escribir de forma asincrónica	568
Escritores y lectores	570
Cómo escribir secuencias con BinaryWriter	570
Cómo leer de secuencias con BinaryReader	572
Cómo escribir XML con un formato correcto mediante la secuencia XmlWriter	573
Resumen	575
26. Cómo dibujar con GDI+	577
Cómo trabajar con gráficos	577
Cómo trabajar con Image en GDI+	586
Cómo trabajar con lápices y pinceles	591
Cómo usar la clase Pen	591

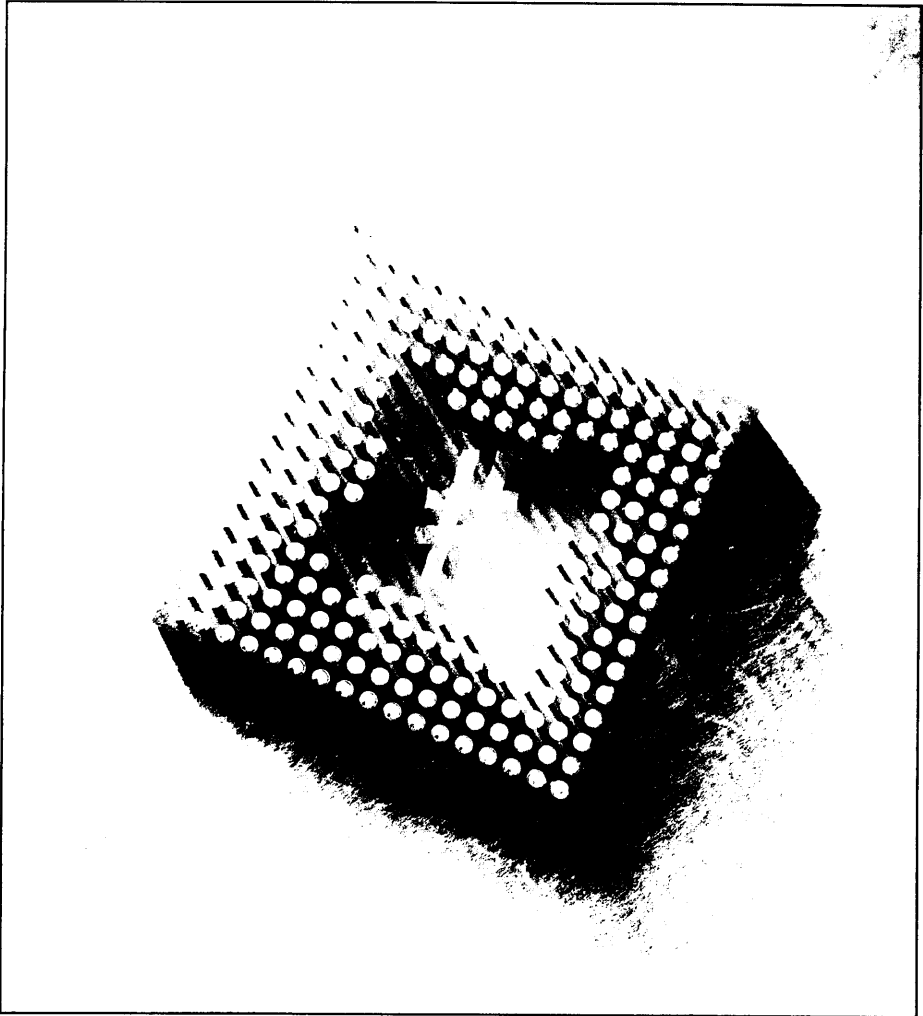
Cómo usar la clase Brush	593
Resumen	597
27. Cómo construir servicios Web	599
Funcionamiento de los servicios Web	600
Servicios Web y Visual Studio .NET	602
Lenguaje de descripción de servicio Web (WSDL)	605
Cómo usar el Protocolo de acceso simple a objetos (SOAP)	607
Cómo crear servicios Web con Visual Studio .NET	609
Cómo usar Visual Studio .NET para acceder a un servicio Web	612
Resumen	614
28. Cómo usar C# en ASP.NET	617
Cómo crear un servicio Web	618
Cómo crear una base de datos para un servicio Web	618
Conceptos del sistema de gestión de bases de datos relacionales	619
Tipos de datos de SQL Server	619
Cómo crear bases de datos y tablas	620
Cómo recuperar datos	621
Cómo insertar, actualizar y eliminar datos	621
Cómo usar procedimientos almacenados	621
Cómo crear la estructura de la base de datos	622
Cómo usar la plantilla Servicio Web ASP.NET	623
Cómo agregar controles de datos al servicio Web	624
Cómo codificar el servicio Web	627
Cómo crear un cliente de servicio Web	629
Cómo crear un nuevo proyecto de aplicación Web ASP.NET	630
Cómo agregar una referencia Web	631
Cómo implementar los métodos del servicio Web	632
Cómo implementar la aplicación	635
Implementación de proyectos en Visual Studio .NET	635
Cómo usar un proyecto de implementación para implementar una aplicación	635
Cómo implementar un proyecto usando la opción Copiar proyecto	637
Resumen	637
29. Cómo construir controles personalizados	641
Biblioteca de control de Windows	641
Propiedades	642
Métodos	644
Campos	645
Eventos	645
Aprender con un ejemplo	646

Cómo crear un temporizador de cuenta atrás	646
Cómo crear una prueba de carga Countdown	651
Cómo usar una biblioteca de clases	653
Cómo crear una clase para calcular el efecto de viento	653
Resumen	656
30. Cómo construir aplicaciones móviles.....	659
La red inalámbrica	659
Introducción al Mobile Internet Toolkit	660
Emuladores	660
Nokia	660
Pocket PC	660
Microsoft Mobile Explorer	661
Cómo crear un calculador de edades	661
Funciones de los dispositivos móviles	666
Funcionamiento de los controles móviles	667
Cómo usar el control Calendar	667
Cómo usar el control Image	668
Paginación en dispositivos móviles	670
Resumen	672
Parte V. C# y .NET Framework	673
31. Cómo trabajar con ensamblados.....	675
Ensamblados	675
Cómo encontrar ensamblados cargados	676
Nombres seguros de ensamblado	678
Cómo asignar la información de versión	680
Cómo asignar la información de referencia cultural	681
Cómo asignar la información de clave	682
Cómo trabajar con la clase Assembly	682
Cómo encontrar la información de ubicación del ensamblado	682
Cómo encontrar puntos de entrada del ensamblado	683
Cómo cargar ensamblados	684
Cómo trabajar con información de tipo de ensamblado	688
Cómo generar código nativo para ensamblados	689
Resumen	691
32. Reflexión	693
La clase Type	694
Cómo recuperar información de tipo	694
Cómo recuperar tipos mediante el nombre	694
Cómo recuperar tipos mediante instancias	695

Cómo recuperar tipos en un ensamblado	696
Cómo interrogar a objetos	697
Cómo generar código dinámico mediante la reflexión	700
Resumen	702
33. Subprocesamiento en C#	705
Subprocesamiento	705
Multitarea preferente	706
Prioridades de subproceso y bloqueo	707
Multiprocesamiento simétrico	707
Cómo usar los recursos: cuantos más, mejor	708
Dominios de aplicación	709
Ventajas de las aplicaciones de varios subprocesos	710
Aplicaciones con procesos largos	710
Aplicaciones de sondeo y escucha	710
Botón Cancelar	710
Cómo crear aplicaciones multiproceso	711
Cómo crear nuevos subprocesos	712
Prioridad de los subprocesos	715
Estado del subproceso	716
Cómo unir subprocesos	719
Cómo sincronizar subprocesos	721
Sondeo y escucha	722
Resumen	723
34. Cómo trabajar con COM	727
Introducción al Contenedor al que se puede llamar en tiempo de ejecución	728
Cómo crear ensamblados .NET a partir de componentes COM	729
Cómo usar la utilidad Tlbimp	729
Cómo crear un componente COM	731
Cómo usar el ensamblado de interoperabilidad desde C#	735
Cómo hacer referencia a la DLL COM desde C#	739
Cómo manejar errores de interoperabilidad	740
Cómo usar la invocación de plataforma	743
Resumen	744
35. Cómo trabajar con servicios COM+	747
El espacio de nombres System.EnterpriseServices	748
La clase ServicedComponent	752
Cómo registrar clases con COM+	754
Cómo usar atributos para clases COM+	756
ApplicationAccessControl	757
ApplicationActivation	757

ApplicationID	758
ApplicationName	758
ApplicationQueuing	758
AutoComplete	759
ComponentAccessControl	759
ConstructionEnabled	759
JustInTimeActivation	760
LoadBalancingSupported	760
SecurityRole	761
Cómo procesar transacciones	761
Propiedades ACID	762
Cómo escribir componentes de transacciones	763
Cómo acceder al contexto de objetos	765
Resumen	768
36. Cómo trabajar con los servicios remotos de .NET	771
Introducción al entorno remoto	771
Cómo crear un ensamblado de servidor remoto	773
Cómo crear un servidor remoto	775
Cómo especificar canales y puertos	777
Cómo especificar un formato de canal	778
Espacio de nombres System.Runtime.Remoting.Channels.Tcp	779
Espacio de nombres System.Runtime.Remoting.Channels.Http	779
Cómo activar el objeto remoto	781
Cómo registrar objetos con RegisterWellKnownServiceType	783
Cómo registrar objetos con el método Configure	785
Cómo escribir el cliente remoto	790
Resumen	794
37. C# y seguridad .NET	797
Seguridad de código	798
Directiva de seguridad de código	799
Permisos de código	799
Seguridad de usuario	800
Seguridad .NET y basada en funciones	802
Cómo asignar las funciones Windows	802
Principales	806
Permisos de acceso a código	806
Cómo crear una sencilla solicitud de código de permiso	807
Denegación de permisos	809
Cómo usar permisos basados en atributos	810
Directiva de seguridad	811
Niveles de directiva de seguridad	811

Grupos de código	812
Conjuntos de permisos con nombre	812
Cómo alterar directivas de seguridad	813
Resumen	814
Parte VI. Apéndices	817
Apéndice A. Manual de XML	819
Objetivos de diseño de XML	819
Objetivo 1: XML debe ser fácilmente utilizable en Internet	821
Objetivo 2: XML debe admitir una amplia variedad de aplicaciones	821
Objetivo 3: XML debe ser compatible con SGML	821
Objetivo 4: Debe ser sencillo escribir programas que procesen documentos XML	822
Objetivo 5: El número de características opcionales en XML debe mantenerse al mínimo, preferentemente a cero	822
Objetivo 6: Los documentos XML deben ser legibles para las personas y razonablemente claros	822
Objetivo 7: El diseño de XML debe ser preparado rápidamente	822
Objetivo 8: El diseño de XML debe ser formal y conciso	823
Objetivo 9: Los documentos XML deben ser fáciles de crear	823
Objetivo 10: La concisión del marcado XML es de mínima importancia	823
Breve lección de HTML	823
XML = HTML con etiquetas definidas por el usuario	827
Definiciones de tipo de documento	829
Esquemas XML	831
Espacios de nombre XML	834
Apéndice B. Contenido del CD-ROM	839
Índice alfabético	841



Introducción

La iniciativa .NET Framework de Microsoft supone el cambio más importante en la metodología del desarrollo de software para un sistema operativo de Microsoft desde la introducción de Windows. Este entorno está construido usando una arquitectura que permite a los lenguajes de software trabajar juntos, compartiendo recursos y código, para proporcionar a los programadores las avanzadas herramientas necesarias para construir la siguiente generación de aplicaciones de escritorio y de Internet. Visual Studio .NET de Microsoft incluye nuevas versiones de sus productos de compilador Visual Basic y C++ dirigidas al desarrollo de .NET, al igual que un lenguaje completamente nuevo llamado C#.

Este libro le mostrará cómo escribir código usando este novísimo lenguaje. Todos los términos de lenguaje tales como declaraciones, variables, bucles de control y clases, son tratados con detalle. Además, el libro le enseñará a usar C# para programar tareas con las que los programadores suelen enfrentarse en el mundo real. La última parte del libro explica cómo usar C# para desarrollar páginas Web, acceder a bases de datos, trabajar con objetos COM y COM+ heredados, desarrollar aplicaciones de escritorio para Windows, trabajar con varios conceptos de .NET Framework y mucho más.

El principal objetivo de este libro es el desarrollo .NET usando C# como el lenguaje de implementación y el compilador de línea de comandos C# de .NET Framework como la principal herramienta de desarrollo. El desarrollo de C#

empleando la herramienta Visual Studio .NET no se trata en este libro, aunque es algo que se puede dominar fácilmente una vez que se comprendan bien los fundamentos del desarrollo .NET usando C#.

Quién debería leer este libro

Este libro fue escrito teniendo en mente a los programadores novatos y los expertos. Si no conoce absolutamente nada sobre las bases del desarrollo de software, este libro le iniciará en sus fundamentos, mostrándole como funcionan las variables, los bucles de control y las clases. El libro también está dirigido a los programadores de cualquier nivel, mostrándoles las herramientas .NET disponibles para el desarrollo en C# y proporcionándoles trucos para hacer que sus propias aplicaciones en C# funcionen perfectamente dentro de las directrices de desarrollo de .NET Framework.

Si ya está introducido en el mundo de la creación de aplicaciones .NET, encontrará en este libro un recurso muy útil porque cubre casi todos los aspectos del desarrollo .NET exhaustivamente. Las primeras tres partes del libro sirven de punto de referencia ilustrativo para usar las características del lenguaje C#. En cambio, las dos últimas partes están dedicadas a mostrar C# como plataforma de desarrollo de aplicaciones, ilustrando el papel de C# en aplicaciones de escritorio, Web, bases de datos y basadas en componentes. En este libro se asume que es la primera vez que utiliza C# y pretende proporcionar una comprensión del lenguaje sin exigir un conocimiento previo. Sin embargo, el libro también supone que el lector está familiarizado con los entornos de aplicaciones usados en conjunción con sus aplicaciones C#.

Las últimas partes del libro abordan el uso de C# con aplicaciones de escritorio, Web, bases de datos y basadas en componentes, pero no explica esas plataformas con detalle. En su lugar, el libro supone que el lector tiene un conocimiento práctico de esas plataformas.

Cómo está organizado este libro

Este libro está organizado en seis partes:

Parte I: Fundamentos del lenguaje C#

Esta primera parte del libro proporciona una breve visión general de la familia de lenguajes de programación C y pasa a tratar los aspectos sintácticos básicos de C#. Variables, declaraciones, bucles de control de flujo y llamadas de método, todas son tratadas. Los programadores principiantes también encontrarán material explicativo sobre el uso de estos elementos sintácticos y aprenderán a elaborar código con estos conceptos.

Parte II: Programación orientada a objetos con C#

Los capítulos de esta segunda parte tratan de la noción de clase en C#. La clase es la unidad fundamental de código en una aplicación C# y comprender las clases es clave para construir una aplicación C# que funcione. Además esta parte se ocupa de temas como el diseño de clases, clases básicas, clases derivadas y sobrecarga de operadores.

Parte III: C# avanzado

La tercera parte del libro se concentra en rasgos de lenguaje específicos empleados por aplicaciones C# más avanzadas. Se abordan temas como el control de excepciones, la implementación de interfaces, los espacios de nombre, los atributos y el código no seguro, todos son tratados. El último capítulo de esta parte está dedicado a presentar algunos problemas de programación complicados y soluciones aplicadas usando C#.

Parte IV: Desarrollando soluciones .NET usando C#

La parte IV muestra cómo usar C# en aplicaciones que utilizan varias partes de .NET Framework. Esta parte del libro se separa de las otras secciones, que están dedicadas a la presentación de las características del lenguaje C#. La parte IV usa C# para construir aplicaciones usando varias plataformas de la aplicación .NET, desde formularios de Windows hasta Web Forms y aplicaciones ASP.NET y acceso a bases de datos. También echaremos un vistazo al trabajo con algunas tecnologías .NET avanzadas usando C#, incluyendo subprocesamientos, ensamblados y reflexión.

Parte V: C# y .NET Framework

La última parte del libro describe cómo se puede usar C# para trabajar con el propio .NET Framework. Se explican conceptos de Framework tales como ensamblados, reflexión, subprocesamiento e interoperabilidad de componentes COM/COM+. Cada capítulo explica el concepto de Framework apropiado y también enseña a aprovechar la tecnología usando C# como lenguaje de implementación.

Parte VI: Apéndices

La última sección del libro consta de dos apéndices: El primero ofrece una introducción al Lenguaje de marcado extensible (XML) y de qué manera los programadores pueden aprovechar este lenguaje para describir datos de una manera estandarizada. Muchos proyectos .NET usan XML de una forma u otra y varios archivos de configuración .NET están basados en la infraestructura XML. El segundo apéndice incluye una descripción del contenido del CD-ROM que acompaña al libro.

Cómo usar este libro

Los lectores que sean completamente novatos en el desarrollo de software (quizás los administradores Web) aprovecharán mejor este libro leyendo primero las partes I y II para conseguir una mejor comprensión de cómo funcionan los mecanismos de una aplicación de software. Puede ser importante que los nuevos programadores comprendan las bases del desarrollo de software y cómo encajan todas las piezas para construir una aplicación C# completa.

A los lectores que se acerquen a C# con un conocimiento previo de C++, el nuevo lenguaje les resultará muy familiar. C# fue construido pensando en C y C++ y la sintaxis se parece a la de estos lenguajes más antiguos. Estos lectores quizás deseen examinar las partes I y II para acostumbrarse a las variantes de sintaxis y luego quizás deseen lanzarse de lleno a la parte III para comprender las avanzadas características del lenguaje. Muchos de los aspectos de la parte III profundizan en los conceptos que distinguen C# de sus predecesores.

Los programadores que ya estén familiarizados con C# encontrarán bastante material útil. Las partes IV y V muestran el uso de C# en varias aplicaciones para la plataforma .NET y presentan varios ejemplos que explican el código C# que puede usarse para realizar tareas variadas. Estas dos últimas partes trasladan el libro del nivel teórico al nivel práctico y son ideales para los programadores de cualquier nivel que deseen comprender cómo puede usarse C# para implementar varias aplicaciones.

Normas usadas en este libro

A lo largo del libro encontrará unos rectángulos sombreados que resaltan la información especial o importante, estos son los siguientes:

ADVERTENCIA: Indica un procedimiento que, en teoría, podría causar dificultades o incluso la pérdida de datos; preste especial atención a los iconos de advertencia para evitar los errores de programación más comunes y los que no lo son tanto.

NOTA: Resalta la información interesante o adicional y suele contener pequeños trozos extra de información técnica sobre un tema.

TRUCO: Llamam la atención sobre hábiles sugerencias, pistas recomendables y consejos útiles.

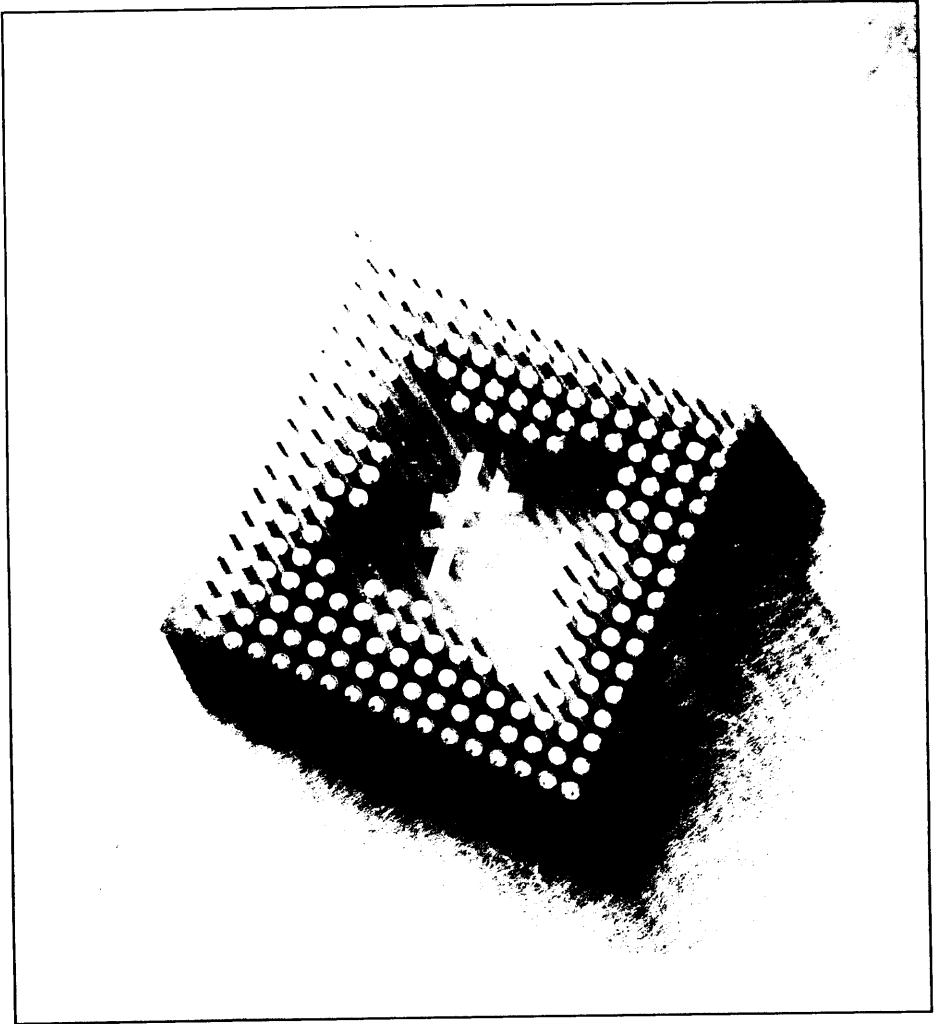
Además en este libro se usan las siguientes convenciones tipográficas:

- Los códigos de ejemplo aparecen en un tipo de letra `Courier`.
- Las opciones de menús se indican en orden jerárquico, con cada instrucción de menú separada por el signo "mayor que" y en un tipo de letra Arial. Por ejemplo, `Archivo>Abrir` quiere decir hacer clic en el comando `Archivo` en la barra de menú y luego seleccionar `Abrir`.
- Las combinaciones de teclas se indican de esta forma: **Control-C**.

Al final de cada capítulo encontrará un resumen de lo que debería haber aprendido al terminar de leer el capítulo.

Parte I

Fundamentos del lenguaje C#



1

Introducción a C#

Durante los últimos 20 años, C y C++ han sido los lenguajes elegidos para desarrollar aplicaciones comerciales y de negocios. Estos lenguajes proporcionan un altísimo grado de control al programador permitiéndole el uso de punteros y muchas funciones de bajo nivel. Sin embargo, cuando se comparan lenguajes, como Microsoft Visual Basic con C/C++, uno se da cuenta de que aunque C y C++ son lenguajes mucho más potentes, se necesita mucho más tiempo para desarrollar una aplicación con ellos. Muchos programadores de C/C++ han temido la idea de cambiar a lenguajes como Visual Basic porque podrían perder gran parte del control de bajo nivel al que estaban acostumbrados.

Lo que la comunidad de programadores necesitaba era un lenguaje que estuviera entre los dos. Un lenguaje que ayudara a desarrollar aplicaciones rápidas pero que también permitiese un gran control y un lenguaje que se integrase bien con el desarrollo de aplicaciones Web, XML y muchas de las tecnologías emergentes.

Facilitar la transición para los programadores de C/C++ existentes y proporcionar a la vez un lenguaje sencillo de aprender para los programadores inexpertos son sólo dos de las ventajas del nuevo lenguaje del barrio, C#. Microsoft presentó C# al público en la *Professional Developer's Conference* en Orlando, Florida, en el verano del 2000. C# combina las mejores ideas de lenguajes como C, C++ y Java con las mejoras de productividad de .NET Framework de Microsoft

y brinda una experiencia de codificación muy productiva tanto para los nuevos programadores como para los veteranos. Este capítulo profundiza en los cuatro componentes que constituyen la plataforma .NET además de analizar la compatibilidad para las tecnologías Web emergentes. A continuación, se analizan muchas de las funciones del lenguaje C# y se comparan con otros lenguajes populares.

.NET Framework

Microsoft diseñó C# desde su base para aprovechar el nuevo entorno .NET Framework. Como C# forma parte de este nuevo mundo .NET, deberá comprender perfectamente lo que proporciona .NET Framework y de qué manera aumenta su productividad.

.NET Framework se compone de cuatro partes, como se muestra en la figura 1.1: el entorno común de ejecución, un conjunto de bibliotecas de clases, un grupo de lenguajes de programación y el entorno ASP.NET. .NET Framework fue diseñado con tres objetivos en mente. Primero, debía lograr aplicaciones Windows mucho más estables, aunque también debía proporcionar una aplicación con un mayor grado de seguridad. En segundo lugar, debía simplificar el desarrollo de aplicaciones y servicios Web que no sólo funcionen en plataformas tradicionales, sino también en dispositivos móviles. Por último, el entorno fue diseñado para proporcionar un solo grupo de bibliotecas que pudieran trabajar con varios lenguajes. Las siguientes secciones analizan cada uno de los componentes de .NET Framework.

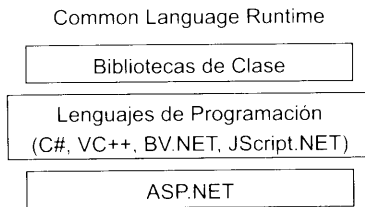


Figura 1.1. Los cuatro componentes de .NET Framework.

Desarrollo Web

.NET Framework fue diseñado con una idea en mente: potenciar el desarrollo de Internet. Este nuevo incentivo para el desarrollo de Internet se llama *servicios Web*. Puede pensar en los servicios Web como en una página Web que interactúa con programas en lugar de con gente. En lugar de enviar páginas Web, un servicio Web recibe una solicitud en formato XML, realiza una función en concreto y luego devuelve una respuesta al solicitante en forma de mensaje XML.

NOTA: XML o el Lenguaje de marcado extensible es un lenguaje autodescriptivo muy parecido a HTML. Por otra parte, XML no consta de etiquetas predefinidas, lo que le concede una gran flexibilidad para representar una amplia variedad de objetos.

Una típica aplicación para un servicio Web podría ser como capa situada en lo alto de un sistema de facturación de una empresa. Cuando un usuario que navega por la red compra los productos de una página Web, la información de la compra es enviada al servicio Web, que calcula el precio de todos los productos, añade una línea a la base de datos de existencias y devuelve una respuesta con una confirmación de pedido. Este servicio Web no sólo puede interactuar con páginas Web, puede interactuar con otros servicios Web, como un sistema de cuentas de pago de una empresa.

Para que el modelo de servicio Web sobreviva a la evolución natural de los lenguajes de programación, debe incluir muchas más cosas que un simple interfaz para la Web. El modelo de servicio Web también incluye protocolos que permiten que las aplicaciones encuentren servicios Web disponibles en una red interna o en Internet. Este protocolo también permite a la aplicación explorar el servicio Web y decidir cómo comunicarse con él y cómo intercambiar información. Para permitir el descubrimiento de servicios Web se estableció la Descripción, descubrimiento e integración universal (UDDI). Ésta permite que los servicios Web sean registrados y consultados, basándose en datos clave como el nombre de la compañía, el tipo de servicio y su localización geográfica.

Desarrollo de aplicaciones

Aparte del desarrollo Web, con .NET Framework también puede construir las tradicionales aplicaciones Windows. Estas aplicaciones creadas con .NET Framework se basan en Windows Forms. Windows Forms es una especie de cruce entre los formularios de Visual Basic 6 y los formularios de Visual C++. Aunque los formularios parecen iguales a sus predecesores, están completamente orientados a objetos y basados en clases, de forma muy parecida a los formularios objeto de Microsoft Foundation Class.

Estos nuevos Windows Forms ahora admiten muchos de los controles clásicos que aparecían en Visual Studio, como `Button`, `TextBox` y `Label`, junto a los controles `ActiveX`. Aparte de los controles tradicionales, también admite nuevos componentes como `PrintPreview`, `LinkLabel`, `ColorDialog` y `OpenFileDialog`.

La creación de aplicaciones con .NET también brinda muchas mejoras no disponibles en otros lenguajes, como la seguridad. Estas medidas de seguridad pueden determinar si una aplicación puede escribir o leer un archivo de disco. También permiten insertar firmas digitales en la aplicación para asegurarse de

que la aplicación fue escrita por una fuente de confianza. .NET Framework también permite incluir información de componentes, y de versión, dentro del código real. Esto hace posible que el software se instale cuando se lo pidan, automáticamente o sin la intervención del usuario. Juntas, todas estas funciones reducen los costes asistencia para la empresa.

Entorno común de ejecución

Los lenguajes de programación suelen componerse de un compilador y un entorno de ejecución. El compilador convierte el código que escribe en código ejecutable que puede ser ejecutado por los usuarios. El entorno de ejecución proporciona al código ejecutable un conjunto de servicios de sistema operativo. Estos servicios están integrados en una capa de ejecución de modo que el código no necesite preocuparse de los detalles de bajo nivel de funcionamiento con el sistema operativo. Operaciones como la gestión de memoria y la entrada y salida de archivos son buenos ejemplos de servicios realizados por un entorno de ejecución.

Antes de que se desarrollara .NET, cada lenguaje constaba de su propio entorno de ejecución. Visual Basic consta de un tiempo de ejecución llamado MSVBVM60.DLL. Visual C++ utiliza una DLL llamada MSVCRT.DLL. Cada uno de estos módulos de entorno de ejecución proporcionaba un conjunto de servicios de bajo nivel para codificar lo que los programadores escribían. Los programadores escribían código y luego lo compilaban con el apropiado tiempo de ejecución en mente. El código ejecutable incluiría su propio tiempo de ejecución, que puede ser instalado en el equipo del usuario si aún no estaba presente.

El principal problema que presentan estos entornos de ejecución es que estaban diseñados para usar un solo lenguaje. El tiempo de ejecución de Visual Basic proporcionaba algunas funciones estupendas para operaciones como trabajar con memoria e iniciar objetos COM, pero estas funciones estaban disponibles sólo para los usuarios de Visual Basic. Los programadores que usaban Visual C++ no podían usar las funciones del tiempo de ejecución de Visual Basic. Los usuarios de Visual C++ tenían su propio tiempo de ejecución, con su propia larga lista de funciones, pero esas funciones no estaban disponibles para los usuarios de Visual Basic. Este enfoque de "tiempos de ejecución separados" impedía que los lenguajes pudiesen funcionar conjuntamente sin problemas. No es posible, por ejemplo, tomar algo de memoria en un fragmento de código en Visual Basic y luego pasárselo a una parte de código en Visual C++, lo que liberaría la memoria. Los diferentes tiempos de ejecución implementan su propio conjunto de funciones a su manera. Los conjuntos de funciones de los diferentes tiempos de ejecución son inconsistentes. Incluso las funciones que se encuentran en más de un tiempo de ejecución se implementan de diferentes formas, haciendo imposible que dos fragmentos de código escritos en diferentes lenguajes trabajen juntos.

Uno de los objetivos de diseño de .NET Framework era unificar los motores de ejecución para que todos los programadores pudieran trabajar con un solo conjunto de servicios de ejecución. La solución de .NET Framework se llama Entorno común de ejecución (CLR). El CLR proporciona funciones como la gestión de memoria, la seguridad y un sólido sistema de control de errores, a cualquier lenguaje que se integre en .NET Framework. Gracias al CLR, todos los lenguajes .NET pueden usar varios servicios de ejecución sin que los programadores tengan que preocuparse de si su lenguaje particular admite una función de ejecución.

El CLR también permite a los lenguajes interactuar entre sí. La memoria puede asignarse mediante código escrito en un lenguaje (Visual Basic .NET, por ejemplo) y puede ser liberada con código escrito en otro (por ejemplo, C#). Del mismo modo, los errores pueden ser detectados en un lenguaje y procesados en otro.

Bibliotecas de clase .NET

A los programadores les gusta trabajar con código que ya ha sido probado y ha demostrado que funciona, como el API Win32 y la biblioteca de clase MFC. La reutilización del código lleva mucho tiempo siendo el objetivo de la comunidad de desarrolladores de software. Sin embargo, la posibilidad de reutilizar el código no ha estado a la altura de las expectativas.

Muchos lenguajes han tenido acceso a cuerpos de código previamente comprobados y listos para ser ejecutados. Visual C++ se ha beneficiado de las bibliotecas de clase, como las Clases de fundación Microsoft (MFC), que permitió a los programadores de C++ crear aplicaciones Windows rápidamente, y la Biblioteca activa de plantillas (ATL), que proporciona ayuda para crear objetos COM. No obstante, la naturaleza específica del lenguaje de estas bibliotecas las ha hecho inservibles para ser usadas en otros lenguajes. Los programadores de Visual Basic tienen vetado el uso de ATL para crear sus objetos COM.

.NET Framework proporciona muchas clases que ayudan a los programadores a reutilizar el código. Las bibliotecas de clase .NET contienen código para programar subprocesos, entrada y salida de archivos, compatibilidad para bases de datos, análisis XML y estructuras de datos, como pilas y colas. Y lo mejor de todo, toda esta biblioteca de clase está disponible para cualquier lenguaje de programación compatible con .NET Framework. Gracias al CLR, cualquier lenguaje .NET puede usar cualquier clase de la biblioteca .NET. Como ahora todos los lenguajes admiten los mismos tiempos de ejecución, pueden reutilizar cualquier clase que funcione con .NET Framework. Esto significa que cualquier funcionalidad disponible para un lenguaje también estará disponible para cualquier otro lenguaje .NET.

El cuadro de reutilización de bibliotecas de clases dibujado por .NET Framework se vuelve aún mejor cuando se da cuenta de que la reutilización se extiende a su código, no sólo al código que Microsoft lanza con .NET. El código

de Microsoft sólo es código que fue escrito usando un lenguaje que .NET admitía y se compilaba usando una herramienta de desarrollo .NET. Esto significa que Microsoft está usando las mismas herramientas que usará para escribir su código. Puede escribir código capaz de ser usado en otros lenguajes .NET, exactamente lo mismo que Microsoft con su biblioteca de clase .NET Framework permite escribir código en C#, por ejemplo, y enviárselo a programadores en Visual Basic .NET, que pueden usar ese código que compiló en sus aplicaciones. La figura 1.2 ilustra una visión muy general de las bibliotecas de clase .NET.

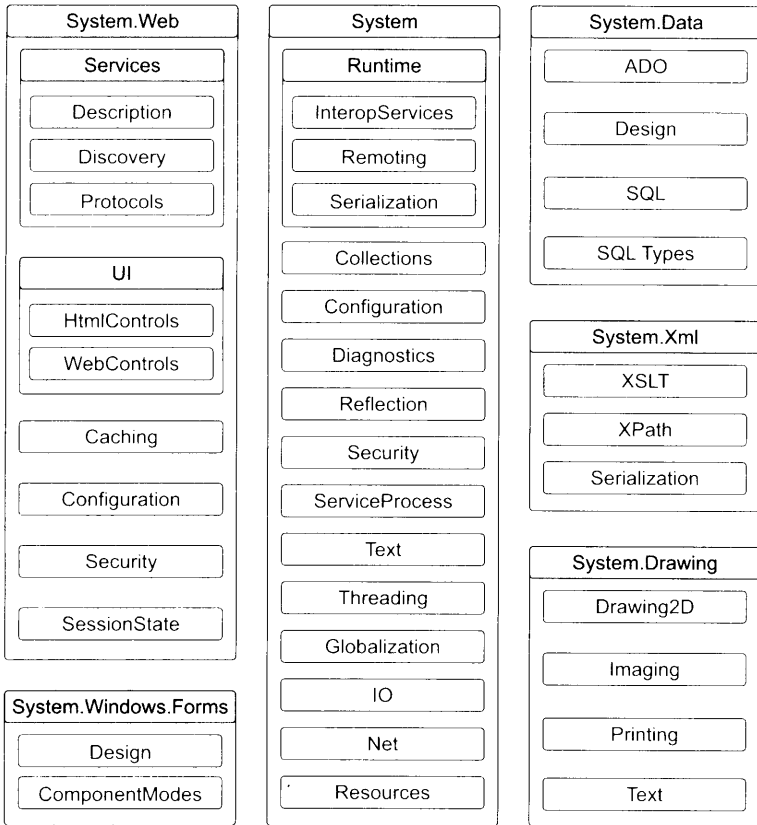


Figura 1.2. Las bibliotecas de clase .NET Framework

Lenguajes de programación .NET

.NET Framework proporciona un conjunto de herramientas que le ayudan a elaborar código que funciona con .NET Framework. Microsoft proporciona un conjunto de lenguajes que ya son "compatibles con .NET". C# es uno de estos lenguajes. También se han creado nuevas versiones de Visual Basic y Visual C++ para aprovechar las ventajas de .NET Framework y hay una versión de Jscript.NET en camino.

El desarrollo de lenguajes compatibles .NET no se limita a Microsoft. El grupo .NET de Microsoft ha publicado documentación que muestra cómo los proveedores pueden hacer que sus lenguajes funcionen con .NET, y estos proveedores están haciendo lenguajes como COBOL y Perl compatibles con .NET Framework. Actualmente, una tercera parte de los proveedores e instituciones están preparando más de 20 lenguajes capaces de integrarse en .NET Framework.

Entorno ASP.NET

Internet fue concebida en un principio para enviar contenido estático a los clientes Web. Estas páginas Web nunca cambiaban y eran las mismas para todos los usuarios que navegaban hasta esa localización. Microsoft lanzó servidores activos para permitir la creación de páginas dinámicas basadas en la aportación e interacción del usuario con una página Web. Esto se consiguió mediante el uso de secuencias de comandos que funcionaban por detrás de la página Web, generalmente escritas en VB Script. Cuando los usuarios visitaban una página Web, se les podía pedir que verificasen la información (manualmente o con una cookie), y luego la secuencia de comandos podía generar una página Web que le era devuelta al usuario.

ASP.NET mejora al original ASP proporcionando "código detrás". En ASP, HTML y las secuencias de comando se mezclaban en un documento. Con ASP.NET y su "código detrás", se puede separar el código y HTML. Ahora, cuando la lógica de una página Web necesite cambiar, no hace falta buscar por cientos o miles de líneas de HTML para localizar la secuencia de comandos que necesita modificarse.

De forma parecida a Windows Forms, ASP.NET admite Web Forms. Los Web Forms permiten arrastrar y colocar controles en sus formularios y codificarlos como haría en cualquier típica aplicación Windows.

Como ASP.NET usa .NET Framework, también usa el compilador Justo a tiempo (JIT). Las páginas ASP tradicionales se ejecutaban muy lentamente porque el código era interpretado. ASP.NET compila el código cuando es instalado en el servidor o la primera vez que es necesario, lo que aumenta enormemente la velocidad.

Historia de C, C++ y C#

El lenguaje de programación C# fue creado con el mismo espíritu que los lenguajes C y C++. Esto explica sus poderosas prestaciones y su fácil curva de aprendizaje. No se puede decir lo mismo de C y C++, pero como C# fue creado desde cero, Microsoft se tomó la libertad de eliminar algunas de las prestaciones más pesadas (cómo los punteros). Esta sección echa un vistazo a los lenguajes C y C++, siguiendo su evolución hasta C#.

El lenguaje de programación C fue diseñado en un principio para ser usado en el sistema operativo UNIX. C se usó para crear muchas aplicaciones UNIX, incluyendo un compilador de C, y a la larga se usó para escribir el mismo UNIX. Su amplia aceptación en el mundo académico se amplió al mundo comercial y los proveedores de software como Microsoft y Borland publicaron compiladores C para los ordenadores personales. El API original para Windows fue diseñado para trabajar con código Windows escrito en C y el último conjunto de API básicos del sistema operativo Windows sigue siendo compatible con C hoy en día.

Desde el punto de vista del diseño, C carecía de un detalle que ya ofrecían otros lenguajes como Smalltalk: el concepto de objeto. Piense en un objeto como en una colección de datos y un conjunto de operaciones que pueden ser realizadas sobre esos datos. La codificación con objetos se puede lograr usando C, pero la noción de objeto no era obligatoria para el lenguaje. Si quería estructurar su código para que simulara un objeto, perfecto. Si no, perfecto también. En realidad a C no le importaba. Los objetos no eran una parte fundamental del lenguaje, por lo que mucha gente no prestó mucha atención a este estándar de programación.

Una vez que el concepto de orientación a objetos empezó a ganar aceptación, se hizo evidente que C necesitaba ser depurado para adoptar este nuevo modo de considerar al código. C++ fue creado para encarnar esta depuración. Fue diseñado para ser compatible con el anterior C (de manera que todos los programas escritos en C pudieran ser también programas C++ y pudieran ser compilados con un compilador de C++). La principal aportación a C++ fue la compatibilidad para el nuevo concepto de objeto. C++ incorporó compatibilidad para clases (que son "plantillas" de objetos) y permitió que toda una generación de programadores de C pensaran en términos de objetos y su comportamiento.

El lenguaje C++ es una mejora de C, pero aún así presenta algunas desventajas. C y C++ pueden ser difíciles de manejar. A diferencia de lenguajes fáciles de usar como Visual Basic, C y C++ son lenguajes de muy "bajo nivel" y exigen que mucho código para funcionar correctamente. Tiene que escribir su propio código para manejar aspectos como la gestión de memoria y el control de errores. C y C++ pueden dar como resultado aplicaciones muy potentes, pero debe asegurarse de que el código funciona bien. Un error en la escritura del programa puede hacer que toda la aplicación falle o se comporte de forma inesperada. Como el objetivo al diseñar C++ era retener la compatibilidad con el anterior C, C++ fue incapaz de escapar de la naturaleza de bajo nivel de C.

Microsoft diseñó C# de modo que retuviera casi toda la sintaxis de C y C++. Los programadores que estén familiarizados con esos lenguajes pueden escoger el código C# y empezar a programar de forma relativamente rápida. Sin embargo, la gran ventaja de C# consiste en que sus diseñadores decidieron no hacerlo compatible con los anteriores C y C++. Aunque esto puede parecer un mal asunto, en realidad es una buena noticia. C# elimina las cosas que hacían que fuese difícil trabajar con C y C++. Como todo el código C es también código C++, C++ tenía que mantener todas las rarezas y deficiencias de C. C# parte de cero y sin ningún

requisito de compatibilidad, así que puede mantener los puntos fuertes de sus predecesores y descartar las debilidades que complicaban las cosas a los programadores de C y C++.

Introducción a C#

C#, el nuevo lenguaje presentado en .NET Framework, procede de C++. Sin embargo, C# es un lenguaje orientado a objetos (desde el principio), moderno y seguro.

Características del lenguaje

Las siguientes secciones hacen un rápido repaso a algunas de las características de C#. Si no está familiarizado con alguno de estos conceptos, no se preocupe. Todos serán tratados con detalle en capítulos posteriores.

Clases

Todo el código y los datos en C# deben ser incluidos en una clase. No puede definir una variable fuera de una clase y no puede escribir ningún código que no esté en una clase. Las clases pueden tener *constructores*, que se ejecutan cuando se crea un objeto de la clase, y un destructor, que se ejecuta cuando un objeto de la clase es destruido. Las clases admiten herencias simples y todas las clases derivan al final de una clase base llamada objeto. C# admite técnicas de versiones para ayudar a que sus clases evolucionen con el tiempo mientras mantienen la compatibilidad con código que use versiones anteriores de sus clases.

Por ejemplo, observe la clase llamada Family. Esta clase contiene los dos campos estáticos que incluyen el nombre y el apellido de un miembro de la familia junto con un método que devuelve el nombre completo del miembro de la familia.

```
class Class1
{
    public string FirstName;
    public string LastName;
    public string FullName()
    {
        return FirstName + LastName;
    }
}
```

NOTA: La herencia simple significa que una clase de C# sólo se puede derivar de una clase base.

C# le permite agrupar sus clases en una colección de clases llamada *espacio de nombre*. Los espacios de nombre tienen nombres y pueden servir de ayuda para organizar colecciones de clases en agrupaciones lógicas.

A medida que empieza a aprender C#, descubrirá que todos los espacios de nombre relevantes para .NET Framework empiezan con el término *System*. Microsoft también ha decidido incluir algunas clases que ayudan a la compatibilidad con versiones anteriores y al acceso a los API. Estas clases se incluyen en el espacio de nombre `Microsoft`.

Tipos de datos

C# permite trabajar con dos tipos de datos: de valor y de referencia. Los de valor contienen valores reales. Los de referencia contienen referencias a valores almacenados en algún lugar de la memoria. Los tipos primitivos como `char`, `int` y `float`, junto con los valores y estructuras comentados, son tipos de valor. Los tipos de referencia tienen variables que tratan con objetos y matrices. C# viene con tipos de referencia predefinidos (`object` y `string`), junto con tipos de valor predefinidos (`sbyte`, `short`, `int`, `long`, `byte`, `ushort`, `uint`, `ulong`, `float`, `double`, `bool`, `char` y `decimal`). También puede definir en el código sus propios tipos de valor y referencia. Todos los tipos de valor y de referencia derivan en última instancia de un tipo base llamado `object`.

C# le permite convertir un valor de un tipo en un valor de otro tipo. Puede trabajar con conversiones implícitas y explícitas. Las conversiones implícitas siempre funcionan y nunca pierden información (por ejemplo, puede convertir un `int` en un `long` sin perder ningún dato porque un `long` es mayor que un `int`). Las conversiones explícitas pueden producir pérdidas de datos (por ejemplo, convertir un `long` en un `int` puede producir pérdida de datos porque un `long` puede contener valores mayores que un `int`). Debe escribir un operador `cast` en el código para que se produzca una conversión explícita.

En C# puede trabajar con matrices unidimensionales y multidimensionales. Las matrices multidimensionales pueden ser rectangulares, en las que cada una de las matrices tiene las mismas dimensiones, o escalonadas, en las que cada una de las matrices puede tener diferentes dimensiones.

Las clases y las estructuras pueden tener miembros de datos llamados *propiedades* y *campos*. Los campos son variables que están asociadas a la clase o estructura a la que pertenecen. Por ejemplo, puede definir una estructura llamada `Empleado` que tenga un campo llamado `Nombre`. Si define una variable de tipo `Empleado` llamada `EmpleadoActual`, puede recuperar el nombre del empleado escribiendo `EmpleadoActual.Nombre`. Las propiedades son como los campos, pero permiten escribir código que especifique lo que debería ocurrir cuando el código acceda al valor. Si el nombre del empleado debe leerse de una base de datos, por ejemplo, puede escribir código que diga "cuando alguien pregunte el valor de la propiedad `Nombre`, lee el nombre de la base de datos y devuelve el nombre como una cadena".

Funciones

Una función es un fragmento de código que puede ser invocado y que puede o no devolver un valor al código que lo invocó en un principio. Un ejemplo de una función podría ser la función `FullName` mostrada anteriormente en este capítulo, en la clase `Family`. Una función suele asociarse a fragmentos de código que devuelven información, mientras que un método no suele devolver información. Sin embargo, para nuestros propósitos, generalizamos y nos referimos a las dos como funciones.

Las funciones pueden tener cuatro tipos de parámetros:

- Parámetros de entrada: tienen valores que son enviados a la función, pero la función no puede cambiar esos valores.
- Parámetros de salida: no tienen valor cuando son enviados a la función, pero la función puede darles un valor y enviar el valor de vuelta al invocador.
- Parámetros de referencia: introducen una referencia en otro valor. Tienen un valor de entrada para la función y ese valor puede ser cambiado dentro de la función.
- Parámetros Params: definen un número variable de argumentos en una lista.

C# y el CLR trabajan juntos para brindar gestión de memoria automática. No necesita escribir código que diga "asigna suficiente memoria para un número entero" o "libera la memoria que está usando este objeto". El CLR monitoriza el uso de memoria y recupera automáticamente más cuando la necesita. También libera memoria automáticamente cuando detecta que ya no está siendo usada (esto también se conoce como recolección de objetos no utilizados).

C# proporciona varios operadores que le permiten escribir expresiones matemáticas y de bits. Muchos (pero no todos) de estos operadores pueden ser redefinidos, permitiéndole cambiar la forma en que trabajan estos operadores.

C# admite una larga lista de expresiones que le permiten definir varias rutas de ejecución dentro del código. Las instrucciones de flujo de control que usan palabras clave como `if`, `switch`, `while`, `for`, `break` y `continue` permiten al código bifurcarse por caminos diferentes, dependiendo de los valores de sus variables. Las clases pueden contener código y datos. Cada miembro de una clase tiene algo llamado *ámbito de accesibilidad*, que define la visibilidad del miembro con respecto a otros objetos. C# admite los ámbitos de accesibilidad `public`, `protected`, `internal`, `protected internal` y `private`.

Variables

Las variables pueden ser definidas como constantes. Las constantes tienen valores que no pueden cambiar durante la ejecución del código. Por ejemplo, el

valor de pi es una buena muestra de una constante porque el valor no cambia a medida que el código se ejecuta. Las declaraciones de *tipo de enumeración* especifican un nombre de tipo para un grupo de constantes relacionadas. Por ejemplo, puede definir una enumeración de planetas con valores de Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano, Neptuno y Plutón, y usar estos nombres en el código. Usando los nombres de enumeraciones en el código hace que sea más fácil leerlo que si usara un número para representar a cada planeta.

C# incorpora un mecanismo para definir y procesar eventos. Si escribe una clase que realiza una operación muy larga, quizás quiera invocar un evento cuando la operación se complete. Los clientes pueden suscribirse a ese evento e incluirlo en el código, lo que permite que se les pueda avisar cuando haya acabado su operación. El mecanismo de control de eventos en C# usa delegados, que son variables que se refieren a una función.

NOTA: Un controlador de eventos es un procedimiento en el código que determina las acciones que deben llevarse a cabo cuando ocurra un evento, como que el usuario pulse un botón.

Si la clase contiene un conjunto de valores, los clientes quizás quieran acceder a los valores como si la clase fuera una matriz. Puede conseguirlo escribiendo un fragmento de código conocido como indexador. Suponga que escribe una clase llamada *ArcoIris*, por ejemplo, que contenga el conjunto de los colores del arco iris. Los visitantes querrán escribir `MiArcoIris[0]` para recuperar el primer color del arco iris. Puede escribir un indexador en la clase *ArcoIris* para definir lo que se debe devolver cuando el visitante acceda a esa clase, como si fuera una matriz de valores.

Interfaces

C# admite interfaces, que son grupos de propiedades, métodos y eventos que especifican un conjunto de funcionalidad. Las clases C# pueden implementar interfaces, que informan a los usuarios de que la clase admite el conjunto de funcionalidades documentado por la interfaz. Puede desarrollar implementaciones de interfaces sin interferir con ningún código existente. Una vez que la interfaz ha sido publicada, no se puede modificar, pero puede evolucionar mediante herencia. Las clases C# pueden implementar muchas interfaces, aunque las clases sólo pueden derivarse de una clase base.

Veamos un ejemplo de la vida real que puede beneficiarse del uso de interfaces para ilustrar su papel extremadamente positivo en C#. Muchas de las aplicaciones disponibles hoy en día admiten módulos complementarios. Supongamos que tiene un editor de código para escribir aplicaciones. Este editor, al ejecutarse, puede cargar módulos complementarios. Para ello, el módulo complementario debe seguir unas cuantas reglas. El módulo complementario de DLL debe expor-

tar una función llamada `CEEntry` y el nombre de la DLL debe empezar por `CEd`. Cuando ejecutamos nuestro editor de código, éste busca en su directorio de trabajo todas las DLL que empiecen por `CEd`. Cuando encuentra una, la abre y a continuación utiliza `GetProcAddress` para localizar la función `CEEntry` dentro de la DLL, verificando así que ha seguido todas las reglas exigidas para crear un módulo complementario. Este método de creación y apertura de módulos complementarios es muy pesado porque sobrecarga al editor de código con más tareas de verificación de las necesarias. Si usáramos una interfaz en este caso, la DLL del módulo complementario podría haber implementado una interfaz, garantizando así que todos los métodos necesarios, propiedades y eventos estén presentes en la propia DLL y que funciona como especifica la documentación.

Atributos

Los atributos aportan información adicional sobre su clase al CLR. Antes, si quería que la clase fuera autodescriptiva, tenía que seguir un enfoque sin conexión, en el que la documentación fuera almacenada en archivos externos como un archivo IDL o incluso archivos HTML. Los atributos solucionan este problema permitiendo al programador vincular información a las clases (cualquier tipo de información). Por ejemplo, puede usar un atributo para insertar información de documentación en una clase, explicando cómo debe actuar al ser usada. Las posibilidades son infinitas y ésa es la razón por la que Microsoft incluye tantos atributos predefinidos en .NET Framework.

Cómo compilar C#

Ejecutar el código C# con el compilador de C# produce dos importantes conjuntos de información: el código y los metadatos. Las siguientes secciones describen estos dos elementos y luego terminan examinando el componente esencial del código .NET: el ensamblado.

Lenguaje intermedio de Microsoft (MSIL)

El código generado por el compilador de C# está escrito en un lenguaje llamado Lenguaje intermedio de Microsoft, o MSIL. MSIL se compone de un conjunto específico de instrucciones que especifican cómo debe ser ejecutado el código. Contiene instrucciones para operaciones como la inicialización de variables, los métodos de llamada a objetos y el control de errores, por citar sólo unos pocos. C# no es el único lenguaje cuyo código fuente se transforma en MSIL durante el proceso de compilación. Todos los lenguajes compatibles con .NET, incluido Visual Basic .NET y C++ gestionado, producen MSIL cuando se compila su código fuente. Como todos los lenguajes .NET se compilan en el mismo conjunto de instrucciones MSIL, y como todos los lenguajes .NET usan el mismo tiempo de ejecución, los códigos de diferentes lenguajes y de diferentes compiladores pueden funcionar juntos fácilmente.

MSIL no es un conjunto de instrucciones específicas para una CPU física. MSIL no sabe nada de la CPU de su equipo y su equipo no conoce nada de MSIL. Entonces, ¿cómo se ejecuta el código .NET, si su CPU no lo puede interpretar? La respuesta es que el código MSIL se convierte en código específico de CPU cuando se ejecuta por primera vez. Este proceso se llama compilación "Justo a tiempo" o JIT. El trabajo de un compilador JIT es convertir el código genérico MSIL en código que su equipo pueda ejecutar en su CPU.

Quizás se esté preguntando por lo que parece ser un paso innecesario en el proceso. ¿Por qué generar MSIL cuando un compilador podría generar directamente código específico para la CPU?

Después de todo, los compiladores siempre lo han hecho en el pasado. Hay un par de razones que lo explican. En primer lugar, MSIL permite que el código compilado se transfiera fácilmente a hardware diferente. Suponga que ha escrito algo de código C# y le gustaría ejecutarlo en su ordenador personal y en su dispositivo portátil. Es muy probable que estos dos equipos tengan diferentes CPU. Si sólo tiene un compilador de C# para una CPU específica, entonces necesita dos compiladores de C#: uno para la CPU de su ordenador personal y otro para la CPU de su dispositivo portátil. Tendría que compilar el código dos veces, asegurándose de poner el código adecuado para cada equipo. Con MSIL, sólo se compila una vez.

Al instalar .NET Framework en su ordenador personal se incluye un compilador JIT que convierte el MSIL en código específico para la CPU de su ordenador personal. Al instalar .NET Framework en su ordenador portátil se incluye un compilador JIT que convierte el mismo MSIL en código específico para la CPU de su dispositivo portátil. Ahora tiene un solo código base MSIL que puede ejecutarse en cualquier equipo que tenga un compilador JIT .NET. El compilador JIT en ese equipo se ocupa de hacer que el código se ejecute.

Otra razón para que el compilador use MSIL es que el conjunto de instrucciones puede leerse fácilmente por un proceso de verificación. Parte del trabajo del compilador JIT es verificar el código para asegurarse de que resulte lo más claro posible. El proceso de verificación asegura que el código accede a la memoria correctamente y de que está usando los tipos de variable correctos al llamar a métodos que esperan un tipo específico. Estos procesos de verificación se aseguran de que el código no ejecute ninguna instrucción que origine un fallo. El conjunto de instrucciones MSIL fue diseñado para hacer este proceso de verificación relativamente sencillo. Los conjuntos de instrucciones específicos para cada CPU están optimizados para la rápida ejecución del código, pero producen código que puede ser difícil de leer y, por tanto, de verificar. Tener un compilador de C# que produce directamente código específico para CPU puede hacer difícil la verificación del código, o incluso imposible. Al permitir al compilador JIT de .NET Framework que verifique el código se asegura de que el código acceda a la memoria en un modo libre de fallos y que los tipos de variable sean usados correctamente.

Metadatos

El proceso de compilación también produce metadatos, lo que es una parte importante de la historia de cómo se comparte el código .NET. Tanto si usa C# para construir una aplicación para un usuario final como si lo usa para construir una biblioteca de clases que será usada por la aplicación de alguna otra persona, querrá usar código .NET ya compilado. Ese código puede ser proporcionado por Microsoft como parte de .NET Framework, o por un usuario a través de Internet. La clave para usar este código externo es hacer que el compilador de C# sepa qué clases y variables están en el otro código base para que pueda comparar el código fuente que ha escrito con el código que aparece en el código base precompilado con el que está trabajando.

Piense en los metadatos como en "tablas de contenidos" para el código compilado. El compilador de C# coloca metadatos en el código compilado junto al MSIL generado. Este metadato describe con exactitud todas las clases que escriba y cómo están estructuradas. Todos los métodos y variables de las clases están completamente descritos en los metadatos, listos para ser leídos por otras aplicaciones. Visual Basic .NET, por ejemplo, puede leer los metadatos para que una biblioteca .NET proporcione la función IntelliSense de listar todos los métodos disponibles para una clase en concreto.

Si alguna vez ha trabajado con COM (Modelo de objetos componentes), quizás esté familiarizado con las *bibliotecas de tipos*. Las bibliotecas de tipos tratan de proporcionar una funcionalidad "de tabla de contenidos" similar para objetos COM. Sin embargo, las bibliotecas de tipos presentaban algunas limitaciones, una de las cuales consistía en que no se incluían todos los datos importantes relacionados con el objeto. Los metadatos de .NET no presentan este inconveniente. Toda la información necesaria para describir una clase en un código está situada en el metadato. Los metadatos tienen todas las ventajas de las bibliotecas de tipos COM, pero sin sus limitaciones.

Ensamblados

A veces usará C# para construir una aplicación para un usuario final. Estas aplicaciones se presentan como archivos ejecutables con extensión .EXE. Windows siempre ha trabajado con archivos .EXE como programas de aplicación y C# admite a la perfección la construcción de archivos .EXE.

Sin embargo, puede haber ocasiones en las que no quiera construir una aplicación completa. Quizás quiera construir en su lugar una biblioteca de código que pueda ser usada por otras personas. Quizás también quiera construir algunas clases de utilidad, por ejemplo, y luego transferir el código a un programador de Visual Basic .NET, que usará sus clases en una aplicación de Visual Basic .NET. En casos como éste, no construirá una aplicación, sino un ensamblado.

Un ensamblado es un paquete de código y metadatos. Cuando utiliza un conjunto de clases en un ensamblado, está usando las clases como una unidad y estas clases comparten el mismo nivel de control de versión, información de seguridad

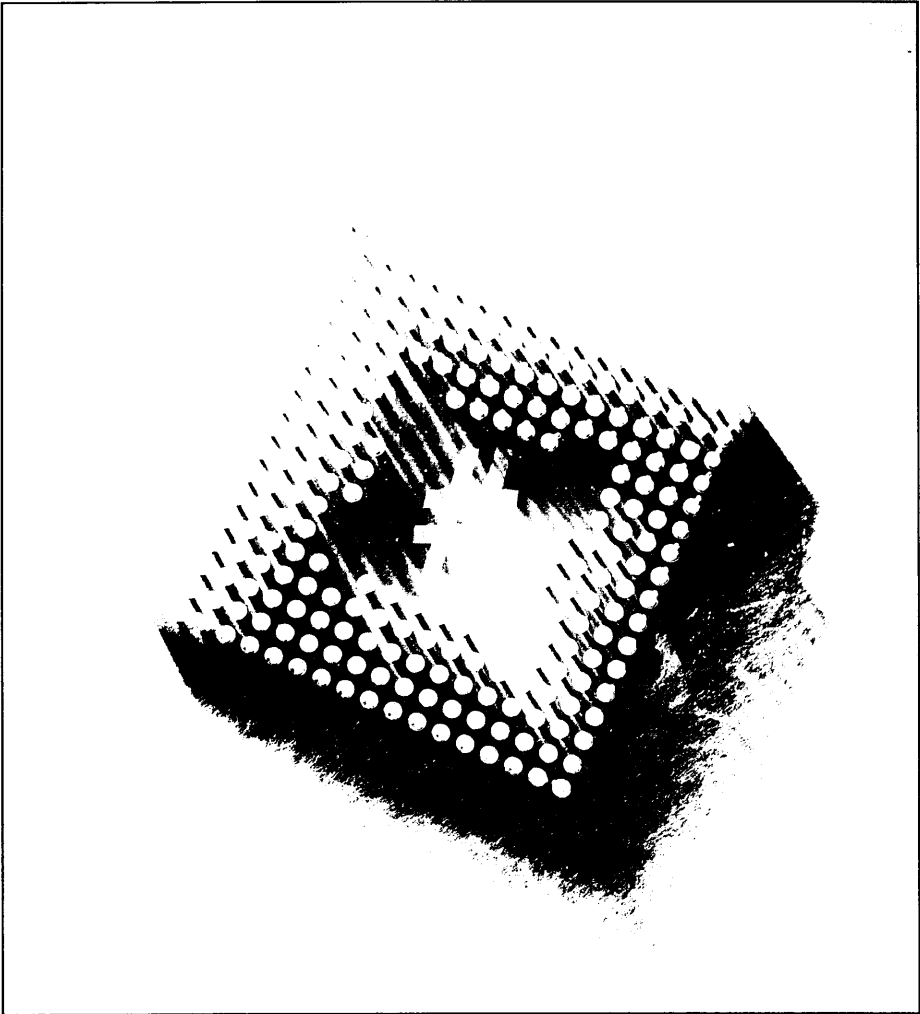
y requisitos de activación. Imagine un ensamblado como una "DLL lógica". Si no está familiarizado con Microsoft Transaction Server o COM+, puede imaginar un ensamblado como el equivalente .NET de un paquete.

Hay dos tipos de ensamblados: ensamblados privados y ensamblados globales. Al construir el ensamblado, no necesita especificar si quiere construir un ensamblado privado o global. La diferencia es ostensible cuando se implementa el ensamblado. Con un ensamblado privado, hace que el código esté disponible para una sola aplicación. El ensamblado se empaqueta como una DLL y se instala en el mismo directorio que la aplicación que lo está usando. Con el uso de un ensamblado privado, la única aplicación que puede usar el código es el ejecutable situado en el mismo directorio que el ensamblado.

Si quiere compartir el código con varias aplicaciones, quizás quiera considerar el uso del código como ensamblado global. Los ensamblados globales pueden ser usados por cualquier aplicación .NET en el sistema, sin importar el directorio en el que esté instalada. Microsoft incorpora ensamblados en .NET Framework y cada uno de los ensamblados de Microsoft se instala como ensamblado global. .NET Framework contiene una lista de ensamblados globales en un servicio llamado *caché de ensamblado global* y el SDK de .NET Microsoft Framework incluye utilidades para instalar y eliminar ensamblados de la caché de ensamblado global.

Resumen

En este capítulo se han explicado las bases de .NET Framework. Tras seguir la evolución desde C a C++ y hasta C#, se han examinado los puntos fuertes de la lista de prestaciones de C#. También se ha investigado el producto del compilador de C#, el código MSIL y los metadatos, y se ha revisado el uso de ensamblados como los bloques de construcción esenciales del código .NET compilado.



2 Escribir su primer programa en C#

Este capítulo le guía a través del desarrollo de una sencilla aplicación de C#. También aprenderá cómo están estructuradas las aplicaciones C# más sencillas y cómo invocar al compilador de C# para convertir el código fuente en código que puede ser ejecutado por .NET Framework. Finalmente, aprenderá a documentar el código empleando comentarios de código fuente y cómo convertir automáticamente sus comentarios en un documento XML.

Cómo escoger un editor

A la hora de escribir código para .NET Framework en C# tiene varias opciones. La elección más lógica es usar Visual Studio .NET. Usando Visual Studio, dispone de todas las ventajas de la tecnología IntelliSense, el marcado de sintaxis y muchas otras herramientas que aumentan la productividad.

Muchos editores de terceros intentan aunar en un paquete las herramientas de producción de Visual Studio. Algunas de estas herramientas pueden ser descargadas como shareware y otras son de libre distribución. Los ejemplos de este capítulo usan sencillamente el Bloc de notas de Windows. Al usar el Bloc de notas, no sólo demostramos que se puede usar cualquier editor de texto para escribir aplica-

ciones en C#, sino que también servirá para aprender las bases necesarias para compilar aplicaciones.

Además, el uso del Bloc de notas servirá para demostrarle que no necesita confiar en ningún asistente para generar el código. Puede simplemente concentrarse en el lenguaje mismo, sin tener que aprender los detalles de un IDE. Sin embargo, tenga en cuenta que para las aplicaciones más grandes quizás prefiera usar un editor que muestre los números de línea, lo que puede ser muy útil cuando se está buscando código defectuoso.

La aplicación Hello World

El código que se muestra en el listado 2.1 es una aplicación de C# completa. Se ejecuta desde una ventana de la consola y presenta el mensaje *Hello World!* en la pantalla. Las siguientes secciones siguen este código línea a línea.

Listado 2.1. Cómo escribir en la consola

```
class HelloWorld
{
    public static void Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

Cómo construir una clase

La primera línea del programa C# define una clase. Una clase es un recipiente para todo el código de la aplicación.

A diferencia de C y C++, todo el código debe estar contenido en una clase, con escasas excepciones. Estas excepciones a la regla son la instrucción `using`, las declaraciones de estructuras y la declaración `namespace`. Cualquier intento de escribir código que no esté contenido en una clase da como resultado un error de compilación.

La primera línea de la aplicación *Hello World* empieza con la palabra clave `class` y, a continuación, la palabra *HelloWorld*. *HelloWorld* es el nombre de la clase que el código está creando. Cada clase debe tener asignado un nombre único para que luego pueda referirse a ellas.

Inmediatamente después de la declaración de clase se debe abrir una llave. La llave de apertura se usa para abrir el cuerpo de la clase del código. Todo el código que escriba en la clase debe incluirse después de esta llave de apertura. Además de la llave de apertura, también debe haber una llave de cierre, como la que aparece en la última línea de la aplicación *HelloWorld*. Asegúrese de que toda su programación esté entre estas dos llaves.

El método Main()

Todas las aplicaciones escritas en C# deben constar de un método llamado `Main()`. Un método es un conjunto de instrucciones que realizan una acción. Este método puede devolver información a la sección de código que lo llamó pero en determinadas circunstancias no es necesario que lo haga.

NOTA: Los términos método y función suelen usarse de forma indistinta, pero hay una diferencia. Un método es una función contenida en una clase. Una función suele ser un grupo de instrucciones que no está contenido en una clase y que suele estar en un lenguaje, como C o C++. Como en C# no se puede añadir código fuera de una clase, nunca tendrá una función.

La palabra clave `public` en la declaración del método `Main()` también contiene la palabra `public`, que informa al compilador de que el método `Main()` debe ser públicamente accesible. El método `Main()` no sólo es accesible por otros métodos desde dentro de la aplicación, sino también externamente por otras aplicaciones. Al declarar el método `Main()` como público, está creando un punto de entrada para que Windows inicie la aplicación cuando un usuario lo desee.

Cuando un usuario haga doble clic sobre el icono de la aplicación *HelloWorld*, Windows explorará el ejecutable en busca de un punto de entrada con ese nombre. Si no encuentra una entrada, la aplicación no podrá ejecutarse.

La palabra `Static` en la declaración del método significa que el compilador sólo debe permitir que exista en memoria una copia del método por vez. Como el método `Main()` es el punto de entrada a la aplicación, sería catastrófico permitir que el punto de entrada se abriese más de una vez ya que permitiría más de una copia de la aplicación en memoria e, indudablemente, algunos errores graves.

Justo antes de la palabra `Main`, verá la palabra `Void`. `Void` es lo que la función principal devuelve cuando ha completado su ejecución. Significa que la aplicación no devuelve ningún valor después de haberse completado. Esta aplicación de ejemplo no es muy avanzada, así que no necesita devolver ningún valor. Sin embargo, en circunstancias normales, la función `Main()` devolvería un valor entero reemplazando la palabra `void` por `int`. Valores de devolución válidos son cualquier tipo de dato simple definido en .NET Framework.

De forma muy parecida a una declaración de clase, cualquier método que defina debe también contener una llave de apertura y otra de cierre entre las que se debe colocar todo el código del método. Puede ver las llaves de apertura y de cierre para el método `Main()` en las líneas 4 y 6 en el listado 2.1.

Cómo escribir en la consola

La línea 5 del listado 2.1 contiene una llamada al método `WriteLine`. Este método está contenido en .NET Framework y escribe una cadena en la consola. Si

se ejecuta desde una ventana de la consola, el texto debería aparecer en la pantalla. Si ejecuta este comando desde un entorno de Visual Studio, cualquier resultado que produzca aparecerá en la ventana de salida.

Ya hemos aprendido que todas las funciones de C# deben ser definidas dentro de una clase. Las funciones de .NET Framework no son una excepción. La función `WriteLine()` se encuentra en una clase llamada `Console`. La palabra clave `Console`, usada justo antes de la llamada a la función `WriteLine()`, indica al compilador que la función `WriteLine()` se encuentra en una clase llamada `Console`. La clase `Console` es una de las muchas clases de .NET Framework y la función `WriteLine()` es un miembro de la clase `Console`. El nombre de la clase está separado del nombre de la función que se invoca por medio de un punto.

El nombre `System` aparece inmediatamente antes del nombre de clase `Console`. Las clases de .NET Framework están organizadas en grupos llamados espacios de nombre. Los espacios de nombre se explican con más detalle en un capítulo posterior. Por ahora, piense en los nombres de espacios como en una colección de clase. La clase `Console` se encuentra en un espacio de nombre de .NET Framework llamado `System` y debe escribir el nombre de este espacio de nombre en el código. El compilador de C# necesita encontrar el código de `WriteLine()` para que la aplicación se ejecute correctamente y debe dar al compilador suficiente información sobre los espacios de nombre y las clases antes de encontrar el código de `WriteLine()`.

El texto que se incluye dentro de los paréntesis de `WriteLine()` es una cadena. Una cadena en C# es una colección de caracteres encerrados entre comillas y guardados juntos como unidad. Al colocar la cadena entre los paréntesis se indica al compilador que queremos pasar la cadena como parámetro de la función `WriteLine()`. La función `WriteLine()` escribe una cadena en la consola y el parámetro de la función indica a `WriteLine()` qué cadena debe escribirse.

La línea 5 incluye una gran cantidad de información que puede interpretarse de la siguiente forma: "Compilador C#, quiero llamar a `WriteLine()` con el parámetro de cadena `'Hello World!'`" La función `WriteLine()` se incluye en una clase llamada `Console` y la clase `Console` se incluye en un espacio de nombre llamado `System`. La línea 5 termina con un punto y coma. Todas las instrucciones deben terminar con un punto y coma. El punto y la coma separan una instrucción de otra en C#.

Compilación y ejecución del programa

Ahora que ya ha revisado el código del listado 2.1, es hora de ejecutarlo. Escriba el código del listado 2.1 en su editor de texto favorito y guárdelo como un archivo llamado `listado_2.1.cs`. La extensión `cs` es la extensión de todos los archivos que contienen código C#.

NOTA: Antes de compilar el ejemplo en C#, debe asegurarse de que el compilador de C# esté en su Path. La aplicación `csc.exe` generalmente está en la carpeta `C:\Windows\Microsoft.NET\Framework\v1.0.xxxx` (reemplace `V1.0.Xxxx` con su versión de .NET Framework), lo que puede comprobar buscándola en Windows. Para añadir entradas a su ruta, busque en la Ayuda de Windows la palabra `Path`.

A continuación abra un símbolo de comandos y dirijase a la carpeta en la que guardó el archivo `HelloWorld.cs`. Una vez allí, puede escribir el siguiente comando:

```
csc HelloWorld.cs
```

El comando `csc` invoca al compilador C# de NET Framework. Al ejecutar este comando se genera un ejecutable llamado `HelloWorld.exe`, que puede ser ejecutado exactamente igual que cualquier aplicación de Windows. Si ejecutamos este archivo, se escribirá texto en la ventana de la consola tal y como se muestra en la figura 2.1.

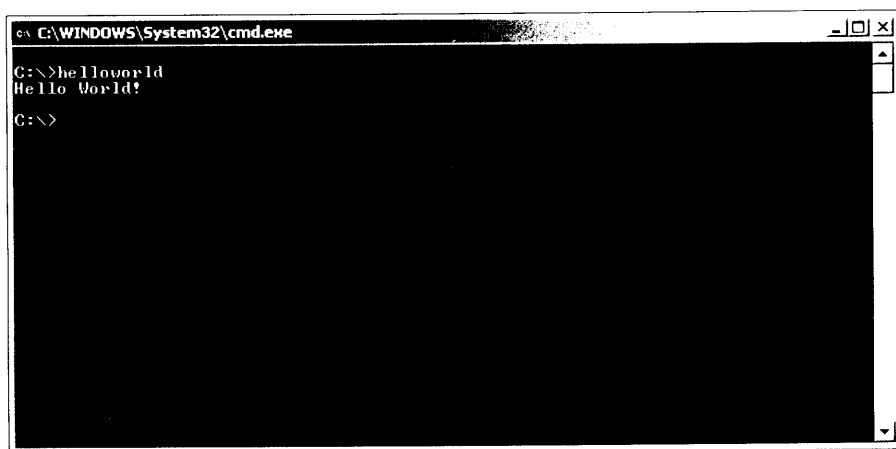


Figura 2.1. La ventana emergente de comando muestra la aplicación Hello World en acción.

¡Enhorabuena! Acaba de escribir su primera aplicación de C#.

Las palabras clave y los identificadores

La aplicación de C# del listado 2.1 contiene muchas palabras, separadas por espacios. En ella, se utilizan dos tipos de nombres: palabras clave e identificadores. Esta sección describe las diferencias entre estos dos tipos de nombres.

Las palabras clave son palabras que tienen un significado especial en el lenguaje C#. Estas palabras han sido reservadas en C# y nos referimos a ellas como

palabras reservadas. Las palabras `class`, `static` y `void` son las palabras reservadas del listado 2.1. Cada palabra clave posee en el lenguaje C# un significado especial. La siguiente lista contiene todas las palabras clave definidas en C#.

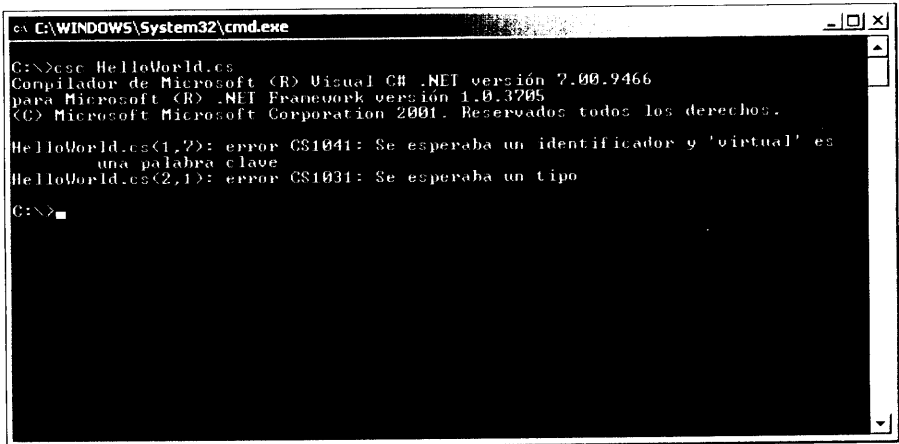
<code>abstract</code>	<code>enum</code>	<code>long</code>	<code>stackalloc</code>
<code>as</code>	<code>event</code>	<code>namespace</code>	<code>static</code>
<code>base</code>	<code>explicit</code>	<code>new</code>	<code>string</code>
<code>bool</code>	<code>extern</code>	<code>null</code>	<code>struct</code>
<code>break</code>	<code>false</code>	<code>object</code>	<code>switch</code>
<code>byte</code>	<code>finally</code>	<code>operator</code>	<code>this</code>
<code>case</code>	<code>fixed</code>	<code>out</code>	<code>throw</code>
<code>catch</code>	<code>float</code>	<code>override</code>	<code>true</code>
<code>char</code>	<code>for</code>	<code>params</code>	<code>try</code>
<code>checked</code>	<code>foreach</code>	<code>private</code>	<code>typeof</code>
<code>class</code>	<code>goto</code>	<code>protected</code>	<code>uint</code>
<code>const</code>	<code>if</code>	<code>public</code>	<code>ulong</code>
<code>continue</code>	<code>implicit</code>	<code>readonly</code>	<code>unchecked</code>
<code>decimal</code>	<code>in</code>	<code>ref</code>	<code>unsafe</code>
<code>default</code>	<code>int</code>	<code>return</code>	<code>ushort</code>
<code>delegate</code>	<code>interface</code>	<code>sbyte</code>	<code>using</code>
<code>do</code>	<code>internal</code>	<code>sealed</code>	<code>virtual</code>
<code>double</code>	<code>is</code>	<code>short</code>	<code>void</code>
<code>else</code>	<code>lock</code>	<code>sizeof</code>	<code>while</code>

Los identificadores son los nombres que se usan en las aplicaciones. C# no reserva nombres de identificadores. Los identificadores son palabras que designan objetos en el código C#. Su clase necesita un nombre y se ha usado el nombre *HelloWorld* para su clase. Esto convierte al nombre *HelloWorld* en un identificador. Su método también necesita un nombre y se ha usado el nombre *Main* para su función. Esto convierte al nombre *Main* en un identificador. El compilador de C# no permite normalmente usar ninguna de las palabras clave reservadas como nombres de identificador. Obtendrá un error si, por ejemplo, intenta aplicar el nombre `static` a una clase. Sin embargo, si realmente necesita usar el nombre de una palabra clave como identificador, puede poner delante del identificador el símbolo `@`. Esto invalida el error del compilador y permite usar una palabra clave como identificador. El listado 2.2 muestra cómo hacerlo. Es una modificación del código del listado 2.1 y define la palabra `virtual` como el nombre de la clase.

Listado 2.2. Cómo usar la palabra clave virtual como identificador de clase

```
class @virtual
{
    static void Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

Sin el precedente símbolo *@*, obtendría un error del compilador, como el que se muestra en la figura 2.2.



```
ex C:\WINDOWS\System32\cmd.exe
C:\>esc HelloWorld.cs
Compilador de Microsoft (R) Visual C# .NET versión 7.00.9466
para Microsoft (R) .NET Framework versión 1.0.3705
(C) Microsoft Corporation 2001. Reservados todos los derechos.
HelloWorld.cs(1,2): error CS1041: Se esperaba un identificador y 'virtual' es
una palabra clave
HelloWorld.cs(2,1): error CS1031: Se esperaba un tipo
C:\>
```

Figura 2.2. Si olvida el símbolo @ el compilador generará errores.

Uso de espacios en blanco

El texto de las aplicaciones C# puede incluir espacios, tabulaciones y caracteres de retorno. Estos caracteres son llamados *caracteres de espacio en blanco*. Los caracteres de espacio en blanco, que pueden ser colocados en cualquier lugar excepto en medio de una palabra clave o un identificador, ayudan a mejorar la legibilidad del código.

El compilador de C# pasa por alto la colocación de espacios en blanco cuando compila un programa. Esto significa que puede colocar cualquier carácter de espacio en blanco en cualquier lugar donde el compilador acepte un carácter de espacio en blanco. El compilador pasa por alto el uso de caracteres de retorno, tabulaciones y espacios. Puede usar cualquier combinación de espacios en blanco que desee en su código.

Los listados de este libro muestran estilos personales de colocación de espacios en blanco: los retornos están colocados antes y después de las llaves de apertura y cierre, y el código está sangrado a partir de las llaves. Sin embargo, no es obligatoria esta disposición en las aplicaciones C#. El listado 2.3 muestra una

disposición alternativa del código usando caracteres de espacio en blanco diferentes. No tema experimentar con el estilo que más le guste.

Listado 2.3. Una disposición de espacios en blanco alternativa

```
Class
HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

Si compila y ejecuta el listado 2.3, verá que se comporta exactamente igual que el código del listado 2.1: produce la cadena "Hello World!". La nueva disposición de espacios en blanco no tiene ningún efecto en el comportamiento del código que se ejecuta en el tiempo de ejecución.

Cómo iniciar programas con la función Main()

La aplicación que se muestra en el listado 2.1 define una clase con una función llamada `Main()`. La función `Main()` es una parte importante de las aplicaciones C#, ya que es donde comienza a ejecutarse nuestro programa. Todas las aplicaciones escritas en C# deben tener una clase con una función llamada `Main()`. La función `Main()` es conocida como el punto de entrada de sus aplicaciones y la ejecución de sus aplicaciones C# empieza con el código en `Main()`. Si el código contiene más de una clase, sólo una de ellas puede tener una función llamada `Main()`. Si olvida definir una función `Main()`, recibirá varios mensajes de error por parte del compilador, como se muestra en la figura 2.3.

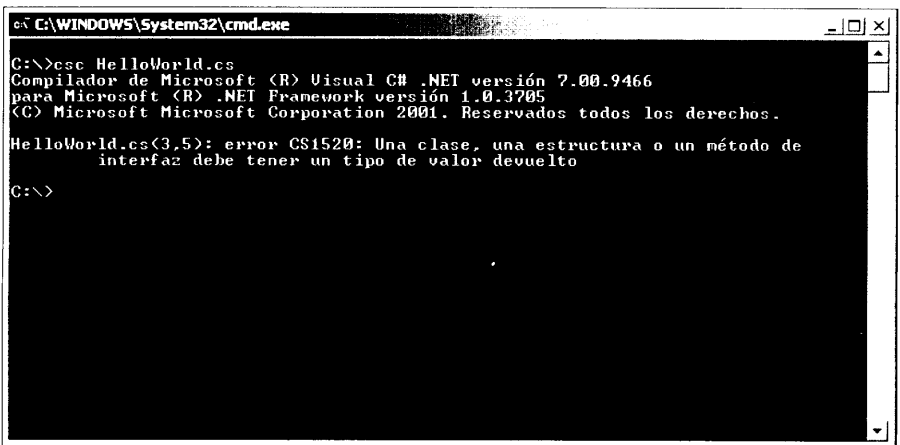


Figura 2.3. La ausencia de una función `Main()` produce errores de compilación.

La función `Main()` definida en el listado 2.1 no devuelve nada (de ahí la palabra clave `void`) y no toma argumentos (de ahí los paréntesis vacíos). El compilador C#, de hecho, acepta cualquiera de las cuatro posibles construcciones de la función `Main()`:

- `public static void Main()`
- `public static void Main(string[] Arguments)`
- `public static int Main()`
- `public static int Main(string [] Arguments)`

La primera variante, `public static void Main()`, es la forma usada en el listado 2.1.

La segunda, `public static void Main(string[] Arguments)`, no devuelve un valor al que la llama. Sin embargo, toma una matriz de cadenas. Cada cadena de la matriz se corresponde a un argumento de la línea de comando suministrado al ejecutar el programa. Por ejemplo, suponga que modifica el código del listado 2.1 para que el método `Main()` acepte una matriz de cadenas como argumento. Además, suponga que ejecuta ese código y suministra algunos argumentos de línea de comandos:

```
Listing2-1.exe Param1 Param2 Param3
```

En este caso, la matriz de cadenas que se pasa a la función `Main()` tiene los siguientes contenidos:

```
Arguments[0]: Param1  
Arguments[1]: Param2  
Arguments[2]: Param3
```

La tercera variante, `public static int Main()`, devuelve un valor entero. Que el valor que se devuelve sea entero se especifica mediante la palabra clave `int` de la declaración. Los valores enteros devueltos por `Main()` se usan como códigos de fin de programa. Por ejemplo, suponga que desea diseñar sus aplicaciones para que devuelvan un valor (supongamos 0) si la operación resulta satisfactoria y otro valor (supongamos 1) si la operación no se cumple. Si ejecuta la aplicación .NET desde un entorno que puede leer este código de terminación de programa, tiene suficiente información para determinar si el programa se ejecutó satisfactoriamente. La última variante de la función `Main()`, `public static int Main(string [] Arguments)`, especifica una función que proporciona argumentos de línea de comando en una matriz de cadenas y permite a la función devolver un código de terminación de programa.

Debe tener presente algunas cosas cuando trabaje con la función `Main()`:

- Las formas de devolución `void` de la función `Main()` siempre tienen un código de terminación de programa de 0.

- La palabra clave `static` es necesaria en todas las variantes de la función `Main()`.

Cuando se ejecuta una aplicación de C#, el usuario siempre proporciona los argumentos de la línea de comando. Sin embargo, si la aplicación de C# está escrita con una de las variantes de la función `Main()` que no toma argumentos, la aplicación será incapaz de leerlos. Está permitido que el usuario especifique argumentos en una aplicación de C# que no fue escrita para admitirlos (aunque no será muy útil).

Cómo comentar el código

Comentar el código le permite añadir notas a sus archivos fuente de C#. Estas notas pueden ayudarle a documentar el diseño y el funcionamiento de la aplicación. Puede colocar comentarios en cualquier parte del código fuente de C# donde sea posible usar espacios en blanco.

Cómo usar comentarios de una línea

Los comentarios de una línea empiezan con dos barras inclinadas y afecta al resto de la línea:

```
{ // esto es una llave de apertura
System.Console.WriteLine("C#"); // call WriteLine()
} // esto es una llave de cierre
```

Usar comentarios regulares

Los comentarios regulares empiezan con una barra inclinada seguida de un asterisco y su efecto permanece hasta que encuentra un asterisco seguido por una barra inclinada. Los comentarios regulares pueden extenderse por varias líneas:

```
/*
Esto es un comentario regular de C#.
Contiene varias líneas de texto,
Separadas por caracteres NewLine.
*/
```

El compilador de C# no permite incrustar un comentario regular en otro:

```
/*
comentario externo
  /*
  comentario interno
  */
mas texto de comentario externo
*/
```

No puede incrustar un comentario regular en otro porque el compilador encuentra los primeros caracteres `*/` y da por hecho que ha alcanzado el final del comentario de varias líneas. A continuación, supone que el siguiente texto seguido por los caracteres es código fuente de C# e intenta interpretarlo como tal.

Sin embargo, puede incrustar un comentario de una sola línea en un comentario regular:

```
/*  
comentario externo  
    // comentario interno  
mas texto de comentario externo  
*/
```

Cómo generar documentación XML a partir de comentarios

Una característica interesante del compilador de C# es que puede leer comentarios en un formato especial y generar documentación XML a partir de los comentarios. Puede entonces colocar este XML en la Web para facilitar un nivel extra de documentación a los programadores que necesiten comprender la estructura de sus aplicaciones.

Para usar esta función, debe hacer dos cosas:

- Usar tres barras inclinadas para los comentarios. El compilador de C# no genera ninguna documentación XML para ningún documento que no empiece con tres barras. Tampoco genera documentación XML para comentarios regulares de varias líneas.
- Use la opción `/doc` del compilador de C# para especificar el nombre del archivo que debería contener la documentación XML generada.

El listado 2.4 muestra la aplicación *Hello World!* con comentarios de documentación XML.

Listado 2.4. La aplicación Hello World! con comentarios XML

```
/// La clase HelloWorld es la única clase de la  
/// clase "HelloWorld". La clase implementa la función  
/// Main() de la aplicación. La clase no contiene otras  
/// funciones.  
  

```

```

static void Main()
{
    System.Console.WriteLine("Hello World!");
}
}

```

Puede compilar esta aplicación con la opción `/doc` para generar documentación XML para el código fuente:

```
csc /doc:HelloWorld.xml HelloWorld.cs
```

El compilador produce un `HelloWorld.exe` como era de esperar y también un archivo llamado `HelloWorld.xml`. Este archivo contiene un documento XML con sus comentarios de documentación XML incrustados en él. El listado 2.5 muestra el documento XML que se genera cuando se compila con la opción `/doc` el código del listado 2.4.

Listado 2.5. Documento XML generado para el código del listado 2.4

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>HelloWorld</name>
  </assembly>
  <members>
    <member name="T:HelloWorld">
      La clase HelloWorld es la única clase de la
      clase "HelloWorld". La clase implementa la función
      Main() de la aplicación. La clase no contiene otras
      funciones.
    </member>
    <member name="M:HelloWorld.Main">
      Esta es la función Main() para la clase del listado 2.4.
      No devuelve un valor y no toma ningún
      argumento. Escribe el texto " HelloWorld!" en la
      consola y luego sale.
    </member>
  </members>
</doc>

```

Ahora puede escribir una hoja de estilo para este documento en XML y mostrarlo en una página Web, proporcionando a los demás usuarios documentación actualizada de su código.

La principal porción del documento XML está en el elemento `<members>`. Este elemento contiene una etiqueta `<member>` para cada objeto documentado en el código fuente. La etiqueta `<member>` contiene un atributo, `name`, que designa al miembro documentado. El valor del atributo `name` empieza con un prefijo de una letra que describe el tipo de información en cuestión. La tabla 2.1 describe los posibles valores del atributo del nombre y su significado.

Tabla 2.1. Prefijos de atributo <member> "name="

Prefijo	Significado
E	El elemento proporciona documentación de un evento.
F	El elemento proporciona documentación de un campo.
M	El elemento proporciona documentación de un método.
N	El elemento proporciona documentación de un nombre de espacio.
P	El elemento proporciona documentación de una propiedad.
T	El elemento proporciona documentación de un tipo definido por el usuario. Éste puede ser una clase, una interfaz, una estructura, una enumeración o un delegado.
!	El compilador C# encontró un error y no pudo determinar el prefijo correcto de este miembro.

Tras el prefijo se colocan dos puntos y el nombre del miembro. El atributo `name=` indica el nombre de la clase para los miembros de tipo. Para los miembros de método, el atributo `name=` indica el nombre de la clase que contiene el método, seguida por un punto y a continuación el nombre del método.

Sus comentarios de documentación XML pueden incluir cualquier elemento XML válido para ayudarle en su tarea de documentación. La documentación de .NET Framework recomienda un conjunto de elementos XML que puede usar en su documentación.

El resto de esta sección examina cada uno de estos elementos. Recuerde que debe emplear XML válido en sus comentarios, lo que significa que cada elemento debe contener su elemento final correspondiente en alguna parte de sus comentarios.

NOTA: El término etiqueta se refiere a cualquier elemento descriptivo contenido en el XML. Las etiquetas siempre están entre los símbolos < y >.

<c>

Puede usar la etiqueta <c> para indicar que una pequeña parte del comentario debe ser tratada como código. Las hojas de estilo pueden usar este elemento para mostrar la porción de código del comentario con una fuente de tamaño fijo, como Courier:

```
/// Esta es la función <c>Main()</c> para la
/// clase HelloWorld.
```


<code>

Puede usar la etiqueta `<code>` para indicar que varias líneas de texto de sus comentarios deben ser tratadas como código:

```
/// Llamar a esta aplicación con tres argumentos
/// hará que la matriz de cadenas suministrada a Main()
/// contenga tres elementos:
/// <code>
/// Argument[0]: command line argument 1
/// Argument[1]: command line argument 2
/// Argument[2]: command line argument 3
/// </code>
```

<example>

Puede usar la etiqueta `<example>` para indicar un ejemplo de cómo usar las clases que desarrolle a otros programadores. Los ejemplos suelen incluir una muestra de código y quizás quiera usar las etiquetas `<example>` y `<code>` juntas:

```
/// <example>Aquí tiene un ejemplo de un cliente llamando
/// a este código:
/// <code>
/// ponga aquí su código de ejemplo
/// </code>
/// </example>
```

<exception>

Puede usar la etiqueta `<exception>` para documentar cualquier excepción que pueda surgir en el código del miembro. La etiqueta `<exception>` debe contener un atributo llamado `cref` cuyo valor especifica el tipo de excepción que se documenta. El valor del atributo `cref` debe estar entre comillas. El texto del elemento describe las condiciones en las que aparece la excepción:

```
/// <exception cref="System.Exception">
/// Aparece si el valor introducido es menor de 0.
/// </exception>
```

El compilador de C# asegura que el valor del atributo `cref` sea un tipo de datos válido. Si no lo es, el compilador genera un mensaje de aviso. A continuación se indica cómo documentar una función `Main()`:

```
/// <exception cref="junk">probando</exception>
```

hace que el compilador de C# genere un mensaje de aviso como el siguiente:

```
aviso CS1574: El comentario en XML 'Main()' tiene un atributo cref
'junk' que no se encuentra
```

En este caso, el compilador de C# todavía escribe la etiqueta `<exception>` en el archivo XML, pero pone un signo de exclamación antes del atributo `cref`:

```

<member name="M:MyClass.Main">
<exception cref="!:junk">probando</exception>
</member>

```

<list>

Puede usar la etiqueta <list> para describir una lista de elementos en la documentación. Puede describir una lista no numerada, una lista numerada o una tabla. La etiqueta <list> usa un atributo llamado `type` para describir el tipo de la lista. La tabla 2.2 enumera los posibles valores para el atributo `type` y describe su significado.

Tabla 2.2. Valores para el atributo "type" de la etiqueta <list>

Valor	Significado
bullet	La lista es una lista no numerada.
number	La lista es una lista numerada.
table	La lista es una tabla.

Los estilos `bullet` y `number` deberían también incluir una o más etiquetas <item> dentro de la etiqueta <list>.

Cada etiqueta <item> se corresponde a un elemento de la lista. Cada etiqueta <item> debería contener una etiqueta <description>, cuyo texto define el texto de la lista de elementos:

```

/// <list type="bullet">
/// <item>
/// <description>Este es el elemento 1.</description>
/// </item>
/// <item>
/// <description>Este es el elemento 2.</description>
/// </item>
/// </list>

```

El tipo de lista `table` también debe incluir una etiqueta <listheader>. La etiqueta <listheader> contiene una o más etiquetas <term> que describen los encabezamientos de las tablas:

```

/// <list type="table">
/// <listheader>
/// <term>Elemento de la tabla </term>
/// </listheader>
/// <item>
/// <description> Éste es el elemento 1.</description>
/// </item>
/// </list>

```

<param>

Use la etiqueta <param> para documentar un parámetro para una función. La etiqueta <param> usa un atributo, name, cuyo valor identifica al parámetro que se está documentando. El texto de la etiqueta <param> proporciona una descripción del parámetro:

```
/// <param name="Flag">
/// El valor debe ser 0 para desactivado o 1 para activado.
/// </param>
```

El compilador de C# asegura que el valor del atributo name realmente especifique el nombre de un parámetro. Si no es así, el compilador emite dos avisos. Por ejemplo, un código fuente como el siguiente:

```
/// <param name="junk">Esto es junk.</param>

public static void Main(string [] strArguments)
{
}
```

produce avisos como los siguientes:

```
aviso CS1572: El comentario XML en 'Main(string[])' tiene una
etiqueta de parametro para 'junk', pero no hay ningun parametro
con ese nombre
```

```
aviso CS1573: El parametro 'strArguments' no tiene una etiqueta
de parametro coincidente en el comentario XML (pero otros
parametros si)
```

El primer aviso dice que se encontró una etiqueta <param> con un atributo name cuyo valor no concuerda con ninguno de los parámetros de la función. El segundo aviso dice que a uno de los parámetros le falta una etiqueta <param>.

La etiqueta <param> se coloca en el archivo XML de documentación, incluso si el atributo name es incorrecto:

```
<member name="M:Class1.Main(System.String[])">
<param name="junk">Esto es junk.</param>
</member>
```

<paramref>

Puede usar la etiqueta <paramref> para hacer referencia a un parámetro desde una descripción. La etiqueta puede no tener ningún texto; sin embargo, lleva un atributo llamado name.

El valor del atributo name debe indicar el nombre del parámetro al que se hace referencia:

```
/// La matriz <paramref name="Arguments" /> contiene
/// los parametros especificados es la linea de comandos.
```

<permission>

Use la etiqueta <permission> para documentar los permisos disponibles en una función o variable dadas. El acceso al código y los datos de una clase puede significar el acceso a todo el código o puede ser restringido a cierto subconjunto de código. Puede usar la etiqueta <permission> para documentar la disponibilidad del código y sus datos.

La etiqueta <permission> hace uso de un atributo: `cref`. El valor del elemento `cref` debe designar la función o la variable cuyos permisos se están documentando:

```
/// <permission name="Main()">
/// Todo el mundo puede acceder a Main().
/// </permission>
```

<remarks>

Use la etiqueta <remarks> para añadir información. El elemento <remarks> es estupendo para mostrar una vista general de un método o una variable y su uso. La etiqueta <remarks> no tiene atributos y su texto contiene las observaciones:

```
/// <remarks>
/// La función Main() es el punto de entrada a la
/// aplicación. El CLR llamará a Main() para iniciar
/// la aplicación una vez que ésta se haya abierto.
/// </remarks>
```

<returns>

Use la etiqueta <returns> para describir un valor devuelto por una función. La etiqueta <returns> no tiene atributos y su texto contiene la información del valor devuelto:

```
/// <returns>
/// La función Main() devolverá 0 si la aplicación
/// procesó los datos correctamente y devolverá 1
/// en caso contrario.
/// </returns>
```

<see>

Use la etiqueta <see> para añadir una referencia a una función o variable que se encuentre en otra parte del archivo. El elemento <see> usa un atributo llamado `cref` cuyo valor especifica el nombre del método o variable al que se hace referencia. La etiqueta <see> no debe contener texto:

```
/// <see cref="Class1.Main" />
```

El compilador de C# asegura que el valor del atributo `cref` realmente especifique el nombre de un método o variable. Si no es así, el compilador emite un aviso. Por tanto, un código fuente como el siguiente:

```

/// <see cref="junk" />

public static void Main(string [] strArguments)
{
}

```

produce un aviso como el siguiente:

```

aviso CS1574: El comentario XML en 'Class1.Main(string[])' tiene
un atributo cref 'junk' que no se encuentra

```

La etiqueta `<see>` está situada en el archivo XML de documentación, incluso si el atributo `cref` es incorrecto:

```

<member name="M:Class1.Main(System.String[])">
  <see cref="!:junk"/>
</member>

```

<seealso>

Como `<see>`, puede usar la etiqueta `<seealso>` para añadir una referencia a una función o variable que esté en otra parte del archivo. Puede que necesite generar documentación que contenga una sección de referencias `<see>` además de una sección de referencias `See Also` y el compilador de C# le permite hacer esa distinción al admitir las etiquetas `<see>` y `<seealso>`. La etiqueta `<seealso>` usa un atributo llamado `cref` cuyo valor especifica el nombre del método o variable al que se hace referencia. La etiqueta `<seealso>` no debe contener texto:

```

/// <seealso cref="Class1.Main" />

```

El compilador de C# asegura que el valor del atributo `cref` realmente especifique el nombre de un método o variable. Si no es así, el compilador emite un aviso. Por tanto, un código fuente como el siguiente:

```

/// <seealso cref="junk" />

public static void Main(string [] strArguments)
{
}

```

produce un aviso como el siguiente:

```

aviso CS1574: El comentario XML en 'Class1.Main(string[])' tiene
un atributo cref 'junk' que no se encuentra

```

La etiqueta `<seealso>` está situada en el archivo XML de documentación, incluso si el atributo `cref` es incorrecto:

```

<member name="M:Class1.Main(System.String[])">
  <seealso cref="!:junk"/>
</member>

```

<summary>

Use la etiqueta `<summary>` para proporcionar una descripción resumida de un fragmento de código. Esta etiqueta no admite ningún atributo. Su texto debe describir la información resumida:

```
/// <summary>
/// La función Main() es el punto de entrada de
/// esta aplicación.
/// </summary>
```

La etiqueta `<summary>` es como la etiqueta `<remarks>`. Generalmente, debe usar la etiqueta `<summary>` para proporcionar información sobre un método o variable y la etiqueta `<remarks>` para proporcionar información sobre el tipo del elemento.

<value>

Use la etiqueta `<value>` para describir una propiedad de la clase. La etiqueta `<value>` no tiene atributos. Su texto debe documentar la propiedad:

```
/// <value>
/// La propiedad MyValue devuelve el número de registros
/// leídos de la base de datos.
/// </value>
```

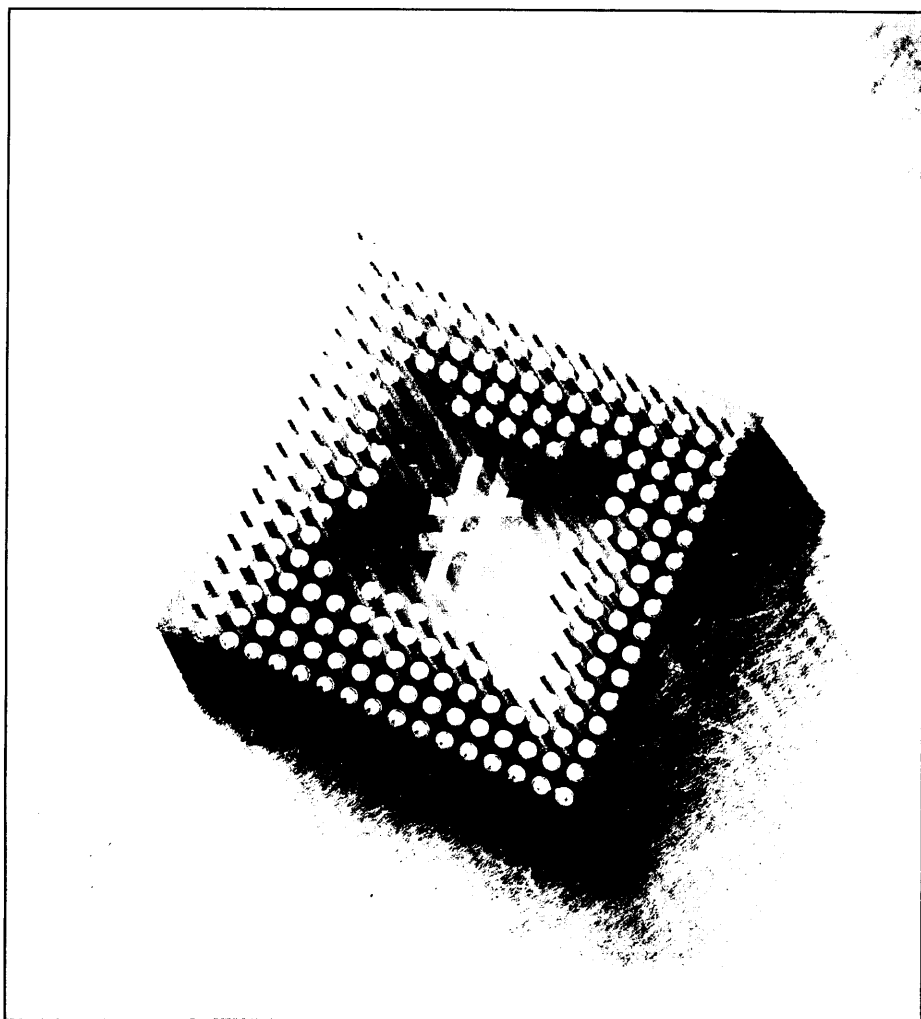
```
public int MyValue
{
    // ... el código de propiedad viene aquí ...
}
```

Resumen

Este capítulo muestra cómo crear aplicaciones C# con un simple editor de texto, como el Bloc de notas. También examina varias alternativas a Visual Studio para escribir código.

Ha construido su primera aplicación de C#. Las aplicaciones C#, independientemente de su tamaño, deben contener una clase con una función llamada `Main()`. La función `Main()` es el punto de partida de su aplicación de C#.

También aprendió a añadir comentarios al código fuente en C#. Puede añadir comentarios al código para ayudar a otros programadores a comprender cómo está estructurado el código fuente. También puede dar formato a los comentarios de tal modo que el compilador pueda convertir los comentarios en un documento XML; y añadiendo palabras clave especiales, puede hacer el documento XML muy rico e informativo.



3 Trabajar con variables

El código de C# suele trabajar con valores que no se conocen cuando se escribe el código. Puede que necesite trabajar con un valor leído de una base de datos en tiempo de ejecución o quizás necesite almacenar el resultado de un cálculo. Cuando necesite almacenar un valor en el tiempo de ejecución, use una variable. Las variables son los sustitutos de los valores con los que trabaja en su código.

Cómo dar nombre a sus variables

Cada variable que use en su código C# debe tener un nombre. El compilador de C# interpreta los nombres de las variables como identificadores y, por tanto, deben seguir las convenciones de designación de los identificadores:

- El primer carácter de un identificador debe empezar con una letra mayúscula o minúscula o con un carácter subrayado.
- Los caracteres que siguen al primero pueden ser cualquiera de los siguientes:
 - Una letra mayúscula o minúscula
 - Un número
 - Un subrayado

NOTA: C# admite código fuente escrito con caracteres Unicode. Si está escribiendo su código fuente usando un conjunto de caracteres Unicode, puede usar cualquier carácter de entre las clases de caracteres Unicode Lu, Ll, Lt, Lm, Lo, Nl, Mn, Mc, Nd, Pc y Cf en su identificador. Consulte la sección 4.5 de las especificaciones de Unicode si desea obtener más información sobre las clases de caracteres Unicode.

También puede usar una palabra clave de C# como nombre de variable, pero sólo si va precedida del carácter `@`. No obstante, no se trata de una práctica recomendable, ya que puede hacer que su código resulte difícil de leer, pero es factible y el compilador de C# lo permite.

Asignación de un tipo a una variable

A las variables de C# se les asigna un tipo, que es una descripción del tipo de datos que va a contener la variable. Quizás quiera trabajar con números enteros, números de coma flotante, caracteres, cadenas o incluso un tipo que puede definir en su código. Cuando define una variable en su código C#, debe asignar un tipo a la variable. La tabla 3.1 describe algunos de los tipos básicos de variables de C#.

Tabla 3.1. Tipos de datos comunes de C#

Tipo	Descripción
<code>sbyte</code>	Las variables con tipo <code>sbyte</code> pueden contener números enteros de 8 bits con signo. La 's' de <code>sbyte</code> significa 'con signo', lo que quiere decir que el valor de la variable puede ser positivo o negativo. El menor valor posible para una variable <code>sbyte</code> es -128 y el mayor es 127.
<code>byte</code>	Las variables con tipo <code>byte</code> pueden contener números enteros de 8 bits sin signo. A diferencia de las variables <code>sbyte</code> , las variables <code>byte</code> no tienen signo y sólo pueden contener números positivos. El valor más pequeño posible para una variable <code>byte</code> es 0; el valor más alto es 255.
<code>short</code>	Las variables con tipo <code>short</code> pueden contener números enteros de 16 bits con signo. El menor valor posible para una variable <code>short</code> es -32.768; el valor más alto es 32.767.
<code>ushort</code>	Las variables con tipo <code>ushort</code> pueden contener números enteros de 16 bits sin signo. La 'u' de <code>ushort</code> signifi-

Tipo	Descripción
int	<p>ca <i>sin signo</i>. El menor valor posible para una variable <code>ushort</code> es 0; el valor más alto es 65.535.</p> <p>Las variables con tipo <code>int</code> pueden contener números enteros de 32 bits con signo. El menor valor posible para una variable <code>int</code> es -2.147.483.648; el valor más alto es 2.147.483.647.</p>
uint	<p>Las variables con tipo <code>uint</code> pueden contener números enteros de 32 bits sin signo. La 'u' en <code>uint</code> significa <i>sin signo</i>. El menor valor posible para una variable <code>uint</code> variable es 0; el valor más alto posible es 4.294.967.295.</p>
long	<p>Las variables con tipo <code>long</code> pueden contener números enteros de 64 bits con signo. El menor valor posible para una variable <code>long</code> es 9.223.372.036.854.775.808; el valor más alto es 9.223.372.036.854.775.807.</p>
ulong	<p>Las variables con tipo <code>ulong</code> pueden contener números enteros de 64 bits sin signo. La 'u' en <code>ulong</code> significa <i>sin signo</i>. El menor valor posible para una variable <code>ulong</code> es 0; el valor más alto es 18.446.744.073.709.551.615.</p>
char	<p>Las variables con tipo <code>char</code> pueden contener caracteres Unicode de 16 bits. El menor valor posible para una variable <code>char</code> es el carácter Unicode cuyo valor es 0; el valor más alto posible es el carácter Unicode cuyo valor es 65.535.</p>
float	<p>Las variables con tipo <code>float</code> pueden contener un valor de coma flotante de 32 bits con signo. El menor valor posible para una variable <code>float</code> es aproximadamente 1,5 por 10 elevado a 45; el valor más alto es aproximadamente 3,4 por 10 elevado a 38.</p>
double	<p>Las variables con tipo <code>double</code> pueden contener un valor de coma flotante de 64 bits con signo. El menor valor posible para una variable <code>double</code> es aproximadamente 5 por 10 elevado a 324; el valor más alto es aproximadamente 1,7 por 10 elevado a 308.</p>
decimal	<p>Las variables con tipo <code>decimal</code> pueden contener un valor de coma flotante de 128 bits con signo. Las variables de tipo <code>decimal</code> son buenas para cálculos financieros. El menor valor posible para una variable <code>decimal</code> es aproximadamente 1 por 10 elevado a 28; el valor más alto es aproximadamente 7,9 por 10 elevado a 28.</p>
bool	<p>Las variables con tipo <code>bool</code> pueden tener uno de los dos posibles valores: <code>true</code> o <code>false</code>. El uso del tipo <code>bool</code> es</p>

Tipo	Descripción
	<p>una de las partes en las que C# se aparta de su legado C y C++. En ellos, el valor entero 0 era sinónimo de <code>false</code> y cualquier valor que no fuese cero era sinónimo de <code>true</code>. Sin embargo, en C# los tipos no son sinónimos. No puede convertir una variable entera en su valor equivalente <code>bool</code>. Si quiere trabajar con una variable que necesita tener una condición verdadera o falsa, use una variable <code>bool</code> y no una variable <code>int</code>.</p>

Cómo aplicar tamaño a sus variables

Se estará preguntando por qué C# admite todos estos tipos de variables diferentes. Los valores más pequeños pueden colocarse en variables de mayores tipos; por lo tanto ¿por qué usar los tipos más pequeños? Si una variable `short` puede contener valores desde -32.768 hasta 32.767, y una grande puede contener valores desde -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807, entonces es evidente que todos los posibles valores `short` pueden ser almacenados en una variable `long`. Entonces, ¿para qué sirven los tipos `short`? ¿Por qué no usar un variable `long` para todo?

Una respuesta es el consumo de memoria. Una variable `long` puede contener valores más grandes, pero también necesita más memoria. Una variable `short` usa 16 bits (dos bytes), mientras que una grande usa 32 bits (cuatro bytes de memoria). Si va a trabajar con valores que no van más allá del límite de una variable `short`, use la variable `short`. Es una buena costumbre usar toda la memoria que necesite, pero no usar más de la necesaria.

Cómo declarar sus variables

Antes de poder usar su variable, debe declararla en su código. Al declarar una variable informa al compilador de C# del tipo y nombre de su variable. Una variable se declara escribiendo su tipo, seguido de algún espacio en blanco y, a continuación, del nombre de la variable. Termine la declaración con un punto y coma.

Algunos ejemplos de declaración de variables son:

```
byte MyByteVariable;
int _Value123;
ulong AVeryLargeNumber;
```

NOTA: Los *espacios en blanco* se definen como cualquier número de espacios necesario para mejorar la legibilidad del código.

Debe declarar sus variables dentro de una clase o de una función. El siguiente código es válido:

```
class MyClass
{
    int MyIntegerVariable;

    static void Main()
    {
        float AnotherVariable;

        System.Console.WriteLine("Hello!");
    }
}
```

NOTA: Puede declarar una variable donde desee, pero tenga en cuenta que si la declara en una función, como se muestra en la variable `AnotherVariable` del ejemplo anterior, sólo el código incluido en esa función podrá trabajar con la variable. Si la declara dentro de la clase, como en la variable `MyIntegerVariable` (también en el ejemplo anterior), todo el código de esa clase podrá trabajar con esa variable. Si toma el código del ejemplo y le añade otra función a la clase, el código de esa nueva función podrá trabajar con la variable `MyIntegerVariable` pero no podrá trabajar con la variable `AnotherVariable`. Si esta nueva función intenta acceder a la variable `AnotherVariable` declarada en la función `Main()`, obtendrá el siguiente mensaje de error del Compilador de C#:

```
error CS0103: El nombre 'AnotherVariable' no existe en la clase o espacio de nombre 'MyClass'
```

Uso de valores por defecto en las variables

En otros lenguajes de programación, es posible trabajar con una variable sin antes asignarle un valor. Este vacío es una fuente de errores, como demuestra el siguiente código:

```
class MyClass
{
    static void Main()
```

```

    {
        int MyVariable;

        // ¿Cuál es el valor de "MyVariable" aquí?
    }
}

```

¿Cuál es el valor de `MyVariable` cuando se ejecuta `Main()`? Su valor es desconocido porque el código no asigna ningún valor a la variable.

Los programadores de C# eran conscientes de los errores que podían aparecer como resultado de usar variables a las que no se las ha asignado explícitamente un valor.

El compilador de C# busca condiciones como ésta y genera un mensaje de error. Si la función `Main()` del código anterior hace referencia a la variable `MyVariable` sin que se le haya asignado un valor, el compilador de C# muestra el siguiente mensaje de error:

```
error CS0165: Uso de la variable local no asignada 'MyVariable'
```

C# distingue entre variables *asignadas* y *no asignadas*. Las variables asignadas han recibido un valor en algún punto del código y las variables no asignadas no han recibido ningún valor en el código. En C# no se permite trabajar con variables no asignadas porque sus valores son desconocidos y emplear estas variables puede generar errores en su código.

En algunos casos, C# otorga valores por defecto a variables. Uno de estos casos es una variable declarada en el nivel de clase. Las variables de clase reciben valores por defecto si no se les asigna un valor en el código. Modifique el código anterior cambiando la variable `MyVariable` de una variable declarada en el nivel de función a una variable declarada en el nivel de clase:

```

class MyClass
{
    static int MyVariable;

    static void Main()
    {
        // MyVariable recibe un valor por
        // defecto y puede ser usada aquí
    }
}

```

Esta acción mueve la declaración de la variable dentro de la variable de clase y la variable ahora es accesible para todo el código incluido en la clase, no sólo en la función `Main()`. C# asigna valores por defecto a variables de nivel de clase y el compilador de C# le permite trabajar con la variable `MyVariable` sin asignarle un valor inicial.

La tabla 3.2. enumera los valores que se asignan por defecto a las variables de clase.

Tabla 3.2. Valores por defecto de las variables

Tipo de variable	Valor por defecto
sbyte	0
byte	0
short	0
ushort	0
int	0
uint	0
long	0
ulong	0
char	carácter Unicode con valor 0
float	0.0
double	0.0
decimal	0.0
bool	false

Asignación de valores a variables

En algún punto de su código, querrá darle a sus variables un valor. Asignar un valor a una variable es sencillo: se escribe el nombre de la variable, el símbolo igual, el valor y se termina la instrucción con un punto y coma:

```
MyVariable = 123;
```

Puede asignar un valor a la variable cuando declara la variable:

```
int MyVariable = 123;
```

Aprenderá otros medios de asignar valores a las variables en las secciones posteriores.

Uso de matrices de variables

Las matrices simplemente son bytes de memoria contiguos que almacenan datos a los que se accede usando un índice que está en la matriz. En esta sección se analizan las matrices unidimensionales, multidimensionales y escalonadas.

Declaración de matrices unidimensionales

Suponga que está escribiendo una aplicación de C# para que los profesores introduzcan las calificaciones de los exámenes para cada uno de los estudiantes de su clase y quiere declarar variables que almacenen la puntuación del examen de cada alumno.

Como la calificación del examen está entre 0 y 100 puede usar tipos `byte`. Si su programa admite 25 estudiantes en una clase, su primer pensamiento puede ser declarar 25 variables por separado:

```
Byte    TestScoreForStudent1;  
Byte    TestScoreForStudent2;  
Byte    TestScoreForStudent3;  
// ... mas ...  
byte    TestScoreForStudent25;
```

Esto requerirá mucho tiempo y su código será difícil de leer y mantener con todas esas variables. Lo que necesita es un modo de decir, "Quiero tener una colección de 25 variables". Es decir, lo que queremos es una matriz.

Una matriz es una colección de variables, cada una de las cuales tiene el mismo tipo de variable. Las matrices tienen un tamaño, que especifica cuántos elementos pueden contener.

La declaración de una matriz es algo así:

```
byte [] TestScoresForStudents;
```

La declaración `byte` especifica que todos los elementos de la matriz son valores de tipo `byte`. Mediante los corchetes se indica al compilador de C# que quiere crear una variable de matriz, en vez de una sola variable, y el identificador `TestScoresForStudents` es el nombre de la matriz.

El elemento que falta en esta declaración es el tamaño de la matriz. ¿Cuántos elementos puede contener esta matriz? El tamaño de la matriz se especifica mediante el operador de C# `new`. Este operador indica al compilador de C# que quiere reservar suficiente memoria para una nueva variable: en este caso, una matriz de 25 variables `byte`:

```
byte [] TestScoresForStudents;  
  
TestScoresForStudents = new byte[25];
```

La palabra clave `byte` indica al compilador que quiere crear una nueva matriz de variables `byte`, y `[25]` indica que quiere reservar suficiente espacio para 25 variables `byte`. Cada variable de la matriz se denomina *elemento* de la matriz, y la matriz que acaba de crear contiene 25 elementos.

Debe acordarse de especificar el tipo de matriz cuando use la palabra clave `new`, aunque ya haya especificado el tipo de la matriz cuando la declaró. Si olvida hacerlo, obtendrá un mensaje de error del compilador. El código:

```
byte [] TestScoresForStudents;  
  
TestScoresForStudents = new [25];
```

hace que el compilador de C# emita el error:

```
error CS1031: Se esperaba un tipo
```

Este error aparece porque el código no tiene un tipo de variable entre la nueva palabra clave y el tamaño de la matriz.

También debe recordar usar el mismo tipo que usó cuando declaró la matriz. Si usa un tipo diferente, obtendrá otro mensaje de error, como demuestra el siguiente código:

```
byte [] TestScoresForStudents;  
  
TestScoresForStudents = new long[25];
```

Este código hace que el compilador de C# emita el error:

```
error CS0029: No se puede convertir implícitamente el tipo  
'long[]' a 'byte[]'
```

El error se produce porque el tipo de la instrucción (`byte`) no concuerda con el tipo usado en la nueva instrucción (`long`).

Las matrices como ésta se llaman *matrices unidimensionales*. Las matrices unidimensionales tienen un factor que determina su tamaño. En este caso, el único factor que determina el tamaño de la matriz es el número de estudiantes de la clase.

El valor inicial de los elementos de la matriz queda determinado por los valores por defecto del tipo de matriz. Cada elemento de la matriz se inicializa con un valor por defecto de acuerdo con la tabla 3.2. Como esta matriz contiene elementos de tipo `byte`, cada elemento de la matriz tiene un valor por defecto de 0.

Cómo trabajar con los valores de las matrices unidimensionales

Acaba de crear una matriz con 25 elementos de tipo `byte`. Cada elemento de la matriz tiene un número. El primer elemento de la matriz ocupa el índice cero, y el último elemento de la matriz es uno menos que el número de elementos de la matriz (en este caso, el último elemento es el elemento 24). Las matrices de C# se llaman *matrices de base cero* porque los números de sus elementos empiezan en el cero.

Trabajar con un elemento individual de la matriz es sencillo. Para obtener un valor de la matriz, acceda a él con el nombre de la variable y el número de la variable entre corchetes, como se muestra en el siguiente código:

```
byte    FirstTestScore;  
  
FirstTestScore = TestScoresForStudents[0];
```


Este código accede al primer elemento de la matriz `TestScoresForStudents` y asigna su valor a la primera variable `FirstTestScore` variable.

Para poner un valor en la matriz, simplemente acceda al elemento usando la misma sintaxis, pero coloque el nombre de la matriz y el número del elemento a la izquierda del signo igual:

```
TestScoresForStudents[9] = 100;
```

Este código almacena el valor 100 en el décimo elemento de la matriz `TestScoresForStudents`. C# no permite acceder a un elemento que no se encuentre en la matriz. Como la matriz que ha definido contiene 25 elementos, los números de los elementos posibles van de 0 a 24, inclusive. Si usa un número de elemento inferior a 0 o mayor que 24, obtendrá un mensaje de tiempo de ejecución, como se muestra en el siguiente código:

```
TestScoresForStudents[1000] = 123;
```

Este código se compila sin errores, pero al ejecutar la aplicación se produce un error porque no hay un elemento 1000 en su matriz de 25 elementos. Cuando se llega a esta instrucción, el Entorno de ejecución común (CLR) detiene el programa e inicia un mensaje de excepción:

```
Exception occurred: System.IndexOutOfRangeException: An exception of type System.IndexOutOfRangeException was thrown.
```

`IndexOutOfRangeException` indica que la aplicación intentó acceder a un elemento con un número de elemento que no tiene sentido para la matriz.

Inicialización de valores de elementos de matriz

Supongamos que quiere crear una matriz de cinco números enteros y que el valor de cada elemento sea distinto del que nos ofrecen por defecto. Puede escribir instrucciones individuales para inicializar los valores de la matriz:

```
int [] MyArray;  
  
MyArray = new int [5];  
MyArray[0] = 0;  
MyArray[1] = 1;  
MyArray[2] = 2;  
MyArray[3] = 3;  
MyArray[4] = 4;
```

Si al escribir el código ya conoce los valores con los que quiere inicializar la matriz, puede especificar los valores en una lista separada por comas y encerrada entre llaves. La lista se coloca en la misma línea que la declaración de matriz. Puede poner todo el código anterior en una sola línea escribiendo lo siguiente:

```
int [] MyArray = { 0, 1, 2, 3, 4};
```

Usando esta sintaxis, no especifica el nuevo operador o el tamaño de la matriz. El compilador de C# examina su lista de valores y calcula el tamaño de la matriz.

Declaración de matrices multidimensionales

Puede pensar en una matriz unidimensional como en una línea. Se extiende en una dirección. Puede imaginarse una matriz multidimensional con dos dimensiones como un trozo de papel de gráfica. Sus dimensiones no sólo se extienden hacia fuera, sino también hacia abajo. Esta sección estudia los tipos más comunes de matriz.

Uso de matrices rectangulares

Continuamos con el ejemplo de las calificaciones de los exámenes. La matriz unidimensional definida en la sección previa contenía un conjunto de calificaciones de 25 estudiantes. Cada estudiante dispone de un elemento en la matriz para almacenar una calificación. Pero, ¿qué ocurre si quiere almacenar varias calificaciones para varios estudiantes? Ahora tiene una matriz con dos factores que afectan a su tamaño: el número de estudiantes y el número de exámenes. Suponga que sus 25 estudiantes harán 10 exámenes a lo largo del curso. Eso significa que el profesor tiene que corregir 250 exámenes por curso. Puede declarar una matriz unidimensional para que contenga las 250 calificaciones:

```
byte [] TestScoresForStudents;  
  
TestScoresForStudents = new byte[250];
```

Pero esto podría resultar confuso. ¿Cómo se usa esta matriz? ¿Aparecen primero las puntuaciones de un solo estudiante o colocamos en primer lugar las calificaciones del primer estudiante?

Un modo mejor de declarar una matriz consiste en especificar cada dimensión por separado. Declarar una matriz multidimensional es tan sencillo como colocar comas entre los corchetes. Coloque una coma menos que el número de dimensiones necesarias para su matriz multidimensional, como en la siguiente declaración:

```
byte [,] TestScoresForStudents;
```

Esta declaración define una matriz multidimensional de dos dimensiones.

Usando el operador `new` para crear una nueva matriz de este tipo es tan sencillo como especificar las dimensiones individualmente, separadas por comas, en los corchetes, como se muestra en el siguiente código:

```
byte [,] TestScoresForStudents;  
  
TestScoresForStudents = new byte [10, 25];
```

Este código indica al compilador de C# que quiere crear una matriz con una dimensión de 10 y otra dimensión de 25. Puede imaginar una matriz de dos di-

mensionales como una hoja de cálculo de Microsoft Excel con 10 filas y 25 columnas. La tabla 3.3 recoge el aspecto que podría tener esta matriz si sus datos estuvieran en una tabla.

Tabla 3.3. Representación en una tabla de una matriz de dos dimensiones

Examen	Estudiante 1	Estudiante 2	Estudiante 3...	Estudiante 25
Examen 1	90	80	85	75
Examen 2	95	85	90	80
...
Examen 10	100	100	100	100

Para acceder a los elementos de una matriz de dos dimensiones, use las mismas reglas para numerar elementos que en las matrices unidimensionales. (Los números de elemento van de 0 a uno menos que la dimensión de la matriz.) También se usa la misma sintaxis de coma que usó con el operador `new`. Escribir código para almacenar una calificación de 75 para el primer examen del alumno 25 sería algo así:

```
TestScoresForStudents[0, 24] = 75;
```

Leer la calificación del quinto examen del alumno 16 sería algo así:

```
byte FifthScoreForStudent16;
```

```
FifthScoreForStudent16 = TestScoresForStudents[4, 15];
```

En otras palabras, cuando trabaje con una matriz de dos dimensiones y considere la matriz como una tabla, considere la primera dimensión como el número de fila de la tabla y el segundo número como el número de columna.

Puede inicializar los elementos de una matriz multidimensional cuando declare la variable de matriz. Para ello, coloque cada conjunto de valores para una sola dimensión en una lista delimitada por comas rodeada por llaves:

```
int [,] MyArray = {{0, 1, 2}, {3, 4, 5}};
```

Esta instrucción declara una matriz de dos dimensiones con dos filas y tres columnas. Los valores enteros 0, 1 y 2 están en la primera fila, y los valores 3, 4 y 5 están en la segunda fila.

Las matrices de dos dimensiones con esta estructura se llaman *matrices rectangulares*. Estas matrices tienen la forma de una tabla: cada fila de la tabla tiene el mismo número de columnas.

C# permite definir matrices con más de dos dimensiones. Para ello basta con utilizar más comas en la declaración de la matriz.

Puede definir una matriz de cuatro dimensiones con tipo `long`. por ejemplo. con la siguiente definición:

```
long [,,,] ArrayWithFourDimensions;
```

Asegúrese de definir todas las dimensiones cuando use el operador `new`:

```
ArrayWithFourDimensions = new long [5, 10, 15, 20];
```

El acceso a los elementos de la matriz se realiza de la misma manera. No olvide especificar todos los elementos de la matriz:

```
ArrayWithFourDimensions[0, 0, 0, 0] = 32768436;
```

Definición de matrices escalonadas

C# permite definir matrices escalonadas. en las que cada fila puede tener un número diferente de columnas. Volvamos al ejemplo de las calificaciones de los estudiantes para explicarlo. Suponga también que el número máximo de exámenes es 10. pero algunos estudiantes no tienen que hacer los últimos exámenes si obtuvieron buena nota en los anteriores. Puede crear una matriz rectangular para lo que necesita almacenar. pero puede acabar con elementos que no se utilizan en su matriz rectangular. Si algunos estudiantes no hacen todos los exámenes. tendrá elementos en su matriz rectangular que no se usan. Estos elementos desaprovechados equivalen a memoria desperdiciada. lo que pretendemos evitar.

Una estrategia mejor consiste en definir una matriz en la que cada elemento de la matriz sea. en si mismo. una matriz. La figura 3.1 ejemplifica este concepto. Muestra al estudiante 1 con espacio para tres calificaciones. al estudiante 2 con espacio para cinco calificaciones. al estudiante 3 con espacio para dos calificaciones y al estudiante 25 con espacio para las diez calificaciones (los otros estudiantes no aparecen en la figura).

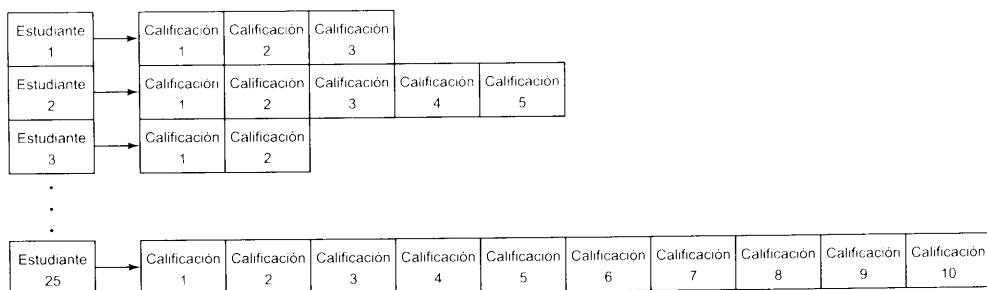


Figura 3.1. Las matrices escalonadas permiten definir una matriz que contiene otras matrices, cada una con un número diferente de elementos.

Estas matrices escalonadas tienen dos dimensiones. como las matrices rectangulares. pero cada fila puede tener un número de elementos diferente (lo que da a las matrices su aspecto escalonado).

Las matrices escalonadas se definen utilizando dos grupos de corchetes inmediatamente después del nombre del tipo de matriz. Cuando hace una llamada a `new`, se especifica un tamaño para la primera dimensión (la matriz `student` en nuestro ejemplo), pero no la segunda. Tras definir la primera matriz, haga una nueva llamada a `new` para definir las otras matrices (las matrices `score` en nuestro ejemplo):

```
byte [][] ArraysOfTestScores;  
  
ArraysOfTestScores = new byte [25] [];  
ArraysOfTestScores[0] = new byte[3];  
ArraysOfTestScores[1] = new byte[5];  
ArraysOfTestScores[2] = new byte[2];  
ArraysOfTestScores[24] = new byte[10];
```

Una vez que ha construido la matriz escalonada, puede acceder a sus elementos de la misma forma que en una matriz rectangular.

Tipos de valor y de referencia

Recuerde que debe usar la palabra clave `new` para crear la matriz. Este requisito es distinto de los tipos que hemos visto hasta ahora. Cuando trabaje con código que use variables `int` o `long`, por ejemplo, puede usar la variable sin usar `new`:

```
int IntegerVariable;  
  
IntegerVariable = 12345;
```

¿Por qué son tan diferentes las matrices? ¿Por qué se necesita utilizar `new` al crear una matriz? La respuesta está en la diferencia entre tipos de valor y tipos de referencia.

Con un tipo de valor, la variable retiene el valor de la variable. Con un tipo de referencia, la variable retiene una referencia al valor almacenado en algún otro sitio de la memoria. Puede imaginar una referencia como una variable que apunta hacia otra parte de memoria. La figura 3.2 muestra la diferencia.

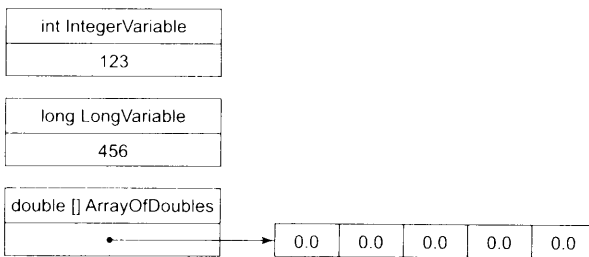


Figura 3.2. Los tipos de valor contienen datos. Los tipos de referencia contienen referencias a datos situados en algún otro lugar de la memoria.

Cada uno de los tipos comentados hasta este punto es un tipo de valor. Las variables proporcionan suficiente capacidad de almacenamiento para los valores que pueden contener y no necesita `new` para crear un espacio para sus variables. Las matrices son tipos de valor y los objetos son tipos de referencia. Sus valores están en algún otro lugar de la memoria y no necesita usar la palabra clave `new` para crear suficiente espacio para sus datos.

Aunque necesita usar la palabra clave `new` para crear espacio de memoria para un tipo de referencia, no necesita escribir ningún código que borre la memoria cuando haya acabado de usar la variable. El CLR contiene un mecanismo llamado *recolector de elementos no utilizados* que realiza la tarea de liberar la memoria que no se usa. El CLR ejecuta el recolector de elementos no utilizados mientras se ejecuta su aplicación de C#. El recolector de elementos no utilizados registra su programa buscando memoria no usada por ninguna de sus variables. Liberar la memoria que ya no se usa es tarea del recolector de elementos no utilizados.

Cómo convertir tipos de variable

Puede topar con una situación en la que tenga una variable de un tipo, pero necesite trabajar con un fragmento de código que necesite otro tipo. Si, por ejemplo, está trabajando con una variable de tipo `int` y necesita pasar el valor a una función que requiere el uso de una variable de tipo `long`, entonces necesita realizar una conversión de la variable `int` en variable `long`.

C# admite dos tipos de conversiones: conversiones implícitas y conversiones explícitas. Las siguientes secciones describen cada uno de estos tipos de conversiones.

Conversiones implícitas

El compilador de C# realiza automáticamente las conversiones implícitas. Examine el siguiente código:

```
int IntegerVariable;
long LongVariable;

IntegerVariable = 123;
LongVariable = IntegerVariable;
```

En este código, a una variable de tipo entero se le asigna el valor 123 y a una variable `long` se le asigna el valor de la variable de tipo entero. Cuando se ejecute este código, el valor de `LongVariable` es 123.

El compilador de C# convierte el valor de la variable de tipo entero a un valor `long` porque la conversión de un valor `int` a un valor `long` es una de las conversiones implícitas permitidas por C#. La tabla 3.4 recoge las conversiones

implícitas permitidas por C#. La primera columna enumera el tipo original de la variable y la fila superior enumera los tipos de datos a los que puede convertirlo. Una X en la celdilla significa que puede convertir implícitamente el tipo de la izquierda al tipo en la parte superior.

Tabla 3.4. Conversiones implícitas de tipo de valor

----	<i>sbyte</i>	<i>byte</i>	<i>short</i>	<i>ushort</i>	<i>int</i>	<i>uint</i>	<i>long</i>	<i>char</i>	<i>float</i>	<i>ulong</i>	<i>decimal</i>	<i>double</i>
<i>sbyte</i>	X	-	X	-	X	-	X	-	X	-	X	-
<i>byte</i>	-	X	X	X	X	X	X	-	X	X	X	-
<i>short</i>	-	-	X	-	X	-	X	-	X	-	X	X
<i>ushort</i>	-	-	-	X	X	X	X	-	X	X	X	X
<i>int</i>	-	-	-	-	X	-	X	-	X	-	X	X
<i>uint</i>	-	-	-	-	-	X	X	-	X	X	X	X
<i>long</i>	-	-	-	-	-	-	X	-	X	-	X	X
<i>char</i>	-	-	-	X	X	X	X	X	X	X	X	X
<i>float</i>	-	-	-	-	-	-	-	-	X	-	-	X
<i>ulong</i>	-	-	-	-	-	-	-	-	X	X	X	X

NOTA: No puede convertir ningún tipo a un tipo `char` (excepto mediante la variable `char`, lo que en realidad no es una conversión). Además, no puede hacer conversiones entre los tipos `floating-point` y los tipos decimales.

Conversiones explícitas

Si escribe código que intente convertir un valor que use tipos no admitidos por una conversión implícita, el compilador de C# genera un error, como muestra el siguiente código:

```
char CharacterVariable;
int IntegerVariable;

IntegerVariable = 9;
CharacterVariable = IntegerVariable;
```

El compilador de C# produce el siguiente error:

```
error CS0029: No se puede convertir implícitamente el tipo
'int' a 'char'
```

Este error se produce porque ninguna conversión implícita admite la conversión de una variable `int` a una variable `char`.

Si realmente necesita hacer esta conversión, tiene que realizar una conversión explícita. Las conversiones explícitas se escriben en su código fuente y le dicen al compilador "haz que se produzca esta conversión, aunque no puede ser realizada implícitamente". Escribir una conversión explícita en el código C# requiere colocar el tipo al que está convirtiendo entre paréntesis. Los paréntesis se colocan justo antes de la variable que está usando como fuente de la conversión. A continuación se incluye el código anterior pero usando una conversión explícita:

```
char CharacterVariable;
int IntegerVariable;

IntegerVariable = 9;
CharacterVariable = (char)IntegerVariable;
```

Esta técnica es conocida como *conversión* de la variable de tipo entero a una variable de tipo carácter. Algunos tipos no pueden ser convertidos, ni siquiera mediante una operación de conversión explícita. La tabla 3.5 enumera las conversiones explícitas que admite C#. La primera columna enumera el tipo original de la variable y la fila superior enumera los tipos de datos a los que puede convertirlo. Una X en la celdilla significa que puede convertir explícitamente el tipo de la izquierda al tipo de la parte superior usando la operación de conversión explícita.

Tabla 3.5. Conversiones explícitas de tipo de valor

---	sbyte	byte	short	ushort	int	uint	long	char	float	ulong	decimal	double
sbyte	X	X	-	X	X	-	X	X	-	-	-	-
byte	X	X	-	-	-	-	-	X	-	-	-	-
short	X	X	X	X	-	X	X	X	-	-	-	-
ushort	X	X	X	X	-	-	-	X	-	-	-	-
int	X	X	X	X	X	X	-	X	-	X	-	-
uint	X	X	X	X	X	X	-	X	-	-	-	-
long	X	X	X	X	X	X	X	X	-	X	-	-
float	X	X	X	X	X	X	X	X	X	X	X	-
ulong	X	X	X	X	X	X	X	X	-	X	-	-
decimal	X	X	X	X	X	X	X	X	X	X	X	X
double	X	X	X	X	X	X	X	X	X	X	X	X

También puede realizar conversiones explícitas sobre tipos de valor convirtiendo explícitamente el valor al tipo apropiado, como se muestra en el siguiente

ejemplo. C# permite usar un operador de conversión explícita incluso con conversiones implícitas, si lo desea:

```
int integerVariable;
long ongVariable;

IntegerVariable = 123;
LongVariable = (long)IntegerVariable;
```

Esta sintaxis no es necesaria, porque C# permite conversiones implícitas de variables `int` a variables `long`, pero puede escribirlo si quiere.

Cómo trabajar con cadenas

C# admite un tipo de referencia llamado `string`. El tipo de dato `string` representa una cadena de caracteres Unicode.

NOTA: Unicode es un estándar mundial de codificación de caracteres. Los caracteres Unicode tienen 16 bits, con lo que admiten 65.536 caracteres posibles. Los caracteres ANSI tienen 8 bits, y aceptan hasta 256 caracteres posibles.

Use el siguiente código para crear e inicializar una cadena en C#:

```
string MyString;

MyString = "Hello from C#!";
```

Al igual que con el resto de las variables, puede inicializar una cadena en la misma línea que su declaración:

```
string MyString = "Hello from C#!";
```

Uso de caracteres especiales en cadenas

C# permite usar una sintaxis especial para insertar caracteres especiales en su cadena. Estos caracteres especiales aparecen en la tabla 3.6.

Tabla 3.6. Caracteres especiales de C#

Caracteres	Función
<code>\t</code>	Los caracteres especiales <code>\t</code> incrustan una tabulación en la cadena. Una cadena definida como <code>hello\tthere</code>

Caracteres**Función**

	se almacena en memoria con un carácter de tabulación entre las palabras <i>hello</i> y <i>there</i> .
<code>\r</code>	Los caracteres especiales <code>\r</code> incrustan un retorno de carro en la cadena. Una cadena definida como <code>hello\rthere</code> se almacena en memoria con un retorno de carro entre las palabras <i>hello</i> y <i>there</i> . El carácter retorno de carro devuelve el cursor al principio de la línea, pero no mueve el cursor una línea por debajo.
<code>\v</code>	Los caracteres especiales <code>\v</code> insertan una tabulación vertical en la cadena. Una cadena definida como <code>hello\vthere</code> se almacena en memoria con un carácter de tabulación vertical entre las palabras <i>hello</i> y <i>there</i> .
<code>\f</code>	Los caracteres especiales <code>\f</code> insertan un carácter de impresión de página en la cadena. Una cadena definida como <code>hello\fthere</code> se almacena en memoria con un carácter de impresión de página entre las palabras <i>hello</i> y <i>there</i> . Las impresoras suelen interpretar un carácter de impresión de página como una señal para pasar a una nueva página.
<code>\n</code>	Los caracteres especiales <code>\n</code> insertan una nueva línea en la cadena. Una cadena definida como <code>hello\nthere</code> se almacena en memoria con un carácter de nueva línea entre las palabras <i>hello</i> y <i>there</i> . La comunidad de desarrolladores de software ha debatido durante mucho tiempo la interpretación del carácter de nueva línea. Siempre ha significado, "mueve la posición de la siguiente salida una línea más abajo". La duda está en si la operación también incluye mover la siguiente posición al primer carácter de la línea anterior. .NET Framework interpreta el carácter de nueva línea como bajarlo una línea y devolver la posición del siguiente carácter al principio de la siguiente línea. Si no está seguro, siempre puede escribir los caracteres especiales <code>\n</code> y <code>\r</code> juntos.
<code>\x</code>	Los caracteres especiales <code>\x</code> permiten especificar un carácter ASCII usando dos dígitos hexadecimales. Los dos dígitos hexadecimales deben seguir inmediatamente a los caracteres <code>\x</code> y deben corresponder con el valor hexadecimal del carácter ASCII que quiere producir. Por ejemplo, el carácter espacio en ASCII tiene el código de decimal 32. El valor decimal 32 es equivalente al valor hexadecimal 20. Por tanto, una cadena definida como <code>hello\x20there</code> se almacena en memoria con un carácter de espacio entre las palabras <i>hello</i> y <i>there</i> .

Caracteres	Función
\u	Los caracteres especiales \u permiten especificar un carácter Unicode usando exactamente cuatro dígitos hexadecimales. Los cuatro dígitos hexadecimales deben colocarse inmediatamente después de los caracteres \u y deben corresponder al valor hexadecimal del carácter Unicode que quiere producir. Por ejemplo, el carácter espacio en Unicode tiene un código de decimal 32. El valor decimal 32 es equivalente al valor hexadecimal 20. Por tanto, una cadena definida como <code>hello\u0020there</code> se almacena en memoria con un carácter de espacio entre las palabras <i>hello</i> y <i>there</i> . Asegúrese de usar exactamente cuatro dígitos después de los caracteres \u. Si el valor es menor de cuatro dígitos, use ceros para rellenar su valor hasta que llegue a los cuatro dígitos.
\\	Los caracteres especiales \\ permiten especificar un carácter de barra invertida en la posición actual. Una cadena definida como <code>hello\\there</code> se almacena en memoria con un carácter barra invertida entre las palabras <i>hello</i> y <i>there</i> . La razón por la que debe haber dos barras invertidas es simple: el uso de una sola barra invertida podría hacer que el compilador de C# la confundiera con el principio de otro carácter especial. Por ejemplo, suponga que olvida la segunda barra invertida y escribe <code>hello\there</code> en su código. El compilador de C# verá la barra invertida y la 't' en la palabra <i>there</i> y los confundirá con un carácter de tabulación. Esta cadena se almacenaría en memoria con un carácter de tabulación entre las palabras <i>hello</i> y <i>there</i> . (Recuerde que la 't' en <i>there</i> se interpretaría como un carácter de tabulación y no sería parte de la palabra real.)

Desactivación de los caracteres especiales en cadenas

Puede ordenar al compilador de C# que ignore los caracteres especiales en una cadena anteponiendo el signo @ a la cadena:

```
string MyString = @"hello\there";
```

Este código asigna a la variable `MyString` el valor del texto `hello\there`. Como la cadena tiene delante el signo @, se desactiva el modo habitual de interpretar los caracteres `\t` como marcador de tabulación. Esta sintaxis también permite escribir nombres de directorio en cadenas de nombre de archivo de C# sin

usar la doble barra invertida. Por defecto, siempre necesitará usar las dobles barras invertidas:

```
string MyFilename = "C:\\Folder1\\Folder2\\Folder3\\file.txt";
```

Sin embargo, con el prefijo @, puede conseguirlo con una sola barra invertida:

```
string MyFilename = @"C:\Folder1\Folder2\Folder3\file.txt";
```

Cómo acceder a caracteres individuales en la cadena

Puede acceder a caracteres en la cadena como si la cadena fuese una matriz. Conceptualmente, puede imaginarse una cadena como una matriz de caracteres. Puede usar la sintaxis de elemento de matriz entre corchetes para acceder a cualquier carácter de la cadena:

```
char MyCharacter;  
string MyString = "Hello from C#!";
```

```
MyCharacter = MyString[9];
```

Este código coloca el valor `'m'` en la variable `MyCharacter`. El carácter `'m'` está en el elemento 9 de la cadena, si imagina la cadena como una matriz de caracteres. Además, tenga en cuenta que esta matriz de caracteres es de base cero. El primer carácter de la cadena se encuentra en realidad en el elemento 0. El décimo carácter de esta cadena, como ha aprendido, está localizado en el elemento 9.

Declaración de enumeraciones

A diferencia de las variables tratadas hasta el momento, una enumeración no es un tipo en sí misma, sino una forma especial de tipo de valor. Una enumeración se deriva de `System.Enum` y proporciona nombres para valores. El tipo subyacente que representa una enumeración debe ser `byte`, `short`, `int` o `long`. Cada campo de una enumeración es estático y representa una constante.

Para declarar una enumeración, debe usar la palabra clave `enum` seguida del nombre de la enumeración. A continuación debe escribir una llave de apertura seguida por una lista de las cadenas de la enumeración y finalizar con una llave de cierre, como se muestra en el siguiente ejemplo:

```
public enum Pizza  
{  
    Supreme,  
    MeatLovers,  
    CheeseLovers,  
}
```

```
    Vegetable,  
}
```

Este código crea una enumeración llamada `Pizza`. La enumeración `Pizza` contiene cuatro pares diferentes nombre/valor que describen diferentes tipos de pizza, pero no se definen valores. Cuando declara una enumeración, el primer nombre que declara toma el valor 1 y así sucesivamente. Puede invalidar esta funcionalidad asignando un valor a cada nombre, como se muestra a continuación:

```
public enum Pizza  
{  
    Supreme = 2,  
    MeatLovers = 3,  
    CheeseLovers = 4,  
    Vegetable = 5,  
}
```

El valor de cada campo de enumeración ha sido incrementado en 1. Aunque no todo este código es necesario. Asignando a `Supreme` un valor de 2, los siguientes campos siguen la secuencia. Por tanto, puede eliminar las asignaciones a `MeatLovers`, `CheeseLovers`, y `Vegetable`.

Se puede hacer referencia a los enumeradores de una de estas dos formas. Puede programar sobre sus nombres de campo o puede programar sobre sus valores. Por ejemplo, puede asignar el nombre de campo a una variable de cadena con el siguiente código:

```
string MyString = Pizza.Supreme;
```

Quizás quiera hacer referencia al valor de un campo. Puede conseguirlo mediante la conversión de tipos. Por ejemplo, puede recuperar el valor del campo `Supreme` con el siguiente código:

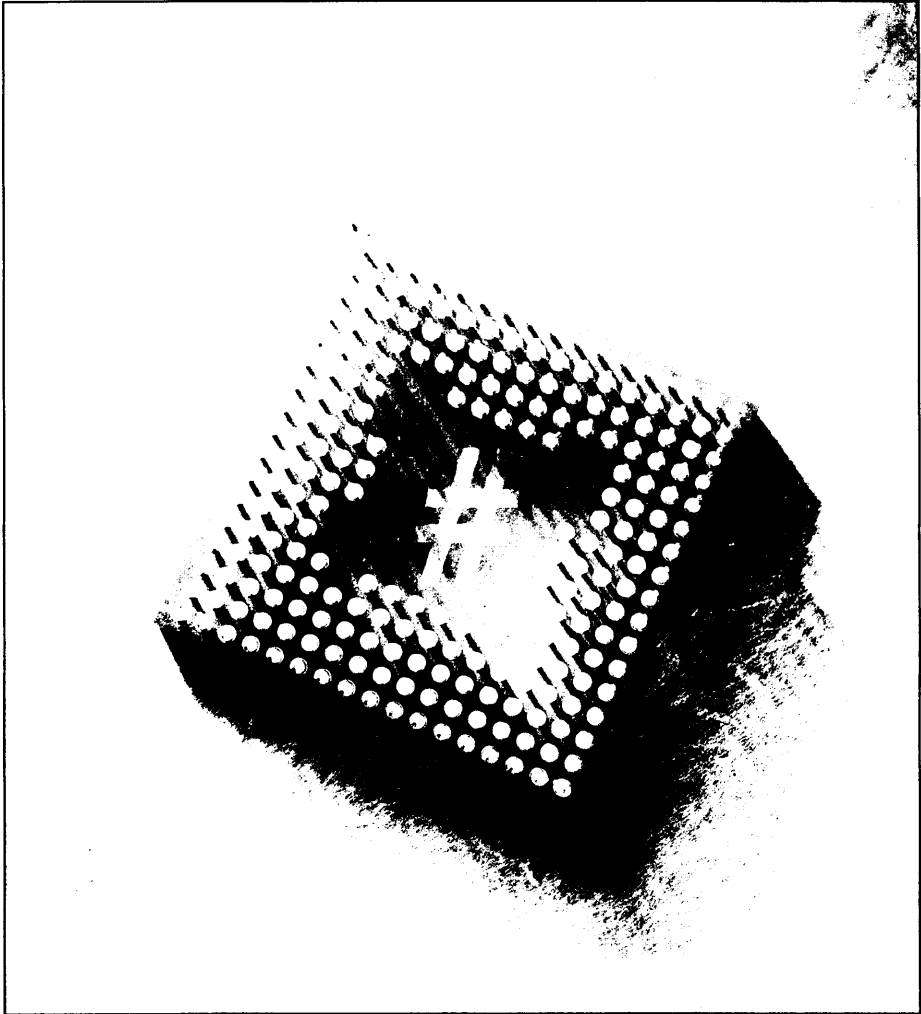
```
int MyInteger = (int)Pizza.Supreme;
```

Resumen

Este capítulo examina las variables y sus tipos. Hay muchas clases de tipos de valor y cada uno tiene sus propias características y requisitos de memoria. Algunos tipos pueden ser convertidos implícitamente a otros tipos, pero otros deben ser convertidos explícitamente usando la sintaxis apropiada.

Las matrices contienen colecciones de variables del mismo tipo. Son útiles cuando se necesita mantener un conjunto de variables del mismo tipo. C# admite matrices unidimensionales y multidimensionales. Las matrices de C# son de base cero: es decir, el número del primer elemento de una matriz es el 0. Las cadenas le ayudan a trabajar con partes de texto en su código. Son conjuntos de caracteres

Unicode. C# permite incrustar caracteres especiales en sus cadenas, pero proporciona el prefijo @ para especificar los casos en los que no necesite que se procesen los caracteres especiales. Se puede acceder a los caracteres de una cadena como si fueran matrices de caracteres.



4

Expresiones

Las *expresiones* son el elemento básico y fundamental de cualquier lenguaje de programación. Mediante el uso de operadores, las expresiones permiten que una operación realice comparaciones simples, asignaciones e incluso operaciones muy complejas que necesitarían millones de años para completarse.

Este capítulo trata del uso de los operadores para realizar funciones matemáticas, asignar valores a variables y realizar comparaciones. Una vez que hayamos comprendido estos elementos básicos, estudiaremos algunas expresiones avanzadas que usan operadores muy específicos del lenguaje C# y que le brindan una ventaja sobre la mayoría de los demás lenguajes de programación. Para cerrar este capítulo, revisaremos las expresiones que usan operadores para manipular la parte más pequeñas de un byte, el bit.

Cómo usar los operadores

Las expresiones pueden escribirse usando variables; valores codificados específicamente, llamados *valores literales* (explicados en la sección "Cómo usar literales", posteriormente en este mismo capítulo); y símbolos llamados *operadores*. C# admite varios operadores, cada uno de los cuales realiza una acción diferente. Las variables o los valores literales que aparecen en una expresión se

llaman *operandos*. Los operadores se aplican a los operandos y el resultado de la operación es otro valor.

C# consta tres tipos de operadores:

- **Operadores unarios.** trabajan con un solo operando. Una expresión con un operando y un operador produce un solo valor.
- **Operadores binarios.** trabajan con dos operandos. Una expresión con dos operandos y un operador produce un solo valor.
- **Operadores ternarios.** trabajan con tres operandos. C# admite sólo un operando ternario.

Uso de expresiones primarias

Las expresiones primarias son la base del código C#. C# define varios tipos diferentes de expresiones primarias:

- Literales
- Identificadores
- Expresiones con paréntesis
- Acceso a miembros
- Expresiones de invocación
- Acceso a elementos
- La palabra clave `this`
- Acceso a bases
- Operadores de incremento y decremento postfijo
- El operador `new`
- El operador `typeof`
- Los operadores `checked` y `unchecked`

Las expresiones primarias permiten definir el orden de las operaciones dentro de una expresión, definir nuevos literales (por ejemplo, valores codificados específicamente) y declarar nuevas variables para la aplicación. En las siguientes secciones se examinarán estas expresiones y cómo usarlas.

Cómo usar los literales

Los literales son valores codificados específicamente que se pueden escribir directamente en el código fuente C#. Hay muchos tipos de literales diferentes.

Para mostrar lo que es un literal, examine la siguiente línea de código en C# que usa el valor literal `Brian`.

```
if (FirstName == "Brian")
```

Aquí se ha codificado el valor de `Brian` para usarlo en una comparación. En lugar de un valor definido por el usuario, es preferible almacenar las cadenas en variables de modo que, si hiciera falta cambiar los valores, sólo habría que cambiarlos en un sitio sin necesidad de buscar cada una de sus ocurrencias en todas las líneas de código.

El siguiente código muestra el mejor método para almacenar y usar una cadena con vistas a compararla:

```
string MyFirstName = "Brian";  
if (FirstName == MyFirstName)
```

Como puede ver, éste es un enfoque mucho más claro para usar un valor literal.

Literales booleanos

C# define dos valores literales booleanos, las palabras clave `True` y `False`:

```
bool MyTrueVariable = true;  
bool MyFalseVariable = false;
```

Ambos valores tienen un tipo de valor `bool`. La palabra clave `True` es el equivalente entero de uno negativo (-1), mientras que el equivalente de `False` es el cero.

Cómo usar los literales enteros en notaciones decimales y hexadecimales

Se pueden escribir literales enteros usando una notación decimal o una notación hexadecimal. De forma parecida a los literales vistos anteriormente, estos literales permiten ordenar el código. Los valores literales pueden ser colocados en la parte superior del listado del código. Si resultara necesario modificar estos en alguna ocasión, resultaría muy sencillo cambiar la ocurrencia del valor.

Los literales decimales integrales se escriben como series de uno o más números usando los caracteres 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9:

```
int MyVariable = 125;
```

Los literales decimales también pueden contener un sufijo de un carácter que especifique el tipo del literal. Si el literal tiene como sufijo una `U` mayúscula o minúscula, el literal decimal se considera de tipo sin signo:

```
uint MyVariable = 125U;
```

El término *sin signo* significa que no se especifica si el número es positivo o negativo. Por tanto, si convierte un valor de 100 negativo (-100) a un valor sin signo, su resultado sería simplemente cien (100).

Si el valor es lo suficientemente pequeño como para poder ser almacenado en un tipo `uint`, el compilador de C# considerará el literal como de tipo `uint`. Si el valor del literal integral es demasiado grande para un tipo `uint`, el compilador de C# considerará el literal como de tipo `ulong`. Los diferentes tipos representan el tamaño de la información que está almacenando. El tipo `uint` puede contener un número comprendido entre 0 y 4.294.967.295; mientras que el valor `ulong` puede contener un valor entre 0 y 8.446.744.073.709.551.615.

Si el literal tiene como sufijo una L mayúscula o minúscula, el literal decimal se considera de tipo `long`:

```
long MyVariable = 125L;
```

Si el valor está dentro del rango de tipo `long`, el compilador de C# considerará el literal como de tipo `long`. Si el valor no está dentro del rango de tipo `long`, el compilador de C# considerará el literal como de tipo `ulong`.

NOTA: Aunque el compilador de C# acepta tanto la l minúscula como la L mayúscula como sufijos, probablemente prefiera usar la L mayúscula. La l minúscula se parece demasiado al número 1 y si otros programadores leen el código, podrían confundir la l con el 1.

Si el literal tiene como sufijos L y U, el literal decimal se considera de tipo `long` sin signo:

```
ulong MyVariable = 125LU;
```

El compilador de C# acepta tanto un sufijo en el que la L aparece delante de la U como un sufijo en el que la U aparece delante de la L. Además, el compilador de C# acepta la combinación de letras en mayúsculas y en minúsculas. Los sufijos LU, Lu, lU, lu, UL, Ul, uL y ul denominan al sufijo `ulong`.

Escribir literales integrales en formato hexadecimal permite escribir un literal usando las letras de la A a la F junto a los números del 0 al 9. Los literales hexadecimales deben tener el prefijo `0X` o `0x`:

```
int MyVariable = 0x7D; // 7D hex = 125 decimal
```

Se pueden usar letras mayúsculas y minúsculas para la notación hexadecimal. También se pueden usar como sufijos los mismos caracteres que estaban disponibles para los literales decimales:

```
long MyVariable = 0x7DL;
```

La decisión de usar un valor hexadecimal queda completamente a discreción del programador. Usar hexadecimales en lugar de otro tipo de literales no supone

ninguna diferencia respecto a usar cualquier otro tipo de número. No obstante, es aconsejable usar valores hexadecimales cuando se está construyendo una aplicación que utilice especificaciones en formato hexadecimal. Por ejemplo, una interfaz para el módem de su ordenador. La referencia del programador para su módem podría especificar los valores de algunas operaciones en formato hexadecimal. En lugar de leer toda la referencia del programador y convertir todos los números al sistema decimal, normalmente sólo tendría que codificar estos números hexadecimales directamente en la aplicación, evitando así cualquier error de conversión.

Cómo usar los literales reales para valores de coma flotante

Los literales reales permiten escribir valores de coma flotante en el código C#. Los literales reales pueden incluir tanto una coma decimal como un exponente.

Las comas decimales pueden aparecer en literales reales y los números pueden aparecer antes y después de la coma decimal. También es posible que un literal real empiece con una coma decimal, lo que es útil cuando se quiere crear un valor mayor de cero, pero menor de uno. Los valores como `2,5` y `,75` son ejemplos de literales reales. C# no impone ningún límite al número de cifras que pueden aparecer antes o después de la coma decimal, mientras el valor del literal quede dentro del rango del tipo deseado. También puede especificar un exponente en sus literales reales. Los exponentes se escriben con una `E` mayúscula o minúscula inmediatamente después de la porción decimal del número. Tras la `E` se incluye uno o más números decimales para indicar el valor del exponente. Esto quiere decir que se puede escribir el valor `750` como un literal real de `7.5e2`. También puede aparecer un signo más o un signo menos entre la `E` y el valor del exponente. Un signo más significa un exponente con valor positivo; un signo menos significa un exponente con valor negativo. El literal real `7.5e+2` define un valor de `750` y el literal real `7.5e-2` define un valor de `.075`. Si no se usa ninguno de los dos signos, el compilador de C# considera que el valor del exponente es positivo. Al igual que los literales decimales, los literales reales también pueden llevar detrás un sufijo para especificar el tipo del literal. Si no se usa un sufijo en el literal real, el compilador de C# considera que el literal es de tipo `double`. Si el literal real tiene como sufijo una `F` mayúscula o minúscula, se considera que el literal decimal es del tipo `float`:

```
float MyVariable = 7.5F;
```

Si el literal real tiene como sufijo una `D` mayúscula o minúscula, se considera que el literal decimal es de tipo `double`:

```
double MyVariable = 7.5D;
```

Si el literal real tiene como sufijo una `M` mayúscula o minúscula, se considera que el literal decimal es de tipo `decimal`:

```
decimal MyVariable = 7.5M;
```

Cómo usar los literales de carácter para asignar valores de carácter

Los literales de carácter permiten escribir valores de carácter en el código C#. Normalmente, los literales de carácter aparecen entre comillas simples:

```
char MyVariable = 'a';
```

También se pueden usar las secuencias de escape vistas en un capítulo anterior para escribir literales de carácter en código C#. Estos literales de carácter deben encerrarse entre comillas simples:

```
char MyVariable = '\t'; // carácter tabulador
```

NOTA: Si quiere utilizar una comilla simple como literal de carácter, deberá anteponerle una barra invertida. Escribir `''` confunde al compilador de C#. Escriba en su lugar `'\''`.

Puede definir valores hexadecimales como literales de carácter usando la secuencia de escape `\x` seguida de uno, dos o tres caracteres hexadecimales:

```
char MyVariable = '\x5C';
```

Cómo usar los literales de cadena para incrustar cadenas

Los literales de cadena permiten incrustar cadenas en el código C#. Los literales de cadena se escriben como se indica en un capítulo anterior, poniendo la cadena entre dobles comillas:

```
string MyVariable = "Hello from C#!";
```

El compilador de C# reutiliza muchos literales de cadena con los mismos contenidos, con lo que conserva espacio en el ejecutable final, como muestra el siguiente código:

```
string String1 = "Hello";  
string String2 = "Hello";
```

Cuando se compila este código, el ejecutable contiene una copia del literal de la cadena `Hello`. Las dos variables de cadena leen el valor de la única copia almacenada en el ejecutable. Esta optimización permite al compilador de C# conservar el uso de memoria del código, ya que almacenar sólo una copia del literal requiere menos memoria que almacenar dos copias del mismo literal.

Cómo usar los literales null

El literal `null` es una palabra clave de C# que permite poner un objeto en un estado nulo o sin uso:

```
object MyObject = null;
```

Uso de identificadores

Los identificadores C# son ejemplos de expresiones simples. Los identificadores tienen un tipo y el tipo se especifica cuando se declara el identificador:

```
int MyVariable = 123;
```

El identificador `MyVariable` se considera una expresión y tiene un tipo `int`. Los identificadores pueden ser definidos en cualquier bloque de código que se encuentre entre llaves, pero su tipo no puede cambiar:

```
public static void Main()
{
    int MyVariable = 123;

    MyVariable = 1; // "MyVariable" todavía es un "int"
    MyVariable = 2; // "MyVariable" todavía es un "int"
}
```

Si intenta redefinir el tipo de un identificador dentro del mismo bloque de código, el compilador de C# generará un mensaje de error:

```
public static void Main()
{
    int MyVariable = 123;
    float MyVariable = 1.25;
}
```

El compilador de C# genera un mensaje de error en la línea que intenta redefinir `MyVariable` como un valor *float*:

```
error CS0128: Ya se ha definido una variable local denominada
'MyVariable' en este ámbito
```

Sin embargo, se puede reutilizar el identificador si aparece en un bloque de código separado:

```
public static void Main()
{
    int MyVariable = 123;
}

public void AnotherFunction()
{
    float MyVariable = 1.25;
}
```

Expresiones entre paréntesis

Como su nombre indica, las expresiones entre paréntesis son expresiones encerradas entre paréntesis. El compilador de C# evalúa la expresión incluida en los

paréntesis y el valor de la expresión entre paréntesis es el resultado de la evaluación. Por ejemplo, el valor de la expresión entre paréntesis (3+2) es 5.

Cómo llamar a métodos con expresiones de acceso a miembros

Cuando se necesita llamar a un método de un objeto, se escribe el nombre del objeto seguido por un punto y por el nombre del método. Cuando el CLR llama al método `Main()` para empezar a ejecutar la aplicación, crea un objeto a partir de su clase y llama a la función `Main()` de ese objeto. Si se escribiese este código en C#, sería algo parecido a lo siguiente:

```
MyClass MyObject;  
  
MyObject = new MyClass();  
MyObject.Main();
```

Los objetos se estudian con más detalle en capítulos posteriores. Lo más importante ahora es darse cuenta de que la instrucción que llama a `Main()` contiene una expresión de acceso a miembros, que contiene un objeto, un punto y una llamada de función.

En capítulos posteriores, verá qué objetos pueden tener datos además de código. Puede acceder a los datos usando la misma sintaxis de expresión de acceso a miembro.

Cómo llamar a métodos con expresiones de invocación

Las expresiones de invocación se usan para hacer una llamada a un método en un objeto. El código usado en el caso del acceso a miembro también muestra una expresión de invocación. El código llama a un método (en este caso `Main()`), que hace que el código invoque al método `Main()` del objeto.

Si se llama a un método desde otro método en el mismo objeto, puede usar el nombre del método en la llamada. No necesita especificar un objeto o un nombre de clase y no es necesaria la sintaxis de acceso a miembro, como muestra el listado 4.1.

Listado 4.1. Expresión de invocación

```
class MyClass  
{  
    public static void Main()  
    {  
  
        MyClass myclass = new MyClass();
```

```

myclass.DoWork();
}

void DoWork()
{
    // haga aquí su trabajo
}
}

```

En este ejemplo, el método `Main()` llama a un método `DoWork()`. Sin embargo, antes hace falta crear una referencia a `myClass` y luego invocar al método `DoWork()`.

El tipo de una expresión de invocación es el tipo que devuelve la función a la que se llama. Si, por ejemplo, el código C# llama a una función que devuelve un tipo `int`, la expresión de invocación que llama a ese método tiene un tipo `int`.

Cómo especificar elementos de matriz con expresiones de acceso a elementos

Las expresiones de acceso a elementos permiten especificar elementos de matriz. El número del elemento de matriz se escribe dentro de corchetes:

```

int [] MyArray;

MyArray = new int [5];
MyArray[0] = 123;

```

En este ejemplo, al elemento cero de la matriz llamada `MyArray` se le asigna el valor de 123.

C# permite que cualquier expresión que produzca un resultado de tipo `int`, `uint`, `long` o `ulong` se utilice como expresión de elemento. C# también permite el uso de cualquier expresión cuyo resultado sea de un tipo que pueda ser convertido implícitamente en tipo `int`, `uint`, `long` o `ulong`. En el código anterior, se usa un literal entero como expresión de elemento. Podría igualmente escribir un tipo de expresión diferente para especificar el elemento, como muestra el listado 4.2.

Listado 4.2. Acceso a elementos

```

class MyClass
{
    public static void Main()
    {
        int [] MyArray;
        MyClass myclass = new MyClass();
        MyArray = new int [5];
        MyArray[myclass.GetArrayIndex()] = 123;
    }
}

```



```

    }

    int GetArrayIndex()
    {
        return 0;
    }
}

```

Este código funciona porque el método `GetArrayIndex()` devuelve un `int` y el resultado de la expresión de invocación es un `int`. Como cualquier expresión cuyo valor sea `int` puede ser usada como expresión de elemento de una matriz. C# permite que este código se ejecute.

El resultado de la expresión de acceso al elemento es el tipo del elemento al que se accede, como se muestra en el siguiente código:

```

int [] MyArray;

MyArray = new int [5];
MyArray[0] = 123;

```

La expresión de acceso al elemento `MyArray[0]` es de tipo `int` porque el elemento al que se accede en la expresión es de tipo `int`.

Cómo acceder a objetos con la palabra clave `this`

C# define una palabra clave `this` que puede usarse para especificar un objeto para un fragmento de código que necesite acceder a ese objeto. La palabra clave `this` se estudia con más detalle en la sección que trata de las clases. El listado 4.3 usa la palabra clave `this`.

Listado 4.3. Acceso mediante palabra clave

```

class MyClass
{
    public static void Main()
    {
        // llame DoWork() en este objeto
        MyClass myclass = new MyClass();
        myclass.DoWork();
    }

    void DoWork()
    {
        MyClass myclass = new MyClass();
        this.DoWork2();
        // haga aquí su trabajo
    }
}

```

```

void DoWork2 ()
{
}
}

```

En este ejemplo, la expresión de acceso `this` tiene el tipo `MyClass` porque la clase `MyClass` contiene el código que incluye la expresión de acceso `this`.

Cómo acceder a objetos con la palabra clave `base`

C# también define la palabra clave `base` para su uso con objetos. En un capítulo posterior aprenderá que puede usar clases como punto de partida para construir nuevas clases. Las clases originales reciben el nombre de *clases base* y las clases construidas a partir de ellas se llaman clases derivadas.

Para ordenar al código C# de la clase derivada que acceda a los datos de la clase base, se usa la palabra clave `base`. El tipo para las expresiones que usan `base` es la clase base de la clase que contiene la palabra clave `base`.

Cómo usar los operadores postfijo de incremento y de decremento

C# permite incrementar o reducir valores numéricos usando símbolos especiales. El operador `++` incrementa el valor y el operador `--` reduce el valor. Se pueden aplicar estos operadores a expresiones de tipo `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` y `ulong`. El listado 4.4 muestra los operadores de incremento y decremento en acción.

Listado 4.4. Operadores de incremento y de decremento

```

class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 125;
        MyInteger++; // el valor ahora es 126
        MyInteger--; // el valor ahora vuelve a ser 125
    }
}

```

El tipo de una expresión que usa operadores postfijos de incremento y de decremento concuerda con el tipo cuyo valor se está incrementando o reduciendo. En el listado 4.4, los operadores de incremento y de decremento tienen tipo `int`.

Creación de nuevos tipos de referencia con el operador new

El operador `new` se usa para crear nuevas instancias de tipos de referencia. Hasta ahora, el operador `new` ha sido usado para crear nuevas matrices y cuando estudie los objetos, aprenderá a usar el operador `new` para crear nuevos objetos.

El operador `new` es considerado una expresión y el tipo de la expresión concuerda con el tipo de la variable creada con la palabra clave `new`.

Cómo devolver información sobre el tipo con el operador typeof

El operador `typeof` es una palabra clave `C#` que devuelve información sobre el tipo de una variable. Se usa como si fuera una función, empleando la palabra clave `typeof` seguida de una expresión:

```
class MyClass
{
    public static void Main()
    {
        System.Console.WriteLine(typeof(int));
    }
}
```

La palabra clave `typeof` devuelve un objeto llamado `System.Type` que describe el tipo de la variable. El tipo de la expresión `typeof` es la clase `System.Type`.

Cómo usar operadores checked y unchecked

Los operadores `checked` y `unchecked` permiten activar o desactivar la verificación en tiempo de ejecución de las operaciones matemáticas. Si se incluye una operación matemática en un operador `checked`, se informará de un error si la operación no tiene sentido. Si se incluye una operación matemática en un operador `unchecked`, se informará de un error incluso si la operación no tiene sentido.

El listado 4.5 muestra un problema de desbordamiento matemático. Se declaran dos variables enteras, `Int1` y `Int2` y una tercera, `Int1PlusInt2`, cuyo valor almacena la suma de las otras dos. Los dos enteros se suman y el resultado se almacena en la tercera variable entera. Entonces el valor de la tercera variable se escribe en la consola.

Listado 4.5. Desborde en operaciones matemáticas

```
class Listing4_5
{
```

```

public static void Main()
{
    int Int1;
    int Int2;
    int Int1PlusInt2;

    Int1 = 2000000000;
    Int2 = 2000000000;
    Int1PlusInt2 = Int1 + Int2;
    System.Console.WriteLine(Int1PlusInt2);
}
}

```

A cada variable entera Int1 y Int2 se le asigna el valor de dos mil millones. Esta operación no supone ningún problema porque las variables enteras pueden almacenar valores por encima de dos mil cien millones. Sin embargo, sumar estos dos integrales y almacenar el resultado en otro integral va a suponer un problema. La suma sería cuatro mil millones, lo que supera el valor límite de un entero, poco más de dos mil cien millones.

Compile el código anterior con la línea de comando habitual:

```
csc Listing4-1.cs
```

Cuando ejecute el archivo Listing4-1.exe, obtendrá un gran número negativo, como se ve en la figura 4.1.

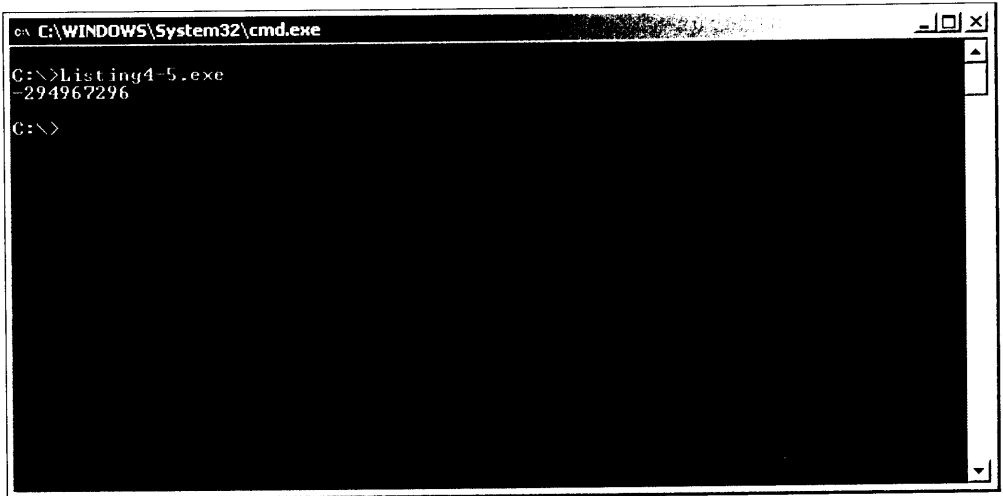


Figura 4.1. Los desbordamientos producen resultados impredecibles.

Se obtiene un resultado negativo debido al modo que tiene C# de procesar los valores demasiado grandes para encajar en las variables destinadas a ellos. C# no puede representar todo el valor de un entero, así que toma el valor propuesto, cuatro mil millones y le resta el valor máximo de un valor de 32 bits (4.294.967.296), extrayendo el resultado a la consola.

Obviamente, el código ha generado un resultado distinto del que queríamos. Si no se da cuenta de este tipo de errores matemáticos, su código puede comportarse de forma impredecible. Para insertar una medida de seguridad en códigos como éste, puede usar el operador `checked`, como aparece en el listado 4.6.

Listado 4.6. Verificación de los desbordamientos de las operaciones matemáticas

```
class Listing4_6
{
    public static void Main()
    {
        int Int1;
        int Int2;
        int Int1PlusInt2;

        Int1 = 2000000000;
        Int2 = 2000000000;
        Int1PlusInt2 = checked(Int1 + Int2);
        System.Console.WriteLine(Int1PlusInt2);
    }
}
```

Si compila y ejecuta el listado 4.6 se escribirá un resultado diferente en la consola:

```
Excepcion no controlada: System.OverflowException: La operacion
aritmética ha provocado un desbordamiento.
at Listing4_1.Main()
```

En lugar de escribir un valor matemático sin sentido en la consola, un mensaje de desbordamiento permite saber que se intentó comprobar la validez del valor de la suma y que la prueba no fue superada. Se informa de una excepción y la aplicación concluye.

La expresión `unchecked()` es la que se usa por defecto. En las expresiones que tienen `unchecked()` no se comprueba la validez de los valores y la aplicación sigue ejecutándose usando los valores no verificados, aunque no tengan sentido.

El comportamiento por defecto es el de no verificar las operaciones. Sin embargo, si quiere que se compruebe si todos los valores de sus operaciones son válidos sin usar el operador `checked()` en el código, puede usar la opción `/checked+` del compilador. Compile el listado 4.1 con la siguiente línea de comando:

```
csc /checked+ Listing4-1.cs
```

Cuando se ejecuta el ejecutable de listing 4-1, se obtiene el mismo mensaje de excepción que se obtuvo con listing 4-2, porque la opción `/checked+` obliga a comprobar la validez de los valores de todas las operaciones matemáticas.

Las expresiones unarias

Las expresiones unarias funcionan sobre un solo operando. C# admite las siguientes expresiones unarias:

- Unario más operador
- Unario menos operador
- Operador de negación lógica
- Operador de complemento bit a bit
- Operador de direccionamiento indirecto
- Operador de direccionamiento
- Operador de incremento y decremento prefijado
- Expresiones de conversión

Las siguientes secciones tratan estas expresiones unarias con detalle.

Cómo devolver valores de operando con el operador unario más

El operador unario más (+) devuelve el valor del operando. Puede pensar en él como el operador *matemático positivo*. C# define el operador unario más para los operandos de tipo `int`, `uint`, `long`, `ulong`, `float`, `double` y `decimal`.

Cómo devolver valores de operando con el operador unario menos

El operador unario menos (-) devuelve el valor del operando. Puede pensar en él como el operador *matemático negativo*. El valor de un operando con un operador unario menos es el equivalente negativo del operando. C# define el operador unario menos para los operandos de tipo `int`, `long`, `float`, `double` y `decimal`.

Expresiones negativas booleanas con el operador de negación lógica

El operador de negación lógica niega el valor de una expresión booleana. El operador cambia el valor `True` a `False` y el valor `False` a `True`. Se usa el signo de exclamación para escribir un operador de negación lógica en el código

C#. El operador se coloca antes de la expresión booleana que quiera negar, como se ilustra en el listado 4.7.

Listado 4.7. Operador de negación lógica

```
class MyClass
{
    public static void Main()
    {
        bool MyBoolean;

        MyBoolean = true;
        MyBoolean = !MyBoolean; // "MyBoolean" ahora es false
    }
}
```

El operador de complemento bit a bit

C# permite aplicar una operación de complemento bit a bit a expresiones `int`, `uint`, `long` y `ulong`. Las operaciones de complemento bit a bit consideran al valor como si fueran un binario y dan la vuelta a todos los bits.

Los bits que tenían un valor 1 se vuelven 0 y los bits que tenían un valor 0 se vuelven 1.

Los operadores de complemento bit a bit se especifican colocando el carácter virgulilla (~) antes de la expresión que debería ser complementada bit a bit, como se ve en el listado 4.8.

Listado 4.8. Operador de complemento bit a bit

```
class MyClass
{
    public static void Main()
    {
        int Int1;

        Int1 = 123;
        Int1 = ~Int1;
    }
}
```

Cómo prefijar operadores de incremento y decremento

Los operadores postfijos `++` y `--` pueden ser usados en uno de los dos modos. Ya ha visto las versiones postfijas de los operadores, que aparece después de la expresión. Las versiones prefijas aparecen antes de la expresión, como se ve en el listado 4.9.

Listado 4.9. Operadores de incremento y decremento prefijados

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 125;
        ++MyInteger; // el valor ahora es 126
        --MyInteger; // el valor ahora vuelve a ser 125
    }
}
```

El tipo de una expresión que usa los operadores postfijos de incremento y decremento concuerda con el tipo cuyo valor se incrementa o reduce. Tenga en cuenta la sutil diferencia entre estos operadores prefijos y los operadores postfijos que se vieron con anterioridad: con los operadores prefijos, el valor se cambia después de que se evalúe la expresión. El listado 4.10 ilustra esta diferencia.

Listado 4.10. Diferencias entre operadores postfijos y prefijos

```
class Listing4_10
{
    public static void Main()
    {
        int Int1;

        Int1 = 123;
        System.Console.WriteLine(Int1++);
        System.Console.WriteLine(++Int1);
    }
}
```

Compile y ejecute el listado 4.3. El resultado de esta aplicación aparece en la figura 4.2. La primera instrucción del listado 4.10 usa el operador postfijo de incremento, lo que significa que el valor se incrementa después de que se ejecute la instrucción. La aplicación escribe el valor actual, 123, en la consola y luego incrementa el valor a 124. La segunda instrucción usa el operador de incremento postfijo, lo que significa que el valor se incrementa antes de que se ejecute la instrucción. La aplicación primero incrementa el valor actual a 125 y luego escribe el valor actual en la consola.

Los operadores aritméticos

Los operadores aritméticos permiten realizar cálculos en el código C#. Las expresiones que usan operadores aritméticos son expresiones binarias porque se necesitan dos operandos para realizar una operación matemática.

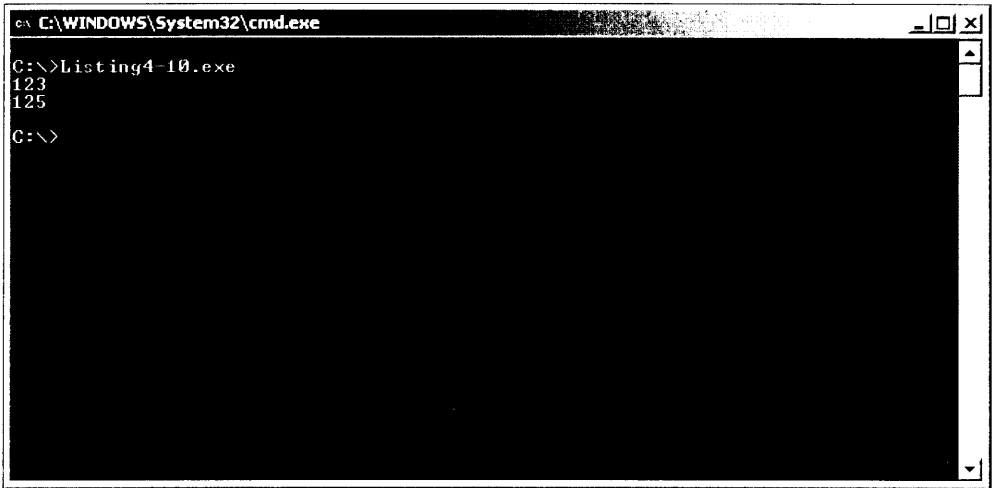


Figura 4.2. Uso de operadores prefijos y postfijos

Cómo asignar nuevos valores con el operador de asignación

El operador de asignación asigna un nuevo valor a una variable. El signo igual se usa como operador de asignación:

```
MyInteger = 3;
```

Se establece el valor de `MyInteger` en 3 y se pierde el valor anterior de `MyVariable`.

Los operadores de asignación compuesta permiten usar el operador de asignación más de una vez en una instrucción:

```
MyInteger = MyOtherInteger = 3;
```

El valor de la expresión a la derecha se usa como el nuevo valor para las variables. En este ejemplo, se asigna 3 a `MyInteger` y a `MyOtherInteger`.

Uso del operador multiplicación

El valor de una expresión que usa el operador de multiplicación es el producto de los valores de los dos operadores. El carácter asterisco se usa como operador de multiplicación, como se ve en el listado 4.11.

Listado 4.11. Operador multiplicación

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

```

    {
        int MyInteger;

        MyInteger = 3 * 6; // MyInteger sera 18
    }
}

```

Si se está multiplicando un valor por una variable y colocando el resultado en la misma variable, se puede escribir una instrucción abreviada para realizar la multiplicación. Al introducir un asterisco seguido por un signo igual se multiplica un valor por una variable y se actualiza el valor de la variable con el resultado:

```
MyInteger *= 3;
```

Esta instrucción es la abreviatura de la siguiente:

```
MyInteger = MyInteger * 3;
```

Uso del operador división

El valor de una expresión que usa el operador de división es el producto de los valores de los operadores. La barra inclinada es el carácter que se usa como operador de división, como se ve en el listado 4.12.

Listado 4.12. Operador división (Ejemplo 1)

```

class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 6 / 3; // MyInteger sera 2
    }
}

```

Si la operación de división da como resultado un resto, el resultado de la operación será sólo el cociente (véase el listado 4.13).

Listado 4.13. Operador división (Ejemplo 2)

```

class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 7 / 3;
    }
}

```

Cuando se ejecuta este código, la variable `MyInteger` tiene un valor de 2, porque si se divide 7 entre 3 queda un cociente de 2 y un resto de 1.

Si se divide un valor entre una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada para realizar la división. Escribiendo una barra inclinada seguida por un signo igual se divide un valor entre una variable y se actualiza el valor de la variable con el resultado:

```
MyInteger /= 3;
```

La instrucción anterior es una abreviatura de la siguiente:

```
MyInteger = MyInteger / 3;
```

Uso del operador resto

El valor de una expresión que usa el operador de resto es el resto de una operación de división. El carácter tanto por ciento se usa como el operador de división (véase el listado 4.14).

Listado 4.14. Operador resto

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;
        MyInteger = 7 % 3;
    }
}
```

Cuando se ejecuta este código, la variable `MyInteger` tiene el valor de 1, porque si se divide 7 entre 3 queda un cociente de 2 y un resto de 1.

Si se está calculando un resto usando una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada para realizar la operación de resto. Si se escribe un signo de tanto por ciento seguido del signo igual se calculará el resto de una variable y se actualizará el valor de la variable con el resultado:

```
MyInteger %= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger % 3;
```

Uso del operador suma

El valor de una expresión que usa el operador de suma es la suma de los valores de los dos operadores. El carácter suma se usa como el operador de multiplicación (véase el listado 4.15).

Listado 4.15. Operador suma

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 3 + 6; // MyInteger será 9
    }
}
```

Si se está sumando un valor a una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada que realice la suma. Al escribir un signo más seguido de un signo igual se añade un valor a una variable y se actualiza el valor de la variable con el resultado:

```
MyInteger += 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger + 3;
```

El operador de suma tiene un significado especial cuando los dos operandos son cadenas. La suma de dos cadenas une la primera cadena a la segunda:

```
string CombinedString = "Hello from " + "C#";
```

El valor de `CombinedString` es `Hello from C#` cuando se ejecuta este código.

Uso del operador resta

El valor de una expresión que usa el operador de resta es la diferencia de los valores de los dos operadores. El carácter guión se usa como el operador de resta (véase el listado 4.16).

Listado 4.16. Operador resta

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 7 - 3; // MyInteger será 4
    }
}
```

Si se está restando un valor a una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada que realice la resta. Al es-

cribir un signo menos seguido de un signo igual se resta un valor de una variable y se actualiza el valor de la variable con el resultado:

```
MyInteger -= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger - 3;
```

Los operadores de desplazamiento

Los operadores de desplazamiento permiten mover bits de lugar en un valor de su código C#. Las expresiones que usan los operadores de desplazamiento son expresiones binarias porque se necesitan dos operandos para realizar una operación de desplazamiento.

Cómo mover bits con el operador de desplazamiento a la izquierda

El valor de una expresión que usa el operador de desplazamiento a la izquierda se mueve a la izquierda la cantidad de bits especificados. Se usan dos caracteres menor que (<<) como operadores de desplazamiento a la izquierda (véase el listado 4.17).

Listado 4.17. Operador de desplazamiento a la izquierda

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 6 << 3;
    }
}
```

Cuando se ejecuta este código, la variable `MyInteger` tiene un valor de 48, porque el valor original, 6, es considerado un número binario con un valor binario de 00000110. Cada bit en el valor original se desplaza tres lugares, que es el valor que aparece después del operador de desplazamiento a la izquierda y se colocan ceros en los bits de orden inferior. Al cambiar cada bit tres lugares da como resultado un valor binario de 00110000 o 48 en el sistema decimal.

Se pueden aplicar desplazamientos a la izquierda a los valores de las expresiones de tipo `int`, `uint`, `long` y `ulong`. También pueden desplazarse a la izquierda otras expresiones que pueden ser convertidas a uno de estos tipos. Las

expresiones de tipo `int` y `uint` pueden desplazarse hasta 32 bits de una vez. Las expresiones de tipo `long` y `ulong` pueden ser desplazadas hasta 64 bits de una vez.

Si se está calculando una operación de desplazamiento a la izquierda de un valor y una variable, y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada que realice esta operación. Al escribir dos signos menor que (`<<`) seguidos por un signo igual se calcula la operación de desplazamiento a la izquierda y se actualiza el valor de la variable con el resultado:

```
MyInteger <<= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger << 3;
```

Cómo mover bits con el operador de desplazamiento a la derecha

El valor de una expresión que usa el operador de desplazamiento a la derecha se mueve a la derecha la cantidad de bits especificados. Se usan dos caracteres mayor que (`>>`) como operadores de desplazamiento a la derecha (véase el listado 4.18).

Listado 4.18. Operador de desplazamiento a la derecha

```
class MyClass
{
    public static void Main()
    {
        int MyInteger;

        MyInteger = 48 >> 3;
    }
}
```

Cuando se ejecuta este código, la variable `MyInteger` tiene un valor de 6, porque el valor original, 48, es considerado un número binario con un valor binario de 00110000. Cada bit en el valor original se desplaza tres lugares, que es el valor que aparece después del operador de desplazamiento a la derecha y se colocan ceros en los bits de orden superior. El cambiar cada bit tres lugares da como resultado un valor binario de 00000110 o 6 decimal.

Se pueden aplicar desplazamientos a la derecha a los valores de las expresiones de tipo `int`, `uint`, `long` y `ulong`. También pueden desplazarse a la derecha otras expresiones que pueden ser convertidas a uno de estos tipos. Las expresiones de tipo `int` y `uint` pueden desplazarse hasta 32 bits de una vez. Las expresiones de tipo `long` y `ulong` pueden ser desplazadas hasta 64 bits de una vez.

Si está calculando una operación de desplazamiento a la derecha de un valor y una variable y colocando el resultado en la misma variable, puede escribir una instrucción abreviada que realice esta operación. Escribir dos signos mayor que seguidos por un signo igual calcula la operación de desplazamiento a la derecha y actualiza el valor de la variable con el resultado:

```
MyInteger >>= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger >> 3;
```

Cómo comparar expresiones con operadores relacionales

Los operadores relacionales permiten comparar dos expresiones y obtener un valor booleano que especifica la relación entre las dos expresiones. Las expresiones que usan operadores relacionales son expresiones binarias porque se necesitan dos operandos para realizar una operación relacional.

Cómo comprobar la igualdad con el operador de igualdad

El operador de igualdad se usa para comprobar la igualdad entre los valores de dos expresiones. Si las expresiones tienen el mismo valor, el operador de igualdad devuelve `True`. Si tienen valores diferentes, el operador de igualdad devuelve `False`. Como operador de igualdad se usan dos signos igual:

```
MyInteger == 123;
```

Si el valor de la variable `MyInteger` es 123, el operador de igualdad devuelve `True`. Si tiene otro valor, el operador de igualdad devuelve `False`.

El operador de igualdad tiene un significado especial cuando los dos operandos son cadenas. Al comparar dos cadenas se comparan los contenidos de las cadenas. Dos cadenas se consideran iguales si tienen la misma longitud y los mismos caracteres en cada posición de la cadena.

Cómo comprobar la desigualdad con el operador de desigualdad

El operador de desigualdad se usa para comprobar la desigualdad entre los valores de dos expresiones. Si las expresiones tienen diferentes valores, el operador de desigualdad devuelve `True`. Si tienen el mismo valor, el operador de

desigualdad devuelve `False`. Como operador de desigualdad se usa un signo de exclamación seguido por un signo igual:

```
MyInteger != 123;
```

Si el valor de la variable `MyInteger` es 123, el operador de desigualdad devuelve `False`. Si tiene otro valor, el operador de desigualdad devuelve `True`.

El operador de desigualdad tiene un significado especial cuando los dos operandos son cadenas. Al comparar dos cadenas se comparan los contenidos de las cadenas.

Dos cadenas se consideran desiguales si tienen diferentes longitudes o diferentes caracteres en, al menos, una posición de la cadena.

Cómo comprobar valores con el operador menor que

El operador menor que se usa para comprobar los valores de dos expresiones y ver si un valor es menor que el otro. Si la primera expresión tiene un valor menor que el de la segunda expresión, el operador menor que devuelve `True`. Si la primera expresión tiene un valor mayor o igual que el de la segunda expresión, el operador menor que devuelve `False`. El operador menor que se representa mediante un signo menor que (`<`):

```
MyInteger < 123;
```

Si el valor de la variable `MyInteger` es menor de 123, el operador menor que devuelve `True`. Si tiene un valor mayor que o igual a 123, el operador menor que devuelve `False`.

Cómo comprobar valores con el operador mayor que

El operador mayor que se usa para comprobar los valores de dos expresiones y ver si un valor es mayor que el otro. Si la primera expresión tiene un valor mayor que el de la segunda expresión, el operador mayor que devuelve `True`. Si la primera expresión tiene un valor menor o igual que el de la segunda expresión, el operador mayor que devuelve `False`. El operador mayor que se representa mediante un signo mayor que (`>`):

```
MyInteger > 123;
```

Si el valor de la variable `MyInteger` es mayor de 123, el operador mayor que devuelve `True`. Si tiene un valor menor que o igual a 123, el operador menor que devuelve `False`.

Cómo comprobar valores con el operador menor o igual que

El operador menor o igual se usa para comprobar los valores de dos expresiones y ver si un valor es menor o igual que el otro. Si la primera expresión tiene un valor menor o igual que el de la segunda expresión, el operador menor o igual que devuelve `True`.

Si la primera expresión tiene un valor mayor que el de la segunda expresión, el operador menor o igual que devuelve `False`.

Como operador menor o igual que se usa un signo menor que seguido de un signo igual (`<=`):

```
MyInteger <= 123;
```

Si el valor de la variable `MyInteger` es menor o igual a 123, el operador menor o igual que devuelve `True`. Si tiene un valor mayor de 123, el operador menor o igual que devuelve `False`.

Cómo comprobar valores con el operador mayor o igual que

El operador mayor o igual que se usa para comprobar los valores de dos expresiones y ver si un valor es mayor o igual que el otro. Si la primera expresión tiene un valor mayor o igual que el de la segunda expresión, el operador mayor o igual que devuelve `True`.

Si la primera expresión tiene un valor menor que el de la segunda expresión, el operador mayor o igual que devuelve `False`.

Como operador mayor o igual que se usa un signo mayor que seguido de un signo igual (`>=`):

```
MyInteger >= 123;
```

Si el valor de la variable `MyInteger` es mayor o igual a 123, el operador mayor o igual que devuelve `True`. Si tiene un valor menor de 123, el operador mayor o igual que devuelve `False`.

Operadores lógicos enteros

Los operadores lógicos enteros permiten realizar operaciones aritméticas booleanas sobre dos valores numéricos. Las expresiones que usan operadores lógicos enteros son expresiones binarias porque se necesitan dos operandos para realizar una operación lógica.

Cómo calcular valores booleanos con el operador AND

El operador AND se usa para calcular el valor AND booleano de dos expresiones. Como operador AND se usa el símbolo de unión (&):

```
MyInteger = 6 & 3;
```

El valor de `MyInteger` es 2. Recuerde que un bit en una operación AND es 1 sólo si los dos bits operandos de la misma posición son 1. El valor 6 en binario es 110 y el valor 3 en binario es 011. Si se realiza un AND booleano de 110 y 011 se obtiene como resultado un valor booleano de 010 o 2 en el sistema decimal.

Si se está calculando una operación AND sobre un valor y una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada que realice la operación AND. Si se escribe un signo de unión (&) seguido de un signo igual se calcula la operación AND sobre una variable y un valor, y se actualiza el valor de la variable con el resultado:

```
MyInteger &= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger & 3;
```

Cómo calcular valores booleanos con el operador exclusivo OR

El operador exclusivo OR se usa para calcular el valor booleano exclusivo OR de dos expresiones. Como operador OR exclusivo se usa el signo de intercalación (^):

```
MyInteger = 6 ^ 3;
```

El valor de `MyInteger` es 5. Recuerde que un bit en una operación exclusiva OR es 1 sólo si uno de los dos bits operando en la misma posición es 1. El valor de 6 en binario es 110 y el valor de 3 en binario es 011. Si realizamos un booleano OR exclusivo entre 110 y 011 obtenemos como resultado un valor booleano de 101 o 5 en el sistema decimal.

Si se está calculando una operación OR exclusiva sobre un valor y una variable y se coloca el resultado en la misma variable, se puede escribir una instrucción abreviada que realice la operación OR exclusiva. Si se escribe un signo de intercalación (^) seguido de un signo igual se calcula la operación OR exclusiva sobre una variable y un valor, y se actualiza el valor de la variable con el resultado:

```
MyInteger ^= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger ^ 3;
```

Cómo calcular valores booleanos con el operador OR

El operador OR se usa para calcular el valor booleano OR de dos expresiones. Como operador OR se usa el carácter barra vertical (|):

```
MyInteger = 6 | 3;
```

El valor de `MyInteger` es 7. Recuerde que un bit en una operación OR es 1 sólo si uno de los dos bits operandos de la misma posición es 1. El valor 6 en binario es 110 y el valor 3 en binario es 011. Si se realiza un booleano OR entre 110 y 011 se obtiene como resultado un valor de 111 o 7 en decimal.

Si se está calculando una operación OR sobre un valor y una variable, y se coloca el resultado en la misma variable, puede escribir una instrucción abreviada que realice la operación OR. Si se escribe una barra vertical (|) seguido de un signo igual se calcula la operación OR sobre una variable y un valor, y se actualiza el valor de la variable con el resultado:

```
MyInteger |= 3;
```

La instrucción anterior es la abreviatura de la siguiente:

```
MyInteger = MyInteger | 3;
```

Operadores condicionales lógicos

Los operadores condicionales lógicos son los equivalentes condicionales de los operadores lógicos enteros. Las expresiones que usan los operadores condicionales lógicos son expresiones binarias porque se necesitan dos operandos para realizar una operación condicional lógica.

Comparación de valores booleanos con el operador AND condicional

El operador AND condicional se usa para comparar dos expresiones booleanas. El resultado de la operación es `True` si ambos operandos devuelven `True` y `False` si uno de los dos operandos devuelve `False`. Como operador AND condicional se usan dos símbolos de unión:

```
MyBoolean = true && false;
```

El valor de `MyBoolean` es `False` porque uno de los dos operandos devuelve `False`.

Comparación de valores booleanos con el operador OR condicional

El operador `OR` condicional se usa para comparar dos expresiones booleanas. El resultado de la operación es `True` si uno de los dos operandos devuelve `True` y `False` si los dos operandos devuelven `False`. Como operador `OR` condicional se usan dos barras verticales:

```
MyBoolean = true || false;
```

El valor de `MyBoolean` es `True` porque uno de los dos operandos devuelve `True`.

Comparación de valores booleanos con el operador lógico condicional

El operador lógico condicional evalúa una expresión booleana. El resultado de la expresión tiene un valor si la expresión de entrada devuelve `True` y otro si la expresión de entrada devuelve `False`. Las expresiones que usan operadores condicionales son expresiones ternarias porque se necesitan tres operandos para realizar una operación lógica condicional. El operador condicional es la única expresión ternaria admitida por el lenguaje `C#`.

Escribir un operador condicional implica escribir la expresión de entrada seguida por un signo de interrogación. El valor `True` aparece después, seguido de dos puntos y a continuación seguido por el valor `False`:

```
MyInteger = (MyVariable == 123) ? 3 : 5;
```

Puede interpretar esta instrucción como "Compara el valor de `MyVariable` con 123. Si esa expresión devuelve `True`, haz que el valor de `MyInteger` sea 3. Si esa expresión devuelve `False`, haz que el valor de `MyInteger` sea 5".

El orden de las operaciones

`C#` permite colocar varios operadores en una sola instrucción:

```
MyVariable = 3 * 2 + 1;
```

¿Cuál es el valor de `MyVariable` aquí? Si `C#` aplica la multiplicación en primer lugar, leerá la instrucción como "multiplica 3 por dos y luego añade 1".

que da como resultado un valor de 7. Si C# aplica la suma en primer lugar, leerá la instrucción como "suma 2 y 1 y luego multiplícalo por 3", que da como resultado un valor de 9.

C# combina los operadores en grupos y aplica un orden de prioridad a cada grupo. Este orden de prioridad especifica qué operadores se evalúan antes que otros. La lista con el orden de prioridad de C# es la siguiente, ordenados de mayor prioridad a menor:

- Expresiones primarias
- Operadores unarios + - ! ~ ++ --
- Operadores multiplicativos * / %
- Operadores aditivos + -
- Operadores de desplazamiento << >>
- Operadores relacionales < > <= >=
- Operadores de igualdad == !=
- AND lógico
- OR lógico exclusivo
- OR lógico
- AND condicional
- OR condicional
- Ternario condicional
- Operadores de asignación

Repase la siguiente instrucción:

```
MyVariable = 3 * 2 + 1
```

C# da a `MyVariable` un valor de 7 porque la prioridad del operador de multiplicación es superior a la del operador de suma. Esto significa que el operador de multiplicación se evalúa primero y en segundo lugar el operador suma.

Se puede invalidar el orden de prioridad con paréntesis. Las expresiones entre paréntesis se evalúan antes de que se apliquen las reglas de prioridad de operadores:

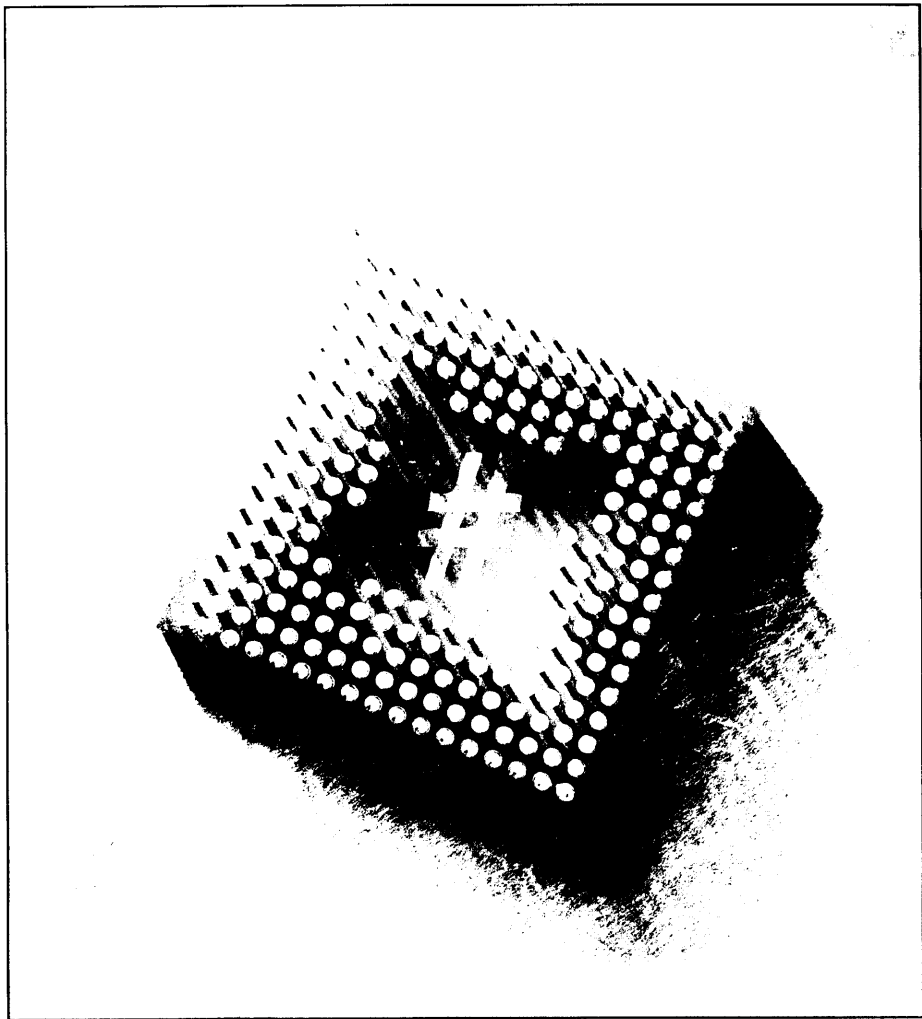
```
MyVariable = 3 * (2 + 1)
```

En este caso, C# da a `MyVariable` un valor de 9, porque la expresión de suma está entre paréntesis, obligando a que se evalúe antes que la operación de multiplicación.

Resumen

C# define muchos operadores para ayudarle a evaluar expresiones y a calcular nuevos valores a partir de esas operaciones. Este lenguaje permite escribir expresiones que realizan funciones matemáticas y booleanas, y compara dos expresiones y obtiene un resultado booleano de esa comparación.

En este capítulo, se presentan los operadores de C# y se aprende a usar estos operadores en expresiones con literales y variables. También se han revisado las expresiones de operador y la prioridad al usar estos operadores en expresiones. Cuando examinemos las clases en un capítulo posterior, descubrirá que sus clases pueden redefinir algunos de estos operadores. A esto se le llama sobrecarga de operadores y le permite redefinir la forma en que los operadores calculan los resultados.



5 Cómo controlar el flujo del código

El comportamiento del código C# a menudo depende de las condiciones que se determinan en tiempo de ejecución. Quizás quiera escribir una aplicación que salude a sus usuarios con un mensaje de "Buenos días" si la hora en ese momento es inferior a las 12:00 P.M., por ejemplo; o "Buenas tardes" si la hora en ese momento está entre las 12:00 P.M. y las 6:00 P.M.

Comportamientos como éste necesitan que el código C# examine valores en tiempo de ejecución y realice una acción basada en dichos valores. C# admite varias construcciones de código que le permiten examinar variables y realizar una o varias acciones basadas en dichas variables. En este capítulo se examinan las instrucciones de flujo de control de C# que actuarán como el cerebro de las aplicaciones que escriba.

Instrucciones de C#

Una *instrucción* es una expresión válida de C# que define una acción realizada por el código. Las instrucciones pueden examinar valores de variables, asignar nuevos valores a una variable, llamar a métodos, realizar una operación, crear objetos o realizar alguna otra acción.

La instrucción más corta posible en C# es la *instrucción vacía*. Ésta consiste en sólo el punto y coma:

```
;
```

Se puede usar la instrucción vacía para decir, "No hagas nada aquí". Esto podría no parecer muy útil, pero tiene su función.

NOTA: Todas las instrucciones de C# terminan en un punto y coma.

Las instrucciones se agrupan en *listas de instrucciones* que se componen de una o más instrucciones escritas en secuencia:

```
int MyVariable;  
  
MyVariable = 123;  
MyVariable += 234;
```

Por lo general, las instrucciones se escriben en su propia línea. Sin embargo, C# no exige esta disposición. C# ignora cualquier espacio en blanco entre instrucciones y acepta cualquier disposición siempre que cada instrucción este separada por un punto y coma:

```
int MyVariable;  
  
MyVariable = 123; MyVariable += 234;
```

Las listas de instrucciones se encierran entre llaves. Una lista de instrucciones entre llaves recibe el nombre de *bloque de instrucciones*. Casi siempre usará bloques de instrucciones para escribir el código de la función. Toda la lista de instrucciones de la función se coloca en un bloque de instrucciones. Es perfectamente posible usar sólo una instrucción en un bloque de instrucciones:

```
public static void Main()  
{  
    System.Console.WriteLine("Hello!");  
}
```

C# no impone ningún límite al número de instrucciones que se pueden colocar en un bloque de instrucciones.

Instrucciones para declarar variables locales

Las *instrucciones de declaración* declaran variables locales en el código. Ya hemos visto varios ejemplos de este tipo de instrucciones. Las instrucciones de declaración especifican un tipo y un nombre para un variable local:

```
int MyVariable;
```

También se puede inicializar la variable cuando se declara usando un signo igual y asignando un valor a la variable:

```
int MyVariable = 123;
```

C# permite enumerar varias variables en la misma instrucción. Para separar los nombres de las variables se usan comas.

```
int MyFirstVariable, MySecondVariable;
```

Cada variable de la instrucción tiene el tipo especificado. En el ejemplo anterior, `MyFirstVariable` y `MySecondVariable` son de tipo `int`.

Las *declaraciones de constantes* definen una variable cuyo valor no puede cambiar durante la ejecución del código. Las declaraciones de constantes usan la palabra clave de C# `const` y deben asignar un valor a la variable cuando se declara dicha variable:

```
const int MyVariable = 123;
```

Las declaraciones de constantes permiten una mejor legibilidad y administración del código. Se pueden tener valores constantes en el código y al asignarles nombres se consigue que el código resulte más legible que si usara su valor. Además, si se usan valores por todo el código y luego se necesita cambiarlos, ésta será una tarea muy pesada. Si se usa una constante, sólo hará falta cambiar una línea de código.

Por ejemplo, suponga que está escribiendo un código para una aplicación que realiza medidas geométricas. Uno de los valores con los que querrá trabajar es π , la relación entre la circunferencia de un círculo y su diámetro. Sin una declaración de constante, tendría que escribir un código de la siguiente forma:

```
Area = 3.14159 * Radius * Radius;
```

Al usar una constante se logra que el código sea un poco más sencillo de entender:

```
const double Pi = 3.14159;
```

```
Area = Pi * Radius * Radius;
```

Esto es especialmente útil si usa muchas veces en el código el valor de π .

Cómo usar instrucciones de selección para seleccionar la ruta del código

Las instrucciones de selección seleccionan una de las muchas rutas posibles para que se ejecute el código. La ruta de código seleccionado se basa en el valor de una expresión.

La instrucción if

La instrucción `if` trabaja con una expresión que devuelve un valor booleano. Si la expresión booleana resulta ser `true`, la instrucción incrustada en la instrucción `if` se ejecuta. Si la expresión booleana resulta ser `false`, la instrucción incrustada en la instrucción `if` no se ejecuta:

```
if(MyVariable == 123)
    System.Console.WriteLine("MyVariable's value is 123.");
```

La instrucción booleana se escribe entre paréntesis. La instrucción incrustada sigue a los paréntesis. Se usa un punto y coma para cerrar la instrucción incrustada, pero no la expresión booleana.

NOTA: Cuando se usa la instrucción `if` para comprobar una igualdad, siempre se deben usar dos signos igual. Dos signos igual hacen una comprobación de igualdad, mientras que un signo igual realiza una asignación. Si se usa accidentalmente un signo igual dentro de una instrucción `if`, ésta siempre devolverá un valor `true`.

En el anterior ejemplo, el valor de `MyVariable` se compara con el valor literal `123`. Si el valor es igual a `123`, la expresión devuelve `true` y se escribe el mensaje `MyVariable's value is 123.` en la consola. Si el valor no es igual a `123`, la expresión devuelve `false` y no se escribe nada.

La instrucción `if` puede ir seguida de una cláusula `else`. La palabra clave `else` va seguida de una instrucción incrustada que se ejecuta si la expresión booleana usada en la cláusula `if` devuelve `false`:

```
if(MyVariable == 123)
    System.Console.WriteLine("MyVariable's value is 123.");
else
    System.Console.WriteLine("MyVariable's value is not 123.");
```

En el anterior ejemplo, el valor de `MyVariable` se compara con el valor literal `123`. Si el valor es igual a `123`, la expresión devuelve `true` y se escribe el mensaje `MyVariable's value is 123.` en la consola. Si el valor no es igual a `123`, la expresión devuelve `false` y se escribe el mensaje `MyVariable's value is not 123.` en la consola.

La cláusula `else` puede ir seguida por su propia cláusula `if`:

```
if(MyVariable == 123)
    System.Console.WriteLine("MyVariable's value is 123.");
else if(MyVariable == 124)
    System.Console.WriteLine("MyVariable's value is 124.");
else
    System.Console.WriteLine("MyVariable's value is not 123.");
```

Las cláusulas `if` y `else` permiten asociar una instrucción a la cláusula. Por lo general, C# permite asociar sólo una instrucción a la cláusula, como se ve en el siguiente código:

```
if(MyVariable == 123)
    System.Console.WriteLine("MyVariable's value is 123.");
System.Console.WriteLine("This always prints.");
```

La instrucción que escribe `This always prints.` en la consola siempre se ejecuta. No pertenece a la cláusula `if` y se ejecuta independientemente de si el valor de `MyVariable` es 123. La única instrucción que depende de la comparación de `MyVariable` con 123 es la instrucción que escribe `MyVariable's value is 123.` en la consola. Si se quiere asociar varias instrucciones con una cláusula `if`, se debe usar un bloque de instrucciones:

```
if(MyVariable == 123)
{
    System.Console.WriteLine("MyVariable's value is 123.");
    System.Console.WriteLine("This prints if MyVariable ==
123.");
}
```

También se pueden usar bloques de instrucciones en las cláusulas `else`:

```
if(MyVariable == 123)
{
    System.Console.WriteLine("MyVariable's value is 123.");
    System.Console.WriteLine("This prints if MyVariable ==
123.");
}
else
{
    System.Console.WriteLine("MyVariable's value is not 123.");
    System.Console.WriteLine("This prints if MyVariable !=
123.");
}
```

Como los bloques de instrucciones pueden contener una sola instrucción, el siguiente código también es válido:

```
if(MyVariable == 123)
{
    System.Console.WriteLine("MyVariable's value is 123.");
}
```

La instrucción `switch`

La instrucción `switch` evalúa una expresión y compara el valor de esa expresión con varios casos. Cada caso se asocia con una lista de instrucciones, que recibe el nombre de *sección de switch*. C# ejecuta la lista de instrucción asociada con la sección de *switch* que concuerde con el valor de la expresión.

La expresión usada como controlador de la instrucción `switch` se encierra entre los paréntesis que siguen a la palabra clave `switch`. La expresión va seguida por llaves y las secciones de *switch* están entre las llaves.

```
switch(MyVariable)
{
    // aqui se colocan las secciones de switch
}
```

La expresión usada en la instrucción `switch` debe evaluar uno de los siguientes tipos:

- `sbyte`
- `byte`
- `short`
- `ushort`
- `int`
- `uint`
- `long`
- `ulong`
- `char`
- `string`

También se puede usar una expresión cuyo valor pueda ser convertido implícitamente a uno de los tipos de la lista anterior.

Las secciones de *switch* empiezan con la palabra clave de C# `case`, seguida de una expresión constante. A esa expresión constante le siguen dos puntos y a continuación escribimos la lista de instrucciones:

```
switch(MyVariable)
{
    case 123:
        System.Console.WriteLine("MyVariable == 123");
        break;
}
```

C# evalúa la expresión en la instrucción `switch` y luego busca un bloque `switch` cuya expresión constante concuerde con el valor de la expresión. Si C# puede encontrar un valor similar en una de las secciones de *switch*, la lista de instrucciones de la sección de *switch* se ejecuta. Una instrucción `switch` puede incluir muchas secciones de *switch*, cada una con un caso diferente:

```
switch(MyVariable)
{
```

```

case 123:
    System.Console.WriteLine("MyVariable == 123");
    break;
case 124:
    System.Console.WriteLine("MyVariable == 124");
    break;
case 125:
    System.Console.WriteLine("MyVariable == 125");
    break;
}

```

C# permite agrupar varias etiquetas de caso juntas. Si se tiene más de un caso que necesite ejecutar la misma lista de instrucciones, se pueden combinar las etiquetas de caso:

```

switch(MyVariable)
{
    case 123:
    case 124:
        System.Console.WriteLine("MyVariable == 123 or 124");
        break;
    case 125:
        System.Console.WriteLine("MyVariable == 125");
        break;
}

```

Una de las etiquetas de caso puede ser la palabra clave de C# `default`. La etiqueta `default` puede incluir su propia lista de instrucciones:

```

switch(MyVariable)
{
    case 123:
        System.Console.WriteLine("MyVariable == 123");
        break;
    default:
        System.Console.WriteLine("MyVariable != 123");
        break;
}

```

La lista de instrucciones `default` se ejecuta cuando ninguna de las otras secciones de `switch` define alguna constante que concuerde con la expresión `switch`. La lista de instrucciones `default` es la parte que dice "Si no puedes encontrar algún bloque `switch` que concuerde, ejecuta este código por defecto". El uso de la palabra clave `default` es opcional en sus instrucciones de `switch`.

Cómo usar instrucciones de iteración para ejecutar instrucciones incrustadas

Las instrucciones de iteración ejecutan instrucciones incrustadas varias veces. La expresión asociada con la instrucción de iteración controla el número de veces que se ejecuta una instrucción incrustada.

La instrucción while

La instrucción `while` ejecuta una lista de instrucciones incrustada siempre que la expresión `while` resulte ser `true`. La expresión booleana que controla la instrucción `while` se encierra entre los paréntesis que siguen a la palabra clave `while`. Tras los paréntesis situamos las instrucciones que se ejecutarán si la expresión booleana es `true`:

```
int MyVariable = 0;

while(MyVariable < 10)
{
    System.Console.WriteLine(MyVariable);
    MyVariable++;
}
```

El código escribe en la consola:

```
0
1
2
3
4
5
6
7
8
9
```

El código incrustado en la instrucción `while` continúa ejecutándose siempre que el valor de `MyVariable` sea menor que 10. Las instrucciones incrustadas escriben el valor de `MyVariable` en la consola y luego incrementan su valor. Cuando el valor de `MyVariable` alcanza 10, la expresión booleana `MyVariable < 10` devuelve `false` y la lista de instrucciones incrustada en la instrucción `while` deja de ejecutarse.

La instrucción que sigue a la instrucción `while` se ejecuta en cuanto la expresión booleana de la instrucción `while` devuelve `false`.

La instrucción do

La instrucción `do` ejecuta sus instrucciones incrustadas cero o más veces. Si la expresión booleana usada en la expresión `while` devuelve `false`, ninguna de las instrucciones incrustadas se ejecuta:

```
int MyVariable = 100;

while(MyVariable < 10)
{
    System.Console.WriteLine(MyVariable);
    MyVariable++;
}
```

Este código no escribe nada en la consola porque la expresión booleana usada en la instrucción `while`, `MyVariable < 10`, devuelve `false` la primera vez que se ejecuta. Como la expresión booleana devuelve `false` inmediatamente, las instrucciones incrustadas nunca se ejecutan. Si quiere asegurarse de que las instrucciones incrustadas se ejecuten al menos una vez, puede usar la instrucción `do`. La instrucción `do` va seguida de instrucciones incrustadas, que a su vez van seguidas de la palabra clave `while`. Tras ella va la expresión booleana que controla el número de veces que se ejecuta el bucle:

```
int MyVariable = 0;

do
{
    System.Console.WriteLine(MyVariable);
    MyVariable++;
}
while(MyVariable < 10);
```

Este código escribe lo siguiente en la consola:

```
0
1
2
3
4
5
6
7
8
9
```

Las sentencias incrustadas siempre se ejecutan al menos una vez debido a que la expresión booleana se evalúa después de que se ejecuten las instrucciones incrustadas, como se puede ver en el siguiente código:

```
int MyVariable = 100;

do
{
    System.Console.WriteLine(MyVariable);
    MyVariable++;
}
while(MyVariable < 10);
```

Este código escribe lo siguiente en la consola:

```
100
```

La instrucción `for`

La instrucción `for` es la instrucción de iteración más potente. El código de control de una instrucción `for` se divide en tres partes:

- Un *iniciador*, que fija las condiciones iniciales de la instrucción de bucle `for`.
- Una *condición*, que especifica la expresión booleana que mantiene ejecutándose la instrucción `for`.
- Un *iterador*, que especifica las instrucciones que se ejecutan al final de cada paso por las instrucciones incrustadas.

La instrucción `for` empieza con la palabra clave `for`, seguida por paréntesis, que contienen las instrucciones iniciadora, de condición y de iteración, todas separadas por puntos y coma. Las instrucciones incrustadas siguen a los paréntesis.

Observe el siguiente bucle simple `for`:

```
int MyVariable;

for(MyVariable = 0; MyVariable < 10; MyVariable++)
{
    System.Console.WriteLine(MyVariable);
}
```

El iniciador en este bucle `for` es la instrucción `MyVariable = 0`. El iniciador sólo se ejecuta una vez en un bucle `for` y se ejecuta antes de que las instrucciones incrustadas se ejecuten por primera vez.

La condición de este bucle `for` es la instrucción `MyVariable < 10`. La condición en un bucle `for` debe ser una expresión booleana. Las instrucciones incrustadas de un bucle `for` se ejecutan mientras esta expresión booleana devuelve `true`. Cuando la expresión devuelve `false`, las instrucciones incrustadas dejan de ejecutarse.

El iterador de este bucle `for` es la instrucción `MyVariable++`. El iterador se ejecuta después de cada paso por las instrucciones incrustadas del bucle `for`.

Si se pone toda esta información junta, se podrá interpretar la instrucción como: "Fija el valor de `MyVariable` igual a cero. Mientras el valor de `MyVariable` sea menor de 10, escribe el valor en la consola y luego aumenta el valor de `MyVariable`". Estas instrucciones escriben lo siguiente en la consola:

```
0
1
2
3
4
5
6
7
8
9
```

El iniciador, la condición y el iterador de un bucle `for` son opcionales. Si prefiere no usar alguna de estas partes, simplemente escriba un punto y coma sin

especificar la instrucción. El siguiente código es, en buena lógica, equivalente al código anterior:

```
int MyVariable = 0;

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

Este código también es equivalente al código original:

```
int MyVariable;

for(MyVariable = 0; MyVariable < 10; )
{
    System.Console.WriteLine(MyVariable);
    MyVariable++;
}
```

Hay que tener cuidado cuando se omite la parte de la condición en un bucle `for`. El siguiente código es un ejemplo de los problemas que pueden surgir si no se incluyen condiciones:

```
int MyVariable;

for(MyVariable = 0; ; MyVariable++)
{
    System.Console.WriteLine(MyVariable);
}
```

Este código se ejecuta hasta que `MyVariable` finalmente provoca un error porque contiene un número demasiado largo para ser almacenado. Esto ocurre porque ninguna condición del bucle `for` llega a devolver `false`, lo que permite a la variable aumentar hasta superar su límite. Las condiciones que faltan devuelven `true` en un bucle `for`. Como la condición en el código anterior de ejemplo siempre es `true`, nunca devuelve `false` y la instrucción `for` nunca deja de ejecutar su instrucción incrustada.

Las expresiones iniciadoras, de condición y de iteración pueden contener varias instrucciones, separadas por comas. El siguiente código es válido:

```
int MyFirstVariable;
int MySecondVariable;

for(MyFirstVariable = 0, MySecondVariable = 0;
    MyFirstVariable < 10;
    MyFirstVariable++, MySecondVariable++)
{
    System.Console.WriteLine(MyFirstVariable);
    System.Console.WriteLine(MySecondVariable);
}
```

La instrucción `foreach`

Se puede usar la instrucción `foreach` para repetir varias veces los elementos de una colección. Las matrices de C# admiten la instrucción `foreach` y pueden usarse para trabajar fácilmente con cada elemento de la matriz.

La instrucción `foreach` se usa escribiendo la palabra clave `foreach` seguida de paréntesis. Estos paréntesis deben contener la siguiente información:

- El tipo del elemento de la colección.
- Un nombre identificador para un elemento de la colección.
- La palabra clave `in`.
- El identificador de la colección.

Tras los paréntesis se colocan las instrucciones incrustadas.

El listado 5.1 muestra la instrucción `foreach` en acción. Crea una matriz entera de cinco elementos y luego usa la instrucción `foreach` para acudir a cada elemento de la matriz y escribir su valor en la consola.

Listado 5.1. Usando la instrucción `foreach`

```
class Listing5_1
{
    public static void Main()
    {
        int [] MyArray;

        MyArray = new int [5];
        MyArray[0] = 0;
        MyArray[1] = 1;
        MyArray[2] = 2;
        MyArray[3] = 3;
        MyArray[4] = 4;

        foreach(int ArrayElement in MyArray)
            System.Console.WriteLine(ArrayElement);
    }
}
```

El identificador `ArrayElement` es una variable definida en el bucle `foreach`. Contiene el valor de un elemento de la matriz. El bucle `foreach` recorre cada elemento de la matriz, lo que es muy útil cuando se necesita trabajar con cada elemento de una matriz sin tener que conocer el tamaño de la misma.

Instrucciones de salto para moverse por el código

Las *instrucciones de salto* saltan hacia una parte específica del código. Siempre se ejecutan y no están controladas por ninguna expresión booleana.

La instrucción break

Ya vio la instrucción `break` en la sección dedicada a las instrucciones `switch`. C# también permite usar la instrucción `break` para salir del bloque de instrucciones en el que se encuentre. Normalmente, la instrucción `break` se usa para salir de un bloque de instrucciones iterativas:

```
int MyVariable = 0;

while(MyVariable < 10)
{
    System.Console.WriteLine(MyVariable);
    if(MyVariable == 5)
        break;
    MyVariable++;
}
System.Console.WriteLine("Out of the loop.");
```

El código anterior escribe lo siguiente en la consola:

```
0
1
2
3
4
Out of the loop.
```

El código se interpreta: "Si el valor de `MyVariable` es 5, sal del bucle `while`". Cuando se ejecuta la instrucción `break`, C# transfiere el control a la instrucción que sigue a las instrucciones incrustadas de la instrucción de iteración. La instrucción `break` suele usarse con bloques de instrucciones `switch`, `while`, `do`, `for` y `foreach`.

La instrucción continue

La instrucción `continue` devuelve el control a la expresión booleana que controla una instrucción de iteración, como se puede ver en el siguiente código:

```
int MyVariable;

for(MyVariable = 0; MyVariable < 10; MyVariable++)
{
    if(MyVariable == 5)
        continue;
    System.Console.WriteLine(MyVariable);
}
```

El código anterior escribe lo siguiente en la consola:

```
0
1
2
```

3
4
6
7
8
9

Este código interpreta: "Si el valor de `MyVariable` es 5, continúa hasta la siguiente iteración del bucle `for` sin ejecutar ninguna otra instrucción incrustada". Por eso no aparece el 5 en pantalla. Cuando el valor de `MyVariable` es 5, el control regresa a la parte superior del bucle `for` y la llamada a `WriteLine()` nunca se produce en esa iteración del bucle `for`.

Al igual que la instrucción `break`, la instrucción `continue` suele usarse en los bloques de instrucciones `switch`, `while`, `do`, `for` y `foreach`.

La instrucción `goto`

La instrucción `goto` transfiere sin condiciones el control a una instrucción etiquetada. Puede etiquetarse cualquier instrucción de C#. Las etiquetas de instrucciones son identificadores que preceden a una instrucción. Después de una etiqueta de instrucción se colocan dos puntos. Un identificador de etiqueta sigue a la palabra clave `goto` y la instrucción `goto` transfiere el control a la instrucción designada por el identificador de etiqueta, como muestra el siguiente código:

```
int MyVariable = 0;

while(MyVariable < 10)
{
    System.Console.WriteLine(MyVariable);
    if(MyVariable == 5)
        goto Done;
    MyVariable++;
}
Done: System.Console.WriteLine("Out of the loop.");
```

El código anterior escribe lo siguiente en la consola:

```
0
1
2
3
4
Out of the loop.
```

Cuando el valor de `MyVariable` es 5, la instrucción `goto` se ejecuta y transfiere el control a la instrucción con la etiqueta `Done`. La instrucción `goto` siempre se ejecuta, independientemente de la instrucción de iteración que pueda estar ejecutándose.

También se puede usar la palabra clave `goto` en conjunción con las etiquetas de caso en una instrucción `switch`, en lugar de una instrucción `break`:

```

switch(MyVariable)
{
    case 123:
        System.Console.WriteLine("MyVariable == 123");
        goto case 124;
    case 124:
        System.Console.WriteLine("MyVariable == 124");
        break;
}

```

NOTA: Usar la instrucción `goto` en muchos sitios puede hacer el código confuso e ilegible. Lo mejor es evitar usar una instrucción `goto` siempre que sea posible. Intente reestructurar el código para no tener que recurrir al uso de una instrucción `goto`.

Cómo usar instrucciones para realizar cálculos matemáticos con seguridad

Ya ha visto cómo las palabras clave `checked` y `unchecked` permiten controlar el comportamiento de las condiciones de error en sus expresiones matemáticas. También se pueden usar estas palabras clave como instrucciones para controlar la seguridad de sus operaciones matemáticas. Use las palabras clave antes de un bloque de instrucciones al que afecte la palabra reservada `checked` o `unchecked`, como en el siguiente código:

```

checked
{
    Int1 = 2000000000;
    Int2 = 2000000000;
    Int1PlusInt2 = Int1 + Int2;
    System.Console.WriteLine(Int1PlusInt2);
}

```

Resumen

C# dispone de varios medios de controlar la ejecución del código, dándole opciones para ejecutar un bloque de código más de una vez o, a veces, ninguna vez, basándose en el resultado de una expresión booleana.

La instrucción `if` ejecuta el código una vez, pero sólo si la expresión booleana que la acompaña devuelve `true`. La instrucción `if` puede incluir una cláusula `else`, que ejecuta un bloque de código si la expresión booleana devuelve `false`.

La instrucción `switch` ejecuta uno de los muchos bloques de código posibles. Cada bloque de código viene precedido de una lista de instrucciones de caso.

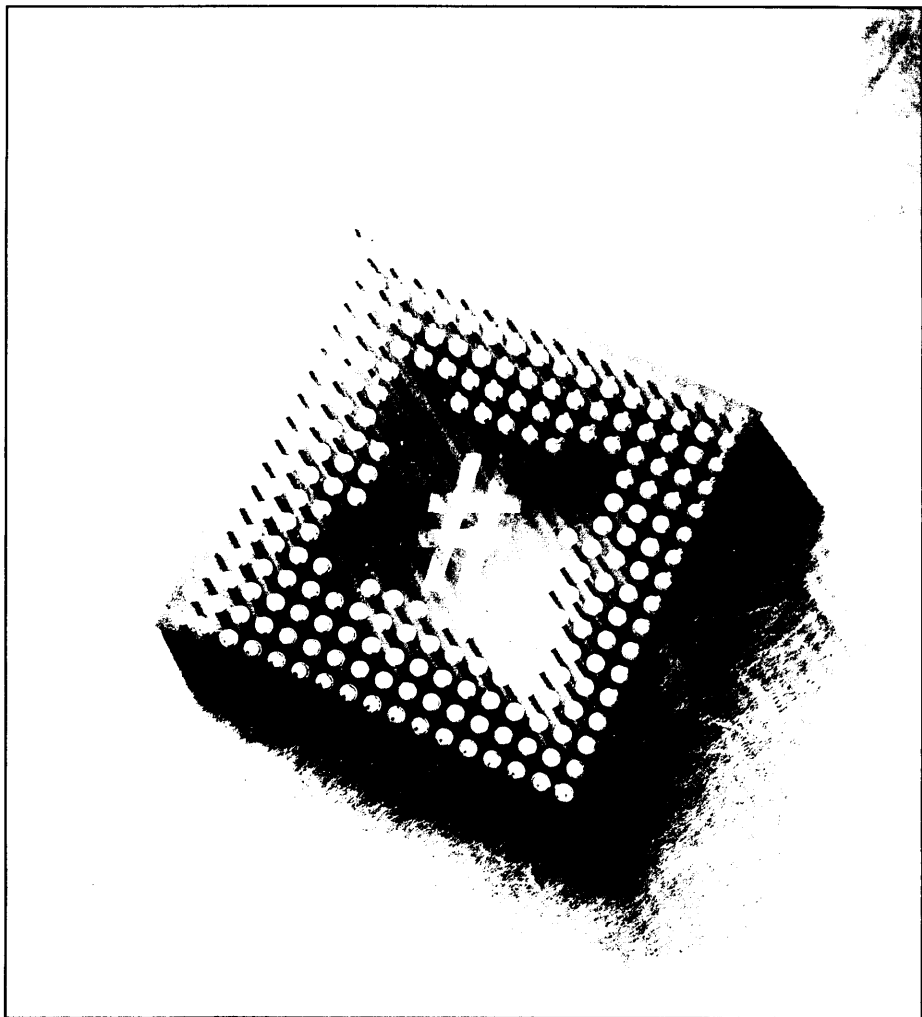
C# evalúa la expresión de la instrucción `switch` y a continuación busca una lista de instrucciones de caso cuyo valor coincida con la expresión evaluada en la instrucción `switch`.

Las instrucciones `while`, `do` y `for` continúan ejecutando el código mientras la expresión booleana indicada sea `true`. Cuando la expresión booleana devuelve `false`, las instrucciones incrustadas dejan de ejecutarse. Las instrucciones `while` y `for` se pueden definir para que sus expresiones booleanas devuelvan inmediatamente `false`, lo que quiere decir que sus instrucciones incrustadas nunca llegan a ejecutarse realmente. La instrucción `do`, sin embargo, siempre ejecuta sus instrucciones incrustadas al menos una vez.

La instrucción `foreach` proporciona un buen modo de recorrer repetida y rápidamente los elementos de una matriz. Puede ordenar a un bucle `foreach` que recorra repetidamente los elementos de una matriz sin conocer el tamaño de la matriz o los elementos que la forman. La instrucción `foreach` prepara un identificador especial formado por el valor del elemento de una matriz durante cada iteración del bucle `foreach`.

Las instrucciones `break`, `continue` y `goto` afectan al flujo normal de una instrucción de iteración, como `while` o `foreach`. La instrucción `break` sale del bucle iterativo, incluso si la expresión booleana que controla la ejecución del bucle sigue devolviendo `true`. La instrucción `continue` devuelve el control a la parte superior del bucle iterativo sin ejecutar ninguna de las instrucciones incrustadas que la siguen. La instrucción `goto` siempre transfiere el control a la instrucción etiquetada.

Puede acompañar sus operaciones matemáticas con instrucciones `checked` o `unchecked` para especificar cómo quiere tratar los errores matemáticos del código C#.



6 Cómo trabajar con métodos

Los métodos son bloques de instrucciones que devuelven algún tipo de valor cuando se ejecutan. Pueden ser llamados mediante el nombre y llamar a un método hace que las instrucciones del método se ejecuten. Ya hemos visto un método: el método `Main()`. Aunque C# permite poner todo el código en el método `Main()`, probablemente quiera diseñar sus clases para definir más de un método. El uso de métodos mantiene el código legible porque las instrucciones se colocan en bloques más pequeños, en lugar de en un gran bloque de código. Los métodos también permiten tomar instrucciones que pueden ser ejecutadas varias veces y colocarlas en un bloque de código que puede ser llamado todas las veces que haga falta.

En este capítulo, aprenderá a crear funciones que devuelven datos y que no los devuelven. Aprenderá a pasar parámetros a los métodos y la mejor manera de estructurar un método para hacer sus aplicaciones modulares.

La estructura de un método

Como mínimo, un método está compuesto de las siguientes partes:

- Tipo devuelto
- Nombre del método

- Lista de parámetros
- Cuerpo del método

NOTA: Todos los métodos se encierran en una clase. Un método no puede existir fuera de una clase.

Los métodos tienen otras partes opcionales, como las listas de atributos y los modificadores de ámbito. Las siguientes secciones analizan los fundamentos de un método.

Tipo devuelto

Un método comienza definiendo el tipo de datos que devolverá cuando se le llame. Por ejemplo, suponga que quiere escribir un método que suma dos números enteros y devuelve el resultado. En ese caso, escribirá el tipo devuelto como `int`.

C# permite escribir un método que no devuelve nada. Por ejemplo, puede escribir un método que simplemente escriba algún texto en la consola, pero que no calcule ningún dato que deba devolver al código que llamó al método. En ese caso, se puede usar la palabra clave `void` para indicar al compilador de C# que el método no devuelve ningún dato.

Si se quiere devolver un valor de un método se usa la palabra clave `return` para especificar el valor que debe devolverse. La palabra clave va seguida de una expresión que evalúa el tipo de valor que debe devolverse. Esta expresión puede ser un valor literal, una variable o una expresión más compleja.

Nombre del método

Todos los métodos deben tener un nombre. Un nombre de método es un identificador y los nombres de método deben seguir las reglas de nomenclatura de cualquier identificador. Recuerde que los identificadores deben empezar con una letra mayúscula o minúscula o con un carácter subrayado. Los caracteres que siguen al primer carácter pueden ser una letra mayúscula o minúscula, un número o un subrayado.

Lista de parámetros

Se puede llamar a métodos con los parámetros que se usan para pasar los datos al método. En el ejemplo anterior, en el que un método suma dos números enteros, se necesitaría enviar al método los valores de los dos números enteros que se van a sumar. La lista de variables recibe el nombre de *lista de parámetros* del método. La lista de parámetros del método aparece entre paréntesis y sigue al nombre del

método. Cada parámetro de la lista de parámetros está separado por una coma e incluye el tipo del parámetro seguido por su nombre.

También se puede prefijar los parámetros de la lista de parámetros con modificadores que especifican cómo se usan sus valores dentro del método. Veremos estos modificadores más adelante en este mismo capítulo.

Se pueden definir métodos que no reciben ningún parámetro. Si se quiere utilizar uno de estos métodos, basta con dejar vacíos los paréntesis. Ya hemos visto esto en los métodos `Main()` escritos. También se puede colocar la palabra clave `void` entre los paréntesis para especificar que el método no acepta parámetros.

Cuerpo del método

El cuerpo del método es el bloque de instrucciones que compone el código del método. El cuerpo del método está entre llaves. La llave de apertura se incluye tras la lista de parámetros del método y la llave de cierre se coloca detrás de la última instrucción del cuerpo del método.

Cómo llamar a un método

Para llamar a un método, se escribe su nombre en el lugar donde debería ejecutarse el código de ese método. Después del nombre del método se escriben dos paréntesis, como se muestra en el listado 6.1. Como en todas las instrucciones de C#, la instrucción de llamada al método debe terminar con un punto y coma.

Listado 6.1. Llamar a un método simple

```
class Listing6_1
{
    public static void Main()
    {
        Listing6_1 MyObject;

        MyObject = new Listing6_1();
        MyObject.CallMethod();
    }

    void CallMethod()
    {
        System.Console.WriteLine("Hello from CallMethod()!");
    }
}
```

NOTA: Necesita las instrucciones de `Main()` que crean un nuevo objeto `Listing6_1` antes de poder llamar a los métodos del objeto.

Si el método se define con una lista de parámetros, sus valores deben ser especificados en el momento de llamar al método. Debe especificar los parámetros en el mismo orden en que son especificados en la lista de parámetros del método, como muestra el listado 6.2.

Listado 6.2. Llamada a un método con un parámetro

```
class Listing6_2
{
    public static void Main()
    {
        int MyInteger;
        Listing6_2 MyObject;

        MyObject = new Listing6_2();
        MyObject.CallMethod(2);
        MyInteger = 3;
        MyObject.CallMethod(MyInteger);
    }

    void CallMethod(int Integer)
    {
        System.Console.WriteLine(Integer);
    }
}
```

Cuando compile y ejecute el listado 6.2 obtendrá un resultado igual al de la figura 6.1.

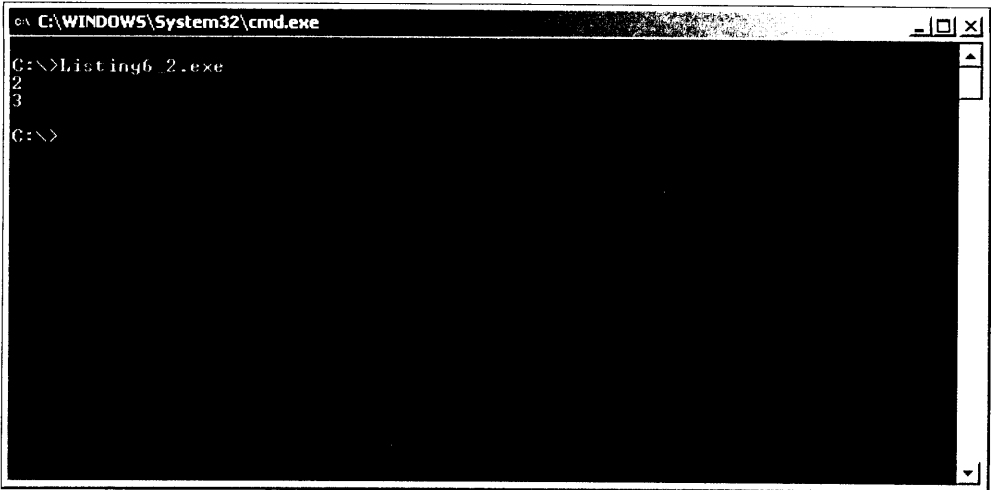


Figura 6.1. Una simple llamada a un método devuelve este resultado.

Esto es debido a que el método `Main()` llama a `CallMethod()` dos veces: una con el valor 2 y otra con el valor 3. El cuerpo del método `CallMethod()` escribe el valor suministrado en la consola.

Cuando suministramos un valor para el parámetro de un método, podemos usar un valor literal, como el 2 del listado 6.2 o suministrar una variable y usar su valor, como en la variable `MyInteger` del listado 6.2.

Cuando se llama a un método, C# toma el valor especificado y asigna esos valores a los parámetros que se usan en el método. Durante la primera llamada a `CallMethod()` en el listado 6.2, el literal 2 se usa como el parámetro del método y el parámetro `Integer` del método recibe el valor 2. Durante la segunda llamada a `CallMethod()` en el listado 6.2, la variable `MyInteger` se usa como el parámetro del método y el parámetro `Integer` del método recibe el valor de la variable `MyInteger`: 3.

Los parámetros que se especifican cuando se llama a un método deben concordar con los tipos especificados en la lista de parámetros. Si un parámetro de la lista de parámetros del método especifica un tipo `int`, por ejemplo, los parámetros que le pase deberán ser de tipo `int` o de un tipo que pueda ser convertido a `int`. Cualquier otro tipo produce un error al compilar el código.

C# es un lenguaje de tipo seguro, lo que significa que se comprueba la legalidad de los tipos de variables al compilar el código C#. Por lo que respecta a los métodos, esto significa que deben especificarse los tipos correctos cuando se especifican parámetros.

El listado 6.3 muestra los tipos correctos durante la especificación de parámetros:

Listado 6.3. Seguridad de tipos en listas de parámetros de método

```
class Listing6_3
{
    public static void Main()
    {
        Listing6_3 MyObject;

        MyObject = new Listing6_3();
        MyObject.CallMethod("a string");
    }

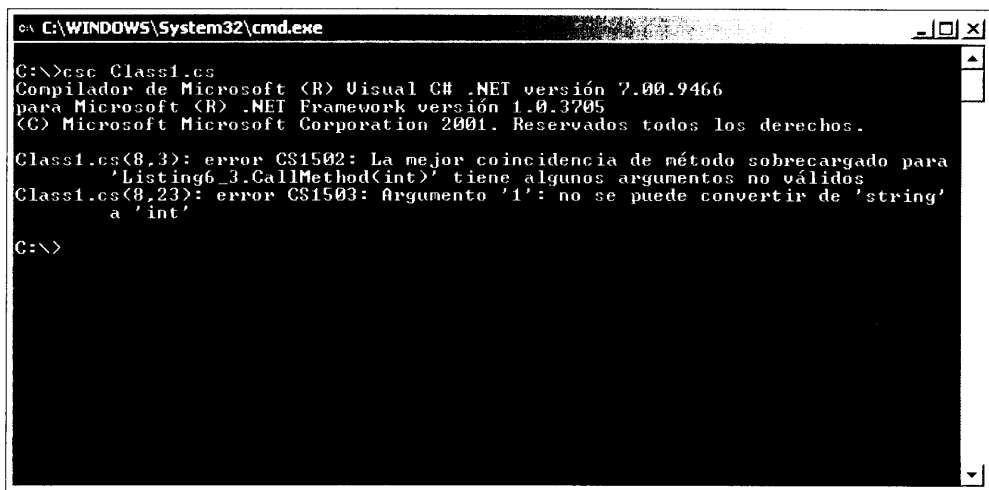
    void CallMethod(int Integer)
    {
        System.Console.WriteLine(Integer);
    }
}
```

Este código no se compila, como puede verse en la figura 6.2.

El compilador de C# emite estos errores porque el método `CallMethod()` se está ejecutando con un parámetro de cadena y la lista de parámetros `CallMethod()` especifica que se debe usar un número entero como parámetro. Las cadenas no son números enteros ni pueden ser convertidas a números enteros y esta discordancia hace que el compilador de C# genere errores.

Si el método devuelve un valor, debe declararse una variable que contenga el valor devuelto. La variable usada para esta operación se coloca antes del nombre

del método y un signo igual separa el identificador de la variable y el nombre del método. como se ve en el listado 6.4.



```
C:\WINDOWS\System32\cmd.exe
C:\>csc Class1.cs
Compilador de Microsoft (R) Visual C# .NET versión 7.00.9466
para Microsoft (R) .NET Framework versión 1.0.3705
(C) Microsoft Microsoft Corporation 2001. Reservados todos los derechos.
Class1.cs(8,3): error CS1502: La mejor coincidencia de método sobrecargado para
'Listing6_3.CallMethod(int)' tiene algunos argumentos no válidos
Class1.cs(8,23): error CS1503: Argumento '1': no se puede convertir de 'string'
a 'int'
C:\>
```

Figura 6.2. La llamada a un método con un tipo de datos no válido produce errores de compilación.

Listado 6.4. Devolución de un valor de un método

```
class Listing6_4
{
    public static void Main()
    {
        Listing6_4 MyObject;
        int ReturnValue;

        MyObject = new Listing6_4();
        ReturnValue = MyObject.AddIntegers(3, 5);
        System.Console.WriteLine(ReturnValue);
    }

    int AddIntegers(int Integer1, int Integer2)
    {
        int Sum;

        Sum = Integer1 + Integer2;
        return Sum;
    }
}
```

En este código suceden varias cosas:

- Se declara un método llamado `AddIntegers()`. El método tiene dos parámetros en su lista de parámetros: un número entero llamado `Integer1` y otro número entero llamado `Integer2`.

- El cuerpo del método `AddIntegers()` suma los valores de los dos parámetros y asigna el resultado a una variable local llamada `Sum`. Se devuelve el valor de `Sum`.
- El método `Main()` llama al método `AddIntegers()` con los valores 3 y 5. El valor devuelto del método `AddIntegers()` se coloca en una variable local llamada `ReturnValue`.
- El valor de `ReturnValue` se escribe en la consola.

La figura 6.3 contiene los resultados del programa que aparece en el listado 6.4.

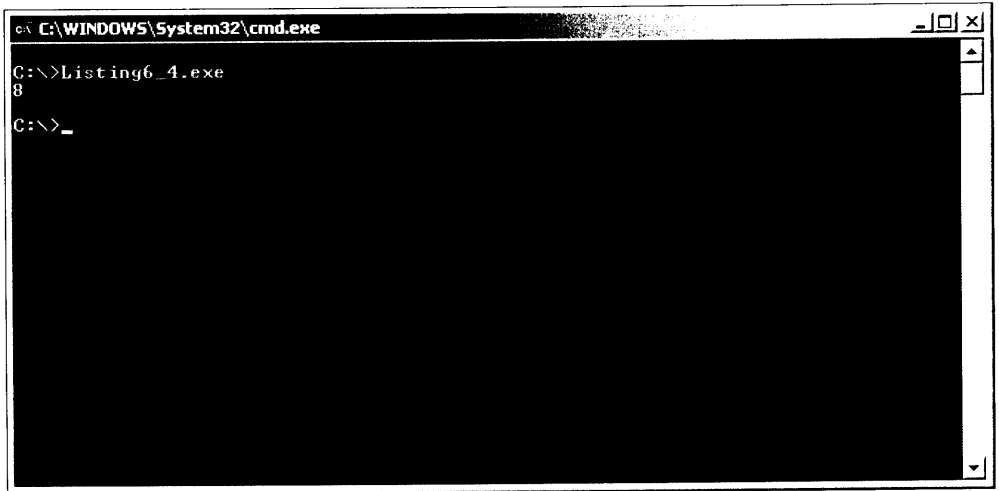


Figura 6.3. Los datos se devuelven de un método y se muestran en la ventana de la consola.

Tipos de parámetros

C# permite cuatro tipos de parámetros en una lista de parámetros:

- Parámetros de entrada
- Parámetros de salida
- Parámetros de referencia
- Matrices de parámetros

Parámetros de entrada

Los *parámetros de entrada* son parámetros cuyo valor es enviado al método. Todos los parámetros que se han usado hasta ahora han sido parámetros de entra-

da. Los valores de estos parámetros de entrada se envían a la función, pero el cuerpo del método no puede cambiar permanentemente sus valores.

El listado 6.4 del anterior ejemplo define un método con dos parámetros de entrada: `Integer1` y `Integer2`. Los valores de estos parámetros se introducen en el método, que lee sus valores y hace su trabajo. Los parámetros de entrada se pasan a los métodos por valor. Básicamente, el método ve una copia del valor del parámetro, pero no se le permite cambiar el valor proporcionado por la parte que realiza la llamada. En el listado 6.5 se puede ver un ejemplo.

Listado 6.5. Cómo modificar copias de parámetros de entrada

```
class Listing6_5
{
    public static void Main()
    {
        int MyInteger;
        Listing6_5 MyObject;

        MyObject = new Listing6_5();
        MyInteger = 3;
        MyObject.CallMethod(MyInteger);
        System.Console.WriteLine(MyInteger);
    }

    void CallMethod(int Integer1)
    {
        Integer1 = 6;
        System.Console.WriteLine(Integer1);
    }
}
```

En el listado 6.5, el método `Main()` establece una variable entera llamada `MyInteger` y le asigna el valor de 3. A continuación llama a `MyMethod()` con `MyInteger` como parámetro. El método `CallMethod()` establece el valor del parámetro en 6 y luego escribe el valor en la consola. Cuando el método `CallMethod()` se devuelve, el método `Main()` continúa y escribe el valor de `MyInteger`. Si ejecuta este código, el resultado debería parecerse al de la figura 6.4. Este resultado se produce porque el método `CallMethod()` modifica su copia del parámetro de entrada, pero esa modificación no afecta al valor del método original proporcionado por `Main()`. El valor de `MyInteger` sigue siendo 3 después de que regrese el método `CallMethod()`, debido a que `CallMethod()` no puede cambiar el valor del parámetro de entrada del elemento que hace la llamada. Sólo puede cambiar el valor de su copia del valor.

Parámetros de salida

Los *parámetros de salida* son parámetros cuyos valores no se establecen cuando se llama al método. En su lugar, el método establece los valores y los devuelve al

elemento que hace la llamada mediante el parámetro de salida. Suponga, por ejemplo, que quiere escribir un método que cuente el número de registros de una tabla de una base de datos. Suponga que también quiere especificar si la operación se realizó satisfactoriamente. (La operación puede no realizarse si, por ejemplo, la tabla de bases de datos no está disponible.) Por tanto, queremos que el método devuelva dos instancias de información:

- Un contador de registros.
- Un indicador de éxito de operación.

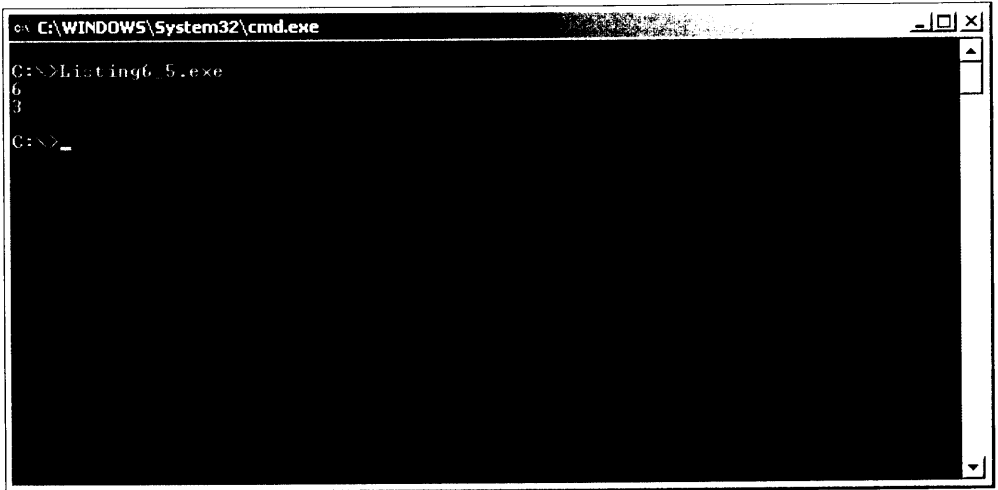


Figura 6.4. Demostración de parámetros de entrada con la función `CallMethod()`.

C# sólo permite a los métodos devolver un valor. ¿Qué hacer si queremos que devuelva dos instancias de información?

La respuesta está en el concepto de parámetro de salida. Puede hacer que su método devuelva el indicador de éxito de la operación como un valor booleano y especificar el recuento de registros como un parámetro de salida. El método almacena el recuento de registros en una variable de salida, cuyo valor es recogido por el elemento que hizo la llamada.

Los parámetros de salida se especifican en listas de parámetros con la palabra clave `out`. La palabra clave `out` debe preceder al tipo de parámetro en la lista de parámetros. Cuando se llama a un método con un parámetro de salida, se debe declarar una variable que contenga ese valor, como se ve en el listado 6.6.

Listado 6.6. Cómo trabajar con parámetros de salida

```
class Listing6_6
{
    public static void Main()
    {
        int MyInteger;
```

```

Listing6_6 MyObject;

MyObject = new Listing6_6();
MyObject.CallMethod(out MyInteger);
System.Console.WriteLine(MyInteger);
}

void CallMethod(out int Integer1)
{
    Integer1 = 7;
}
}

```

El listado 6.6 define un método llamado `CallMethod()`, que define un parámetro de salida entero llamado `Integer1`. El cuerpo del método establece el valor del parámetro de salida en 7. El método `Main()` declara un entero llamado `MyInteger` y lo usa como parámetro de salida para el método `CallMethod()`. Cuando `CallMethod()` se devuelve, el valor de `MyInteger` se escribe en la consola. La figura 6.5 contiene el resultado de estas aplicaciones de prueba de parámetros.

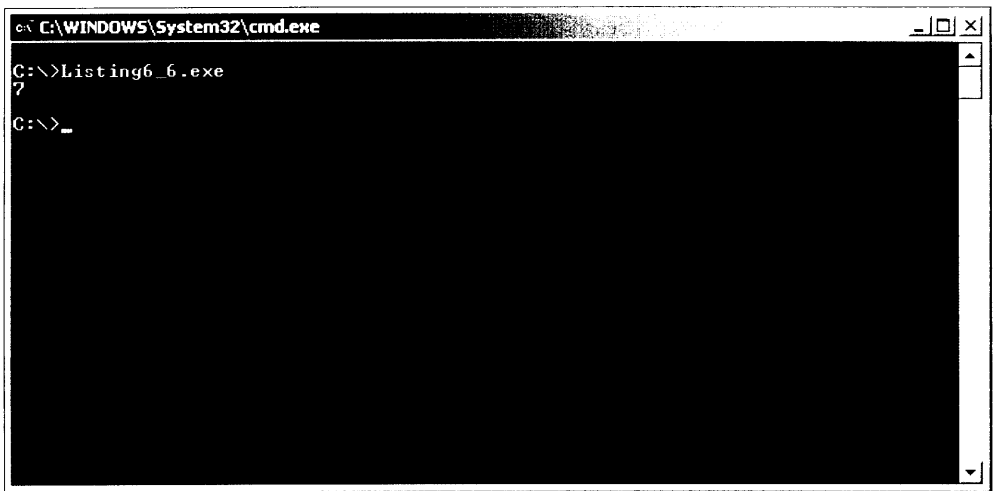


Figura 6.5. Un parámetro de salida devuelve el valor apropiado.

Debe usarse la palabra clave `out` dos veces por cada parámetro: una vez cuando se declara el parámetro y otra vez cuando se especifica el parámetro de salida al llamar al método. Si se olvida la palabra clave `out` al llamar a un método con un parámetro de salida, se obtienen los siguientes errores del compilador:

```

Listing6-6.cs(9,6): error CS1502: La mejor coincidencia de
metodo sobrecargado para 'Listing6_6.CallMethod(out int)'
tiene algunos argumentos no validos
Listing6-6.cs(9,26): error CS1503: Argumento '1': no se
puede convertir de 'int' a 'out int'

```

Cualquier valor asignado a variables usadas como parámetros de salida antes de que se llame al método se pierde. Los valores originales se sobrescriben con los valores que les asignó el método.

Parámetros de referencia

Los *parámetros de referencia* proporcionan valores por referencia. En otras palabras, el método recibe una referencia a la variable especificada cuando se llama al método. Piense en un parámetro de referencia como en una variable de entrada y de salida: El método puede leer el valor original de la variable y también modificar el valor original como si fuera un parámetro de salida.

Los parámetros de referencia se especifican en listas de parámetros con la palabra clave `ref`. La palabra clave `ref` debe preceder al tipo del parámetro en la lista de parámetros. Cuando se llama a un método con un parámetro de referencia, se debe declarar una variable para que contenga el valor del parámetro de referencia, como se ve en el listado 6.7.

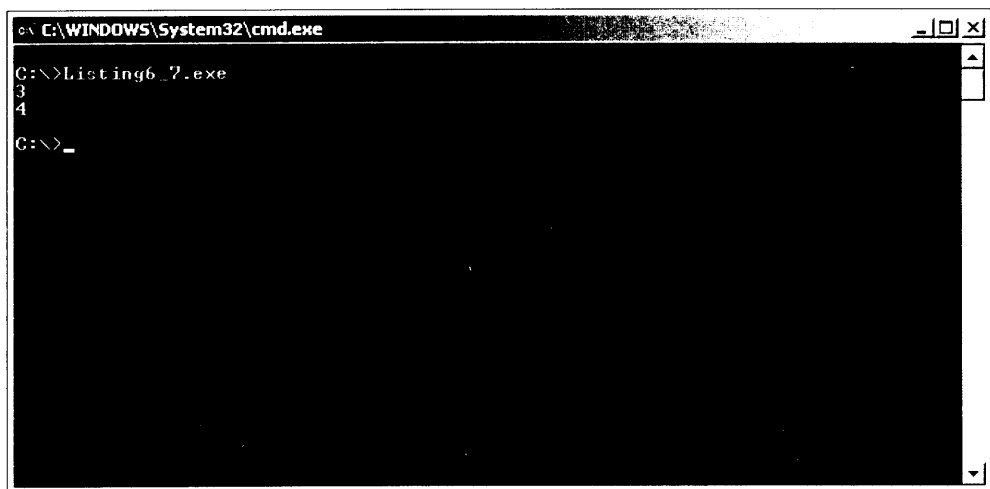
Listado 6.7. Cómo trabajar con parámetros de referencia

```
class Listing6_7
{
    public static void Main()
    {
        int MyInteger;
        Listing6_7 MyObject;

        MyObject = new Listing6_7();
        MyInteger = 3;
        System.Console.WriteLine(MyInteger);
        MyObject.CallMethod(ref MyInteger);
        System.Console.WriteLine(MyInteger);
    }

    void CallMethod(ref int Integer1)
    {
        Integer1 = 4;
    }
}
```

El método `CallMethod()` del listado 6.7 usa un parámetro de referencia llamado `Integer1`. El cuerpo del método establece el valor del parámetro de referencia en 4. El método `Main()` declara una variable entera llamada `MyInteger` y le asigna un valor de 3. Escribe el valor de `MyInteger` en la consola y luego lo usa como el parámetro del método `CallMethod()`. Cuando `CallMethod()` se devuelve, el valor de `MyInteger` se escribe en la consola una segunda vez. Si ejecuta el código del listado 6.7 debería obtener los valores que aparecen en la figura 6.6.



```
C:\WINDOWS\System32\cmd.exe
C:\>Listing6_7.exe
3
4
C:\>_
```

Figura 6.6. Un parámetro de referencia cambia su variable directamente.

La segunda línea lee 4 porque la sintaxis del parámetro de referencia permite al método cambiar el valor de la variable original. Éste es un cambio respecto al ejemplo de parámetro de entrada del listado 6.5.

Debe usarse la palabra clave `ref` dos veces para cada parámetro: una vez cuando se declara el parámetro en la lista de parámetros y otra vez cuando se especifica el parámetro de referencia al llamar al método.

Matrices de parámetros

Los métodos suelen escribirse para recibir un número específico de parámetros. Un método con una lista de tres parámetros siempre espera ser llamada con tres parámetros, ni más, ni menos. Sin embargo, a veces puede ocurrir que un método no conozca cuántos parámetros debe aceptar al ser diseñado. Puede escribir un método que acepte una lista de cadenas que especifiquen los nombres de los registros que deberán borrarse del disco. ¿Cuántas cadenas debe permitir el método? Para ser flexible, el método debe diseñarse de manera que el invocador pueda especificar las cadenas que necesita. Esto hace que llamar al método sea un poco más flexible porque el elemento que realiza la llamada ahora puede decidir cuántas cadenas deben pasarse al método. Sin embargo, ¿cómo escribiremos la lista de parámetros de la lista cuando el método no conoce cuántos parámetros le serán pasados?

Las matrices de parámetros resuelven este problema de diseño ya que permiten especificar que el método acepte un número variable de argumentos. La matriz de parámetros de la lista de parámetros se especifica usando la palabra clave de C# `params`, seguida del tipo de variable que deberá proporcionar el llamador. La especificación de tipos va seguida de corchetes, que a su vez van seguidos del identificador de la matriz de parámetros, como se ve en el listado 6.8.

Listado 6.8. Cómo trabajar con matrices de parámetros

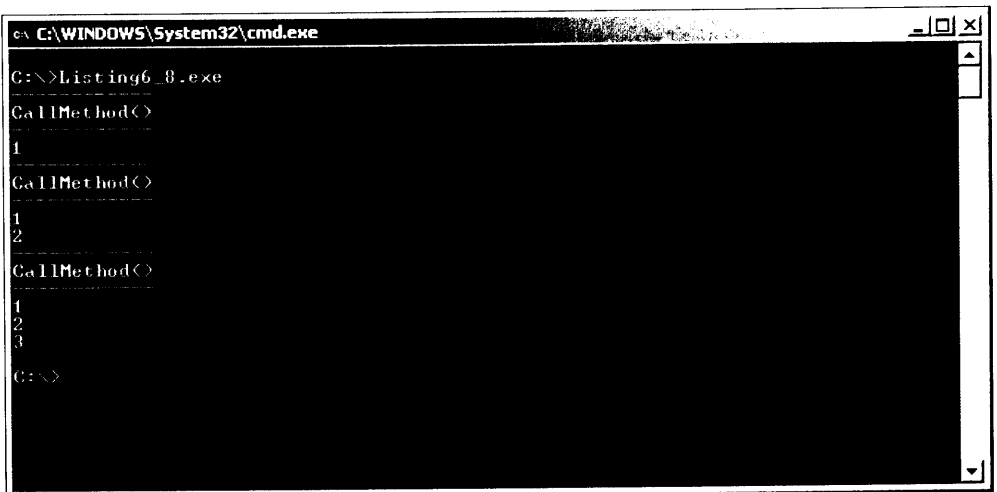
```
class Listing6_8
{
    public static void Main()
    {
        Listing6_8 MyObject;

        MyObject = new Listing6_8();
        MyObject.CallMethod(1);
        MyObject.CallMethod(1, 2);
        MyObject.CallMethod(1, 2, 3);
    }

    void CallMethod(params int[] ParamArray)
    {
        System.Console.WriteLine("-----");
        System.Console.WriteLine("CallMethod()");
        System.Console.WriteLine("-----");
        foreach(int ParamArrayElement in ParamArray)
            System.Console.WriteLine(ParamArrayElement);
    }
}
```

En el listado 6.8, el método `CallMethod()` está escrito para aceptar un número variable de enteros. El método recibe los parámetros en forma de matriz de números enteros. El cuerpo del método usa la instrucción `foreach` para iterar la matriz de parámetros y escribe cada elemento en la consola.

El método `Main()` llama al método `CallMethod()` tres veces, cada vez con un número de argumentos diferente. Esto es posible sólo porque `CallMethod()` se declara con una matriz de parámetros. La figura 6.7 indica que todos los parámetros fueron pasados intactos al método.



```
C:\WINDOWS\System32\cmd.exe
C:\>Listing6_8.exe
CallMethod<>
1
-----
CallMethod<>
1
2
-----
CallMethod<>
1
2
3
C:\>
```

Figura 6.7. La palabra clave `params` permite cualquier número de parámetros.

Puede usar una matriz de parámetros en su lista de parámetros de método. Puede combinar una matriz de parámetros con otros parámetros en la lista de parámetros de método. Sin embargo, si usa una matriz de parámetros en una lista de parámetros de método, debe especificarla como el último parámetro de la lista. No se puede usar las palabras clave `out` o `ref` en una matriz de parámetros.

Sobrecarga de métodos

C# permite definir varios métodos con el mismo nombre en la misma clase, siempre que esos métodos tengan listas de parámetros diferentes. Esta operación se conoce como *sobrecargar* el nombre del método. Observe el ejemplo en el listado 6.9.

Listado 6.9. Cómo trabajar con métodos sobrecargados

```
class Listing6_9
{
    public static void Main()
    {
        Listing6_9 MyObject;

        MyObject = new Listing6_9();
        MyObject.Add(3, 4);
        MyObject.Add(3.5, 4.75);
    }

    void Add(int Integer1, int Integer2)
    {
        int Sum;

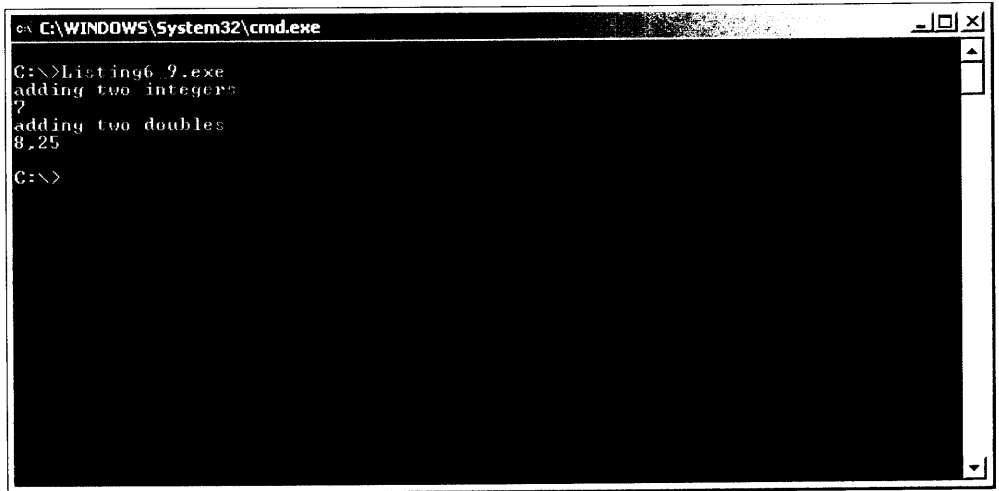
        System.Console.WriteLine("adding two integers");
        Sum = Integer1 + Integer2;
        System.Console.WriteLine(Sum);
    }

    void Add(double Double1, double Double2)
    {
        double Sum;

        System.Console.WriteLine("adding two doubles");
        Sum = Double1 + Double2;
        System.Console.WriteLine(Sum);
    }
}
```

El listado 6.9 implementa dos métodos `Add()`. Uno de ellos toma dos números enteros como parámetros de entrada y el otro toma dos dobles. Como las dos implementaciones tienen diferentes listas de parámetros, C# permite que los dos

métodos `Add()` coexistan en la misma clase. El método `Main()` llama al método `Add()` dos veces: una vez con dos parámetros integrales y otra con dos valores de punto flotante. Como puede ver en la figura 6.8, los dos métodos se ejecutan satisfactoriamente, procesando los datos correctos.



```
cmd C:\WINDOWS\System32\cmd.exe
C:\>Listing6_9.exe
adding two integers
7
adding two doubles
8.25
C:\>
```

Figura 6.8. El método sobrecargado suma números enteros y dobles.

Cuando el compilador de `C#` encuentra una llamada a un método que tiene más de una implementación, examina los parámetros usados en la llamada y llama al método con la lista de parámetros que mejor concuerde con los parámetros usados en la llamada. En la primera llamada a `Add()` se usan dos números enteros. Entonces, el compilador de `C#` empareja esta llamada con la implementación de `Add()` que toma los dos parámetros de entrada enteros porque los parámetros de la llamada concuerdan con la lista de parámetros que tiene los números enteros. En la segunda llamada a `Add()` se usan dos dobles. El compilador de `C#` entonces empareja esta llamada con la implementación de `Add()` que toma los dos parámetros de entrada dobles porque los parámetros de la llamada concuerdan con la lista de parámetros que tiene los dobles. No todos los métodos sobrecargados necesitan usar el mismo número de parámetros en su lista de parámetros, ni todos los parámetros de la lista de parámetros tienen que ser del mismo tipo. El único requisito que `C#` exige es que las funciones tengan diferentes listas de parámetros. Una versión de una función sobrecargada puede tener un entero en su lista de parámetros y la otra versión puede tener tipos de datos como `string`, `long` y `character` en su lista de parámetros.

Métodos virtuales

Para proseguir con el tema de los métodos virtuales, hay que comprender el concepto de herencia. La *herencia* basa una clase en otra ya existente, añadiendo

o quitando funcionalidad según se necesite. En las siguientes secciones examinaremos cómo se crean y se usan los métodos virtuales.

Métodos sobrecargados

Para empezar esta sección, construirá un ejemplo de clase llamado `Books`. Esta clase contiene dos métodos llamados `Title` y `Rating`. El método `Title` devuelve el nombre de un libro y el método `Rating` devuelve una cadena indicando el número de estrellas con que ha sido calificado el libro en cuestión. En el listado 6.10 se recoge el código completo de su aplicación. Escríbalo en su editor de texto preferido y compílelo como hizo antes.

Listado 6.10. Cómo mostrar la información del título y la puntuación de un libro con las siguientes clases

```
using System;

namespace BookOverride
{
    class Book
    {
        public string Title()
        {
            return "Programming Book";
        }
        public string Rating()
        {
            return "5 Stars";
        }
    }

    class Class1
    {
        static void Main(string[] args)
        {
            Book bc = new Book();
            Console.WriteLine(bc.Title());
            Console.WriteLine(bc.Rating());
        }
    }
}
```

Antes de ejecutar este programa, repáselo rápidamente. Como puede observar, una clase contiene el método `Main()`. Este método es donde se inicializa una instancia de la clase `BookOverride`, que contiene los métodos `Title` y `Rating`.

Después de inicializar una instancia, se llama a los métodos `Title` y `Rating` y se escribe la salida en la consola. El resultado puede verse en la figura 6.9.

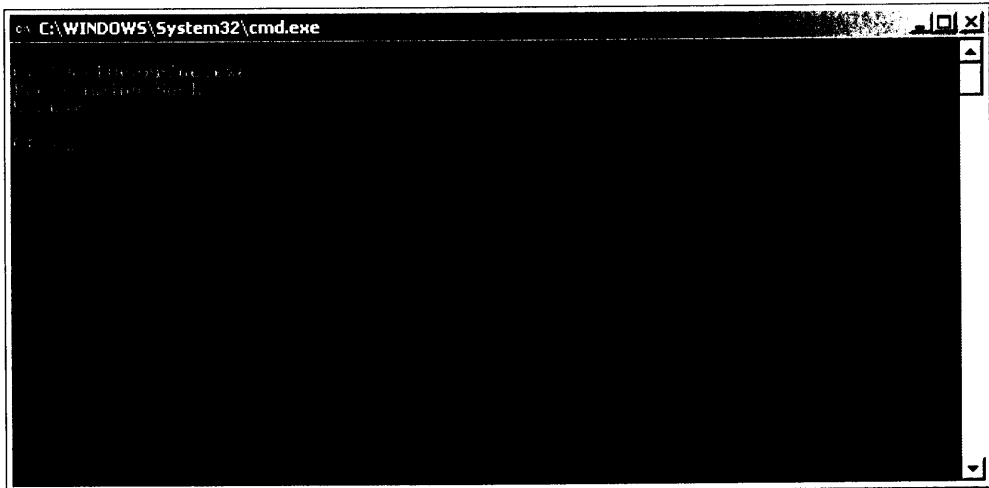


Figura 6.9. El título y la puntuación de su libro aparecen como se esperaba.

A continuación, sobrecargue el método `Title` creando una clase basada en la clase `Book`. Para crear una clase basada en otra clase (y que por tanto permite sobrecargar métodos), simplemente declare una clase de la forma habitual y ponga a continuación del nombre de la clase dos puntos y el nombre de la clase en la que quiere que se base. Añada el código del listado 6.11 a la aplicación.

Listado 6.11. Cómo sobrecargar métodos derivando la clase `Book`

```
class Wiley : Book
{
    new public string Title()
    {
        return "C# Bible";
    }
}
```

Este código crea una clase `Wiley` que hereda la clase `Book`. Ahora puede crear un nuevo método público llamado `Title`. Como ya se ha asignado a este método el mismo nombre que al definido en la clase `Book`, se sobrecarga el método `Title` aunque sigue disponiendo de acceso a los otros miembros dentro de la clase `Book`.



Ahora que ha sobrecargado el método `Title`, debe cambiar el método `Main()` para usar su nueva clase. Cambie su método `Main()` como se muestra en el listado 6.12.

Listado 6.12. Cómo modificar el método Main() para sobrecargar una clase

```
static void Main(string[] args)
{
    Wiley bc = new Wiley();
    Console.WriteLine(bc.Title());
    Console.WriteLine(bc.Rating());
}
```

En su método Main(), cambie la variable `bc` para crear la nueva clase `Wiley`. Como habrá adivinado, al llamar al método `Title`, el título del libro cambia de `Programming Book` a `C# Bible`. Fijese en que todavía tiene acceso al método `Rating`, que fue definido originalmente en la clase `Book`.

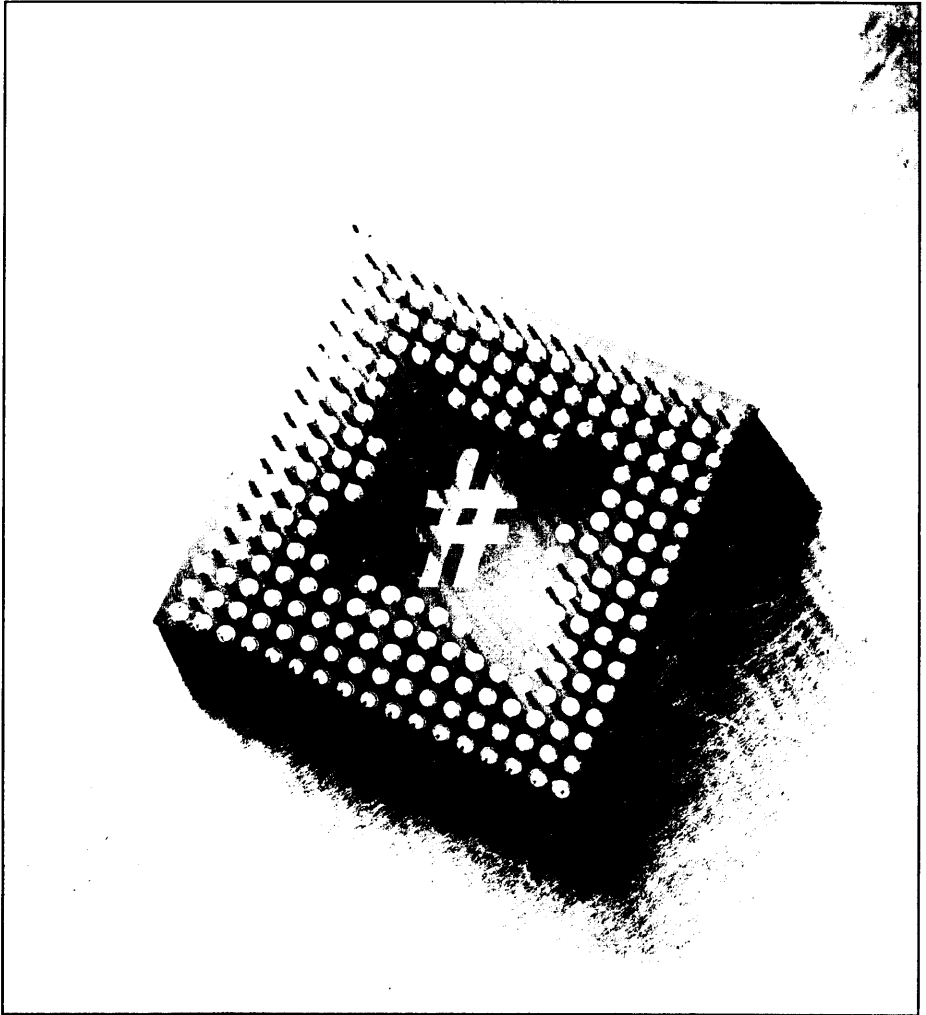
La sobrecarga de métodos dentro de una clase base es una excelente manera de cambiar la funcionalidad específica sin grandes problemas

Resumen

C# permite escribir métodos en sus clases de C#. Los métodos pueden ayudar a dividir el código en partes fáciles de entender y pueden brindar un lugar único en el que almacenar código que puede ser llamado varias veces.

Las funciones pueden recibir parámetros. Los parámetros de entrada contienen valores que han sido pasados a los métodos, pero sus valores no pueden cambiar. Los parámetros de salida tienen valores que les son asignados por un método y el valor asignado es visible para el elemento que hace la llamada. Los parámetros de referencia contienen valores que pueden ser proporcionados dentro de la función y además, su valor puede ser modificado por el método. Las matrices de parámetros permiten escribir métodos que toman un número variable de argumentos.

C# también permite sobrecargar métodos. Los métodos sobrecargados tienen el mismo nombre pero diferentes listas de parámetros. C# usa los parámetros proporcionados en una llamada para determinar qué método debe invocar cuando se ejecute el código.



7

Agrupación de datos usando estructuras

C# permite agrupar las variables en estructuras. Al definir una estructura para los datos, todo el grupo puede ser procesado con un solo nombre de estructura, sin importar el número de variables que contenga la estructura. El uso de una sola estructura facilita la manipulación de un conjunto de variables, en lugar de tener que seguir la pista de cada variable por separado. Una estructura puede contener campos, métodos, constantes, constructores, propiedades, indizadores, operadores y otras estructuras.

Las estructuras de C# son tipos de valor, no tipos de referencia. Esto significa que las variables de estructura contienen directamente los valores de las estructuras, en lugar de mantener una referencia a una estructura que se encuentra en otra parte de la memoria.

Algunas de las variables declaradas en el código C# pueden tener una relación lógica con otras variables ya declaradas. Suponga, por ejemplo, que quiere escribir un código que trabaje con un punto de la pantalla. Puede declarar dos variables para describir el punto:

```
int XCoordinateOfPoint;  
int YCoordinateOfPoint;
```

El punto tiene dos valores, la coordenada x y la coordenada y, que funcionan juntas para describir el punto.

Aunque puede escribir el código C# de esta manera, es bastante pesado. Los dos valores deben poder usarse en cualquier código que quiera trabajar con el punto. Si quiere que un método trabaje con el punto, tendrá que pasar los valores uno a uno:

```
void WorkWithPoint(int XCoordinate, int YCoordinate);  
void SetNewPoint(out int XCoordinate, out int YCoordinate);
```

La situación resulta incluso más complicada cuando varias variables trabajan juntas para describir una sola entidad. Por ejemplo, un empleado en una base de datos de recursos humanos puede tener variables que representen un nombre de pila, un apellido, una dirección, un número de teléfono y un salario actual. Controlar todas estas variables por separado y asegurarse de que todas se usan como un grupo puede volverse complicado.

Cómo declarar una estructura

Los contenidos de una estructura se declaran usando la palabra clave `struct`. Para dar nombre a una estructura use un identificador después de la palabra clave `struct`. La lista de variables que forman la estructura se encierra entre llaves a continuación del identificador de la estructura. Las declaraciones de miembro de la estructura suelen llevar antepuesta la palabra clave `public` para avisar al compilador de que sus valores deben ser públicamente accesibles a todo el código de la clase. Cada declaración de miembro termina con un punto y coma. La declaración de una estructura que defina un punto puede parecerse a lo siguiente:

```
struct Point  
{  
    public int X;  
    public int Y;  
}
```

En el ejemplo anterior, los miembros de la estructura, `X` e `Y`, tienen el mismo tipo. Sin embargo, esto no es obligatorio. Las estructuras también pueden estar formadas por variables de distintos tipos. El ejemplo anterior del empleado puede presentar este aspecto:

```
struct Employee  
{  
    public string FirstName;  
    public string LastName;  
    public string Address;  
    public string City;  
    public string State;  
    public ushort ZIPCode;  
    public decimal Salary;  
}
```

Como con todas las instrucciones de C#, sólo puede declararse una estructura desde el interior de una clase.

NOTA: C# no permite que una estructura tenga miembros con valores iniciales.

Los valores iniciales de los miembros de la estructura siguen las reglas de inicialización de valores descritas en un capítulo anterior. Los valores se inicializan con alguna representación del cero y las cadenas se vacían. C# no permite inicializar miembros de estructuras en el momento de declararse. Observe el error en el siguiente código:

```
struct Point
{
    public int X = 100;
    public int Y = 200;
}
```

Esta declaración produce estos errores del compilador:

```
error CS0573: 'Point.X': no se permiten inicializadores de
campo de instancia en las estructuras
error CS0573: 'Point.Y': no se permiten inicializadores de
campo de instancia en las estructuras
```

Puede usar un método especial llamado *constructor* para inicializar miembros de estructuras con valores distintos de cero. Más adelante, en este mismo capítulo, se examinarán los constructores.

Cómo usar estructuras en el código

Después de haber definido la estructura, puede usar su identificador como un tipo de variable, igual que si fuera un tipo `int` o `long`. Indique el identificador de la estructura, seguido de algún espacio en blanco y del identificador de la variable de estructura:

```
Point MyPoint;
```

Esta declaración declara una variable llamada `MyPoint` cuyo tipo es el de la estructura `Point`. Se puede usar esta variable igual que cualquier otra variable, incluso dentro de expresiones y como parámetro de un método.

El acceso a cada miembro de la estructura resulta tan sencillo como escribir el nombre del identificador de la variable de la estructura, un punto y a continuación el miembro de la estructura. El listado 7.1 muestra cómo se puede usar una estructura en el código.

Listado 7.1. Acceso a miembros de la estructura

```
class Listing7_1
{
    struct Point
    {
        public int X;
        public int Y;
    }

    public static void Main()
    {
        Point MyPoint;

        MyPoint.X = 100;
        MyPoint.Y = 200;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);
    }
}
```

El resultado de este ejemplo debería ser el siguiente:

```
100
200
```

Se puede asignar una variable de estructura a otra, siempre que las estructuras sean del mismo tipo. Cuando se asigna una variable de estructura a otra, C# asigna el valor de la variable de estructura que aparece antes del signo igual a los valores correspondientes de la estructura que aparece después del signo igual, como se puede observar en el listado 7.2.

Listado 7.2. Asignación de una variable de estructura a otra

```
class Listing7_2
{
    struct Point
    {
        public int X;
        public int Y;
    }

    public static void Main()
    {
        Point MyFirstPoint;
        Point MySecondPoint;

        MyFirstPoint.X = 100;
        MyFirstPoint.Y = 100;
        MySecondPoint.X = 200;
        MySecondPoint.Y = 200;
    }
}
```

```

System.Console.WriteLine(MyFirstPoint.X);
System.Console.WriteLine(MyFirstPoint.Y);

MyFirstPoint = MySecondPoint;

System.Console.WriteLine(MyFirstPoint.X);
System.Console.WriteLine(MyFirstPoint.Y);
}
}

```

El código anterior asigna el valor 100 a los miembros de `MyFirstPoint` y el valor de 200 a los miembros de `MySecondPoint`. Los valores de `MyFirstPoint` se escriben en la consola y luego los valores de la variable `MyFirstPoint` se copian en los valores de la variable `MySecondPoint`. Tras la asignación, los valores de `MyFirstPoint` se vuelven a escribir en la consola.

Si compila y ejecuta este código, obtendrá el resultado ilustrado en la figura 7.1.

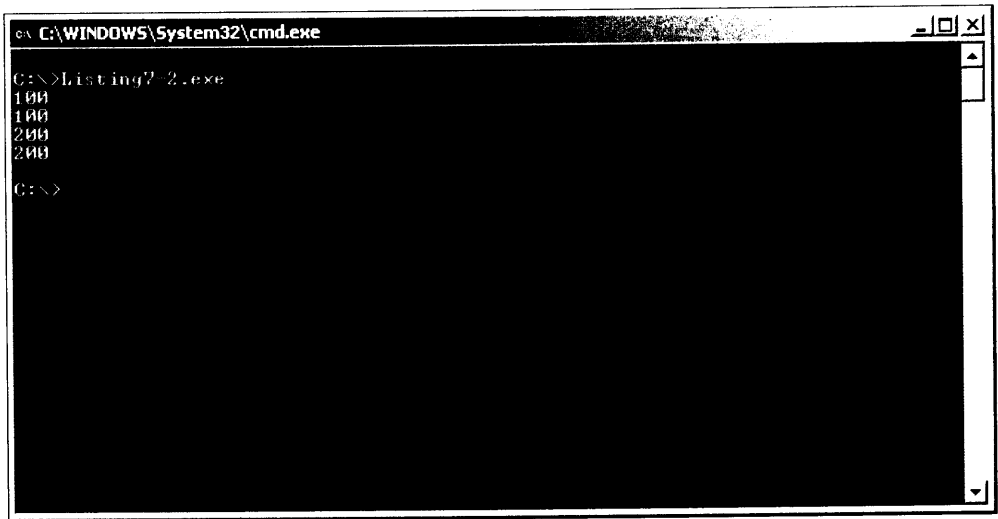


Figura 7.1. Asignación de una estructura a otra

Todos los valores de una estructura se sobrescriben en una asignación con los valores de la variable de estructura indicada después del signo igual.

Cómo definir métodos en estructuras

Además de variables, en las estructuras se pueden incluir métodos. Si necesita escribir un código que trabaje con los contenidos de una estructura, podría considerar la opción de escribir el método dentro de la misma estructura.

Cómo usar métodos constructores

Una estructura puede incluir un método especial llamado *constructor*. Un método constructor se ejecuta cuando se ejecuta en tiempo de ejecución una declaración de variable que usa el tipo de la estructura.

Las estructuras pueden tener varios constructores o ninguno. Las declaraciones de constructores de estructura son muy parecidas a las declaraciones de métodos de clase, con las siguientes excepciones:

- Los constructores no devuelven ningún valor. No se pueden usar palabras clave de tipo de devolución para escribir un constructor de estructura, ni siquiera `void`.
- Los identificadores de constructores tienen el mismo nombre que la estructura.
- Los constructores deben tener al menos un parámetro. `C#` no permite definir un constructor sin parámetros. `C#` siempre define un constructor por defecto sin parámetros por nosotros. Éste es el constructor que inicializa todos los miembros de la estructura a cero o su equivalente.

Una estructura puede definir más de un constructor, siempre que los constructores tengan diferentes listas de parámetros. El listado 7.3 muestra una estructura `Point` con dos constructores.

Listado 7.3. Constructores de estructuras

```
class Listing7_3
{
    struct Point
    {
        public int X;
        public int Y;

        public Point(int InitialX)
        {
            X = InitialX;
            Y = 1000;
        }

        public Point(int InitialX, int InitialY)
        {
            X = InitialX;
            Y = InitialY;
        }
    }

    public static void Main()
    {
        Point MyFirstPoint = new Point();
    }
}
```

```

Point MySecondPoint = new Point(100);
Point MyThirdPoint = new Point(250, 475);

System.Console.WriteLine(MyFirstPoint.X);
System.Console.WriteLine(MyFirstPoint.Y);

System.Console.WriteLine(MySecondPoint.X);
System.Console.WriteLine(MySecondPoint.Y);

System.Console.WriteLine(MyThirdPoint.X);
System.Console.WriteLine(MyThirdPoint.Y);
}
}

```

La figura 7.2 ilustra el resultado de compilar y ejecutar el código del listado 7.3.

```

C:\WINDOWS\System32\cmd.exe
C:\>Listing7-3.exe
0
0
100
1000
250
475
C:\>_

```

Figura 7.2. La estructura revela los valores predefinidos.

Tenga en cuenta los siguientes conceptos del listado 7.3:

- La estructura `Point` declara dos constructores. Uno recibe como argumento un solo número entero y el otro recibe dos números enteros. Ambos llevan antepuesta la palabra clave `public` por lo que su código es accesible al resto del código de la clase.
- El constructor con un parámetro entero asigna al miembro `X` de la estructura el valor del argumento entero y asigna al miembro `Y` de la estructura el valor `1.000`.
- El constructor con dos parámetros enteros asigna al miembro `X` de la estructura el valor del primer argumento entero y asigna al miembro `Y` de la estructura el valor del segundo argumento entero.

- El código declara tres variables de tipo `Point`. Cada una de ellas llama a uno de los constructores `Point`. La declaración de `MyFirstPoint` llama al constructor sin argumentos. Éste es el constructor por defecto que C# define para cada estructura. La declaración de `MySecondPoint` llama al constructor que tiene un argumento y la declaración de `MyThirdPoint` llama al constructor con dos argumentos.

Preste mucha atención a la sintaxis del listado 7.3, que invoca a un constructor de estructura. Si se quiere invocar a un constructor en una estructura, se debe emplear la palabra clave `new` seguida del nombre de la estructura y de los parámetros del constructor entre paréntesis. El valor de esa expresión se asigna a la variable que se está declarando. Observe la siguiente declaración:

```
Point MyThirdPoint = new Point(250, 475);
```

Esta declaración indica: "Crea una nueva estructura `Point` usando el constructor que tiene dos enteros. Asigna su valor a la variable `MyThirdPoint`". Debido a las reglas de asignación de estructuras anteriormente descritas, los miembros de la variable `MyThirdPoint` reciben los valores de los miembros de la nueva estructura. No es necesario hacer nada más con la nueva estructura creada cuando se llamó a `new`. El entorno común de ejecución (CLR) detecta que la estructura ya no se usa y se deshace de ella mediante el mecanismo de recolección de elementos no utilizados.

En el listado 7.3 también aparece la sintaxis del constructor sin parámetros:

```
Point MyFirstPoint = new Point();
```

Así se indica al compilador de C# que se quiere inicializar la estructura de la forma habitual. Se deben asignar valores a todos los miembros de una estructura antes de usarla, bien invocando su constructor sin parámetros o asignando explícitamente todos los campos de un valor. Observe el listado 7.4.

Listado 7.4. Si se usa una estructura antes de inicializarla se producen errores de compilación

```
class Listing7_4
{
    struct Point
    {
        public int X;
        public int Y;
    }

    public static void Main()
    {
        Point MyPoint;

        System.Console.WriteLine(MyPoint.X);
    }
}
```

```

        System.Console.WriteLine(MyPoint.Y);
    }
}

```

El código anterior es erróneo y al compilarlo el compilador de C# produce los siguientes mensajes de error:

```

error CS0170: Uso del campo 'X', posiblemente no asignado
error CS0170: Uso del campo 'Y', posiblemente no asignado
warning CS0649: El campo 'Listing7_4.Point.X' nunca se asigna y siempre tendrá el valor predeterminado 0
warning CS0649: El campo 'Listing7_4.Point.Y' nunca se asigna y siempre tendrá el valor predeterminado 0

```

Los mensajes de error avisan de que las llamadas a `WriteLine()` usan miembros de datos en la estructura, pero que a esos miembros de datos todavía no se les ha asignado valor. La variable `MyPoint` no ha sido inicializada con una llamada al constructor sin parámetros ni tampoco se han asignado valores explícitamente a sus miembros. C# no invoca al constructor sin parámetros a menos que se escriba la llamada en el código.

Éste es otro ejemplo de cómo el compilador de C# protege el código para evitar que se comporte de forma impredecible. Todas las variables deben inicializarse antes de ser usadas.

Cómo llamar a métodos desde estructuras

También se pueden escribir métodos en las estructuras. Estos métodos siguen las mismas reglas que los métodos de clase: deben especificar un tipo de devolución (`void`) y tener un identificador y una lista de argumentos, que puede estar vacía. Para llamar a un método en una estructura se usa la misma notación de puntos que para acceder a un método de clase. Observe el listado 7.5.

Listado 7.5. Cómo llamar a métodos de estructura

```

class Listing7_5
{
    struct Point
    {
        public int X;
        public int Y;

        public Point(int InitialX, int InitialY)
        {
            X = InitialX;
            Y = InitialY;
        }

        public bool IsAtOrigin()
        {

```

```

        if((X == 0) && (Y == 0))
            return true;
        else
            return false;
    }
}

public static void Main()
{
    Point MyFirstPoint = new Point(100, 200);
    Point MySecondPoint = new Point();

    if(MyFirstPoint.IsAtOrigin() == true)
        System.Console.WriteLine("MyFirstPoint is at the
origin.");
    else
        System.Console.WriteLine("MyFirstPoint is not at the
origin.");

    if(MySecondPoint.IsAtOrigin() == true)
        System.Console.WriteLine("MySecondPoint is at the
origin.");
    else
        System.Console.WriteLine("MySecondPoint is not at the
origin.");
}
}

```

La estructura `Point` del listado 7.5 declara un método llamado `IsAtOrigin`. El código de ese método comprueba los valores de los métodos de la estructura y devuelve `true` si las coordenadas del punto son (0, 0) y `false` en cualquier otro caso. El método `Main()` declara dos variables de tipo `Point`: a la variable `MyFirstPoint` se le asignan las coordenadas (100, 200) usando el constructor explícito y a la variable `MySecondPoint` se le asignan las coordenadas (0, 0) usando el constructor por defecto sin parámetros. En ese momento el método `Main()` llama al método `IsAtOrigin` con los dos puntos y escribe un mensaje basado en el valor devuelto por el método. Si se compila y ejecuta el código del listado 7.5, se obtiene el siguiente resultado en la consola:

```

MyFirstPoint is not at the origin.
MySecondPoint is at the origin.

```

Hay que asegurarse de prefijar los métodos con la palabra clave `public` si se quiere que sean accesibles por todo el resto del código de la clase.

Cómo definir propiedades en estructuras

Las propiedades de una estructura permiten leer, escribir y calcular valores usando descriptores de acceso. A diferencia de los campos, las propiedades no se

consideran variables; por tanto, no designan espacios de almacenamiento. Debido a esto, no pueden ser pasadas como parámetros `ref` o `out`.

El listado 7.6 incluye una propiedad dentro de la estructura `Point`.

Listado 7.6. Definición de una propiedad en el interior de una estructura

```
class Listing7_6
{
    struct Point
    {
        private int x;
        public int X
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        }
    }

    public static void Main()
    {
        int RetValue;
        Point MyPoint = new Point();

        MyPoint.X = 10;
        RetValue = MyPoint.X;
        System.Console.WriteLine(RetValue);
    }
}
```

Este código asigna un valor al miembro `X` de la estructura `Point` y luego devuelve este valor a la variable `RetValue`. El resultado del listado 7.6 se ilustra en la figura 7.3.

El uso de propiedades es una forma excelente de leer, escribir y calcular datos dentro de una estructura. No hace falta incluir métodos voluminosos para que realicen los cálculos y se puede definir cómo y cuándo pueden actuar los descriptores de acceso `get` y `set`.

Cómo definir indizadores en estructuras

Los indizadores son objetos que permiten indizar una estructura de forma muy parecida a una matriz. Con un indizador, se pueden declarar varias estructuras al mismo tiempo y hacer referencia a cada estructura usando un número de índice.

El listado 7.7, que declara una estructura llamada `MyStruct` que contiene una cadena y un índice, lo demuestra.

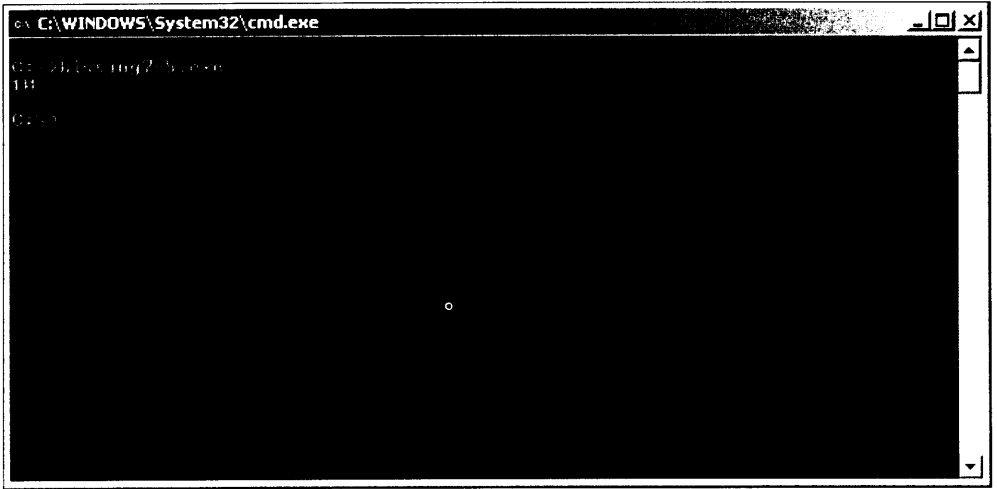


Figura 7.3. Definición de una propiedad en el interior de una estructura

Listado 7.7. Cómo incluir un indizador en una estructura

```
class Listing7_7
{
    struct MyStruct
    {
        public string []data ;
        public string this [int index]
        {
            get
            {
                return data[index];
            }
            set
            {
                data[index] = value;
            }
        }
    }
}

public static void Main()
{
    int x;
    MyStruct ms = new MyStruct();
    ms.data = new string[5];
    ms[0] = "Brian D Patterson";
    ms[1] = "Aimee J Patterson";
    ms[2] = "Breanna C Mounts";
    ms[3] = "Haileigh E Mounts";
    ms[4] = "Brian W Patterson";
}
```

```

    for (x=0;x<5;x++)
        System.Console.WriteLine(ms[x]);
    }
}

```

Como se puede ver, este ejemplo crea un nuevo objeto `MyStruct` y asigna a los miembros `data` el valor 5, lo que indica que se usan cinco copias de esta estructura. Para hacer referencia a cada copia de esta estructura se usa un número indizador (de 0 a 5) y se almacenan los nombres dentro de la estructura. Para asegurarse de que todos los datos permanecen intactos, se aplica un simple bucle a los posibles números de índice y se escribe el resultado en la consola.

En la figura 7.4 se ilustra el resultado del listado 7.7.

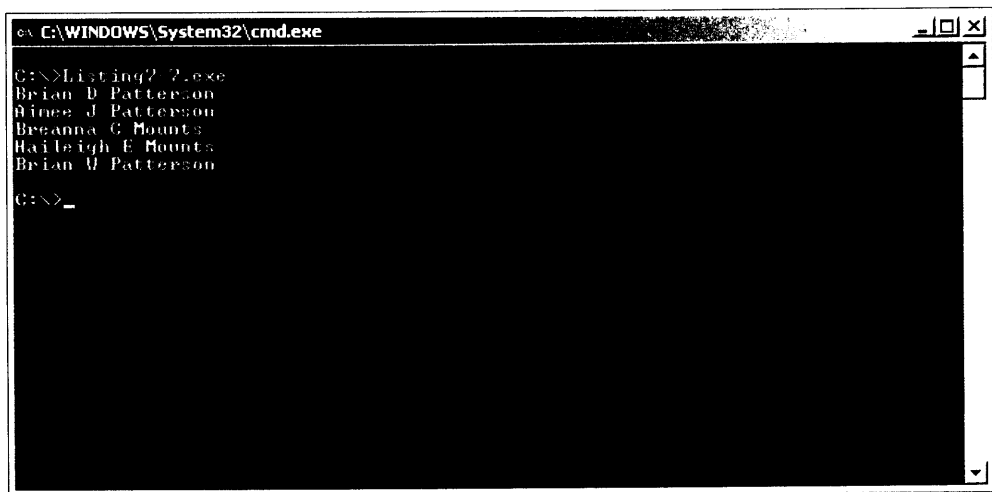


Figura 7.4. Para devolver fácilmente los datos se incluye un indizador dentro de la estructura

Un indizador en una estructura puede ser muy útil cuando se trabaja con grandes cantidades de datos del mismo tipo. Por ejemplo, si se va a leer la información de una dirección desde una base de datos, éste es un excelente lugar para almacenarla. Todos los campos se mantienen mientras se proporciona un mecanismo para acceder fácilmente a cada dato de los registros.

Cómo definir interfaces en estructuras

Las interfaces son un modo de asegurarse de que cualquiera que use la clase cumple con todas las reglas impuestas para hacerlo. Éstas pueden incluir la implementación de ciertos métodos, propiedades y eventos. Cuando se expone una interfaz, sus usuarios deben heredarla y al hacerlo están obligados a crear ciertos métodos y así sucesivamente. Esto asegura que la clase y/o estructura se use de forma correcta.

También se puede incluir una interfaz dentro de una estructura. El listado 7.8 muestra cómo implementar correctamente una interfaz.

Listado 7.8. Cómo implementar una interfaz en una estructura

```
class Listing7_8
{
    interface IInterface
    {
        void Method();
    }
    struct MyStruct : IInterface
    {
        public void Method()
        {
            System.Console.WriteLine("Structure Method");
        }
    }

    public static void Main()
    {
        MyStruct DemoStructure = new MyStruct();
        DemoStructure.Method();
    }
}
```

Este código crea una interfaz llamada `IInterface`. Esta interfaz contiene la definición de un método llamado `Method`. Se crea una estructura y al final de su nombre se incluyen dos puntos seguidos del nombre de la interfaz que desea derivar. El método, que simplemente escribe una línea de texto en la consola, se incluye en la estructura. En la figura 7.5 se ilustra el resultado del programa.

Para demostrar lo importante que es la interfaz, si elimine las cuatro líneas que componen el método `Method` en la estructura `MyStruct` y vuelve a compilar el programa, obtendrá el siguiente mensaje de error:

```
Class1.cs(8,9): error CS0535: 'Listing7_8.MyStruct' no implementa el miembro de interfaz Listing7_8.IInterface.Method()'
```

El compilador de `C#` determinó que no implementamos todos los métodos estipulados por la interfaz. Como no se implementó el método correcto, el programa no se pudo compilar, indicando de esta forma que no se cumplen todas las reglas.

Cómo usar los tipos simples de `C#` como estructuras

Los tipos primitivos (`int`, `uint`, `long` y similares) descritos en un capítulo anterior en realidad se implementan como estructuras en el CLR de `.NET`. La

tabla 7.1 enumera las palabras clave con valor variable y los nombres de las estructuras .NET que actualmente los implementan.

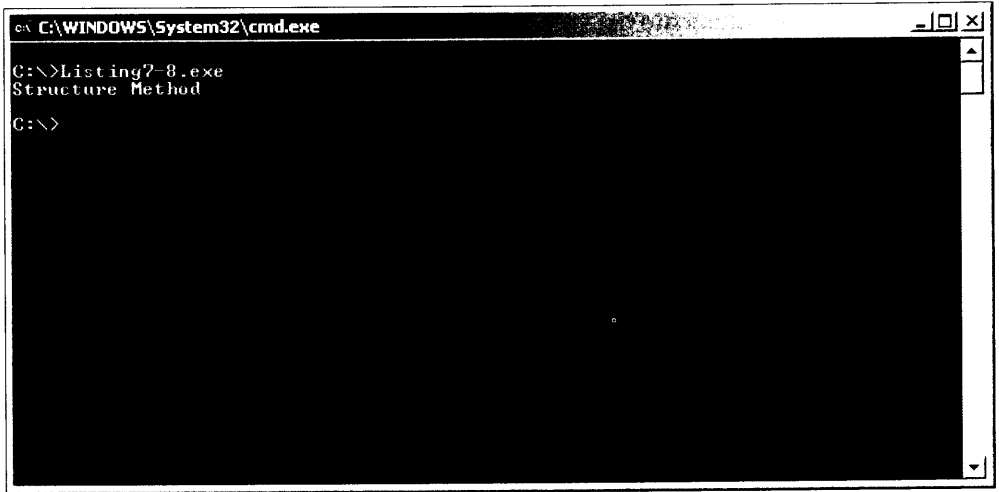


Figura 7.5. Cómo implementar un interfaz en una estructura

Tabla 7.1. Nombres de estructura .NET para tipos de valor

Palabra clave de C#	Nombre de estructura .NET
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Este esquema es parte de lo que hace que el código C# sea compatible con otros lenguajes .NET. Los valores C# se asigna a las estructuras .NET que pue-

de usar cualquier lenguaje .NET, porque el CLR puede usar cualquier estructura .NET. La asignación de las palabras claves de C# con estructuras .NET también permite que las estructuras usen técnicas como la sobrecarga de operadores para definir el comportamiento del valor cuando se usan en una expresión con un operador. Se analizará la sobrecarga de operadores cuando se traten las clases C#.

Si lo desea, puede usar los nombres reales de estructura .NET en lugar de las palabras clave de C#. El listado 7.9 muestra el aspecto que tendría el listado 7.5 si se escribiera usando los nombres de estructuras .NET.

Listado 7.9. Cómo usar los nombres de tipo de estructura .NET

```
class Listing7_9
{
    struct Point
    {
        public System.Int32 X;
        public System.Int32 Y;

        public Point(System.Int32 InitialX, System.Int32 InitialY)
        {
            X = InitialX;
            Y = InitialY;
        }

        public System.Boolean IsAtOrigin()
        {
            if((X == 0) && (Y == 0))
                return true;
            else
                return false;
        }
    }

    public static void Main()
    {
        Point MyFirstPoint = new Point(100, 200);
        Point MySecondPoint = new Point();

        if(MyFirstPoint.IsAtOrigin() == true)
            System.Console.WriteLine("MyFirstPoint is at the origin.");
        else
            System.Console.WriteLine("MyFirstPoint is not at the
origin.");

        if(MySecondPoint.IsAtOrigin() == true)
            System.Console.WriteLine("MySecondPoint is at the
origin.");
        else
            System.Console.WriteLine("MySecondPoint is not at the
origin.");
    }
}
```

Resumen

Las estructuras permiten agrupar un conjunto de variables con un solo nombre. Las variables pueden declararse usando el identificador de estructura.

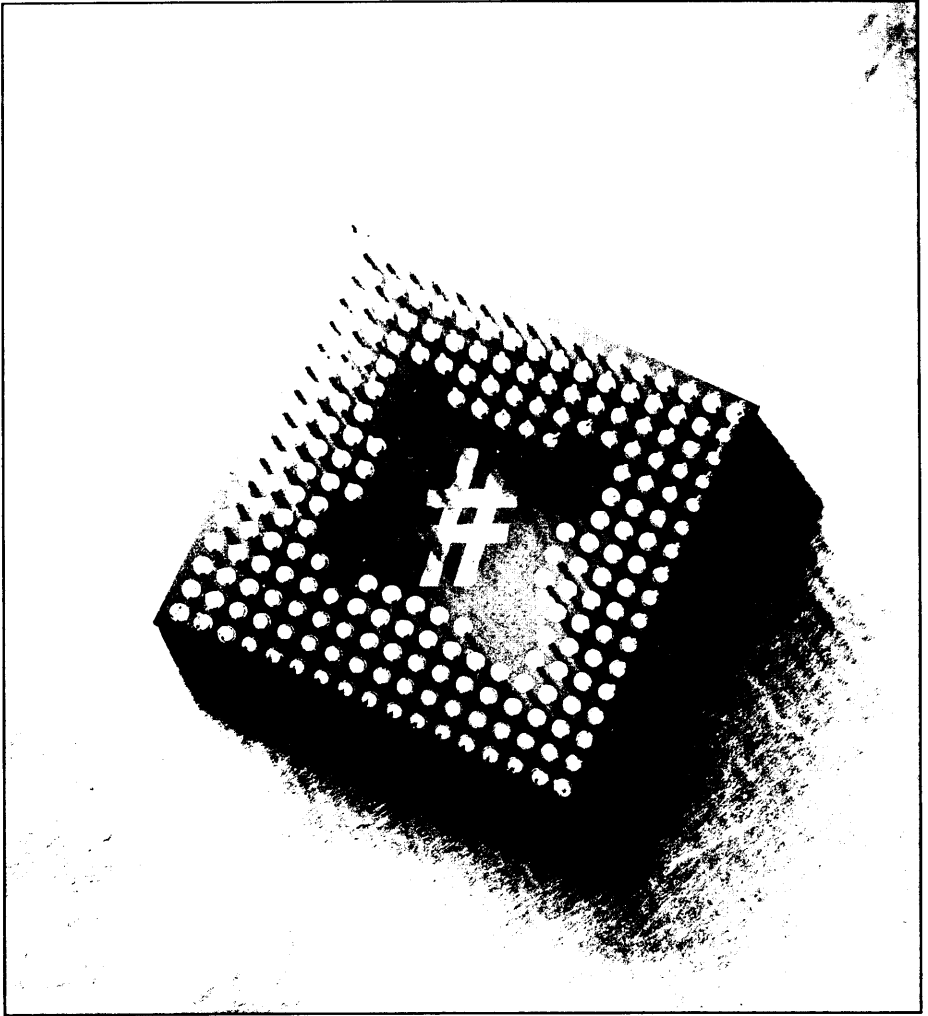
Las estructuras se declaran usando la palabra clave de C# `struct`. Todas las estructuras de C# tienen un nombre y una lista de miembros de datos. C# no pone ningún límite al número de miembros de datos que puede contener una estructura.

Se accede a los miembros de estructuras mediante la notación `StructName.MemberName`. Los miembros se pueden usar en cualquier parte donde esté permitido usar su tipo de datos, incluyendo expresiones y parámetros de métodos.

Las estructuras pueden implementar tanto métodos como variables. Los miembros de estructuras se invocan usando la notación `StructName.MethodName` y se usan igual que los nombres de método de clase. Las estructuras también implementan métodos especiales llamados constructores, que inicializan la estructura con un estado conocido antes de usar la estructura.

Los tipos de valor de C# son asignados a estructuras definidas por el CLR de .NET. Esto es lo que permite que otros códigos de .NET usen los datos. Todas las variables son compatibles con CLR de .NET porque las variables se definen usando estructuras compatibles con el CLR.

Parte II
Programación
orientada a
objetos con C#



8 Escribir código orientado a objetos

Los lenguajes de programación siempre se han diseñado en torno a dos conceptos fundamentales: los datos y el código que opera sobre los datos. Los lenguajes han evolucionado a lo largo del tiempo para cambiar el modo en que estos dos conceptos interactúan. En un principio, lenguajes como Pascal y C invitaban a los programadores a escribir software que tratara al código y a los datos como dos cosas separadas, sin ninguna relación. Este enfoque dio a los programadores la libertad, pero también la obligación, de elegir el modo en que su código gestiona los datos.

Además, este enfoque obligaba al programador a traducir el mundo real que se quería modelar usando software a un modelo específico para ordenadores que usara datos y código.

Los lenguajes como Pascal y C se construyeron en torno al concepto de *procedimiento*. Un procedimiento es un bloque de código con nombre, exactamente igual que los actuales métodos de C#. El estilo de software desarrollado usando estos lenguajes se llama *programación procedural*.

En la programación procedural, el programador escribe uno o más procedimientos y trabaja con un conjunto de variables independientes definidas en el programa. Todos los procedimientos pueden verse desde cualquier parte del código de la aplicación y todas las variables pueden ser manipuladas desde cualquier parte del código.

En los años 90, la programación procedural dio paso a lenguajes como Smalltalk y Simula, que introdujeron el concepto de objeto. Los inventores de estos lenguajes se dieron cuenta de que el ser humano no expresa ideas en términos de bloques de código que actúan sobre un grupo de variables; en su lugar, expresan ideas en términos de objetos. Los *objetos* son entidades que tienen un conjunto de valores definidos (el estado del objeto) y un conjunto de operaciones que pueden ejecutarse sobre ese objeto (los *comportamientos* del objeto). Por ejemplo, imagine un cohete espacial.

Un cohete espacial tiene estados, como la cantidad de combustible o el número de pasajeros a bordo, y comportamientos, como "despegar" y "aterrizar". Además, los objetos pertenecen a clases. Los objetos de la misma clase tienen el mismo estado y el mismo conjunto de comportamientos. Un objeto es un caso concreto de una clase. El cohete espacial es una clase, mientras que el cohete espacial llamado Discovery es un objeto, un caso concreto de la clase cohete espacial.

NOTA: En realidad, ni siquiera en la programación procedural son visibles todos los procedimientos ni todas las variables. Igual que en C#, los lenguajes procedurales tienen *reglas de ámbito* que controlan la visibilidad del código y de los datos. Por lo general podemos hacer visibles los procedimientos y los datos (a los que nos referiremos en este capítulo como elementos) en el procedimiento, archivo fuente, aplicación o nivel externo. El nombre de cada ámbito es autoexplicativo. Un elemento visible en el nivel de procedimiento sólo es accesible dentro del procedimiento en el que se define. No todos los lenguajes permiten crear procedimientos dentro de otros procedimientos. Un elemento visible en el nivel de archivo fuente es visible dentro del archivo en el que se define el elemento. En el nivel de aplicación, el elemento es visible desde cualquier parte de código en la misma aplicación. En el nivel externo, el elemento es visible desde cualquier parte de código en cualquier aplicación.

El punto principal es que, en programación procedural, la interacción entre datos y código está controlada por los detalles de implementación, como el archivo fuente en el que se define una variable. Una vez que se decide hacer visible una variable fuera de sus propios procedimientos, no se obtiene ayuda para proteger el acceso a esa variable. En aplicaciones grandes con varios miles de variables, esta falta de protección suele acarrear fallos difíciles de encontrar.

El desarrollo de software orientado a objetos tiene dos claras ventajas sobre el desarrollo de software procedural. La primera ventaja es que se puede especificar lo que debe hacer el software y cómo lo hará usando un vocabulario familiar a los usuarios sin preparación técnica. El software se estructura usando objetos. Estos objetos pertenecen a clases con las que el usuario del mundo de los negocios, al

que está destinado, está familiarizado. Por ejemplo, durante el diseño de software ATM, se usan clases como CuentaBancaria, Cliente, Presentación y similares.

Esto reduce el trabajo necesario para traducir una situación del mundo real al modelo de software y facilita la comunicación con la gente ajena al software y que está interesada en el producto final. Este modo más sencillo de diseñar software ha conducido a la aparición de un estándar para describir el diseño del software orientado a objetos. Este lenguaje es el Lenguaje unificado de modelado o UML.

La segunda ventaja del software orientado a objetos se demuestra durante la implementación. El hecho de que ahora podemos tener ámbitos de nivel de clases, permite ocultar variables en las definiciones de clase. Cada objeto tendrá su propio conjunto de variables y estas variables por lo general solamente serán accesibles mediante las operaciones definidas por la clase. Por ejemplo, las variables que contienen un estado del objeto CuentaBancaria sólo serán accesibles llamando a la operación Retirada() o Depósito() asociada a ese objeto. Un objeto CuentaBancaria (o cualquier otro objeto) no tiene acceso a otro estado privado del objeto CuentaBancaria, como el balance. A este principio se le llama *encapsulación*.

El desarrollo de software orientado a objetos se fue haciendo más popular a medida que los programadores adoptaban este nuevo modo de diseñar software. C# es un lenguaje orientado a objetos y su diseño garantiza que los programadores de C# sigan los conceptos correctos de la programación orientada a objetos.

NOTA: SmallTalk, Java y C# son lenguajes orientados a objetos puros porque no se puede escribir un programa sin usar objetos. A otros lenguajes, como C y Pascal, se les llama lenguajes procedurales o no orientados a objetos porque no disponen de compatibilidad integrada que permita crear objetos. Existe un tercer tipo de lenguaje híbridos, como C++, en los que se puede elegir si usar o no objetos. Bjarne Stroustrup, el inventor de C++, decidió no obligar a los programadores de C++ a usar objetos porque C++ también era una versión mejorada del lenguaje de programación C. La compatibilidad con el código C existente ayudaba a que C++ se convirtiera en un lenguaje importante.

En este capítulo se estudian los conceptos que componen un lenguaje orientado a objetos, empezando por sus componentes esenciales (las clases y objetos), hasta los términos más avanzados (abstracción, tipos de datos abstractos, encapsulación, herencia y polimorfismo).

Se tratarán los conceptos básicos y se procurará evitar los detalles específicos sobre cómo están implementados estos conceptos en C#. Estos detalles específicos se tratarán en capítulos posteriores.

Clases y objetos

En primer lugar, esta sección vuelve a tratar la diferencia entre un objeto y una clase. Este libro usa mucho estos dos términos y es importante distinguir entre ellos. Una *clase* es una colección de código y de variables. Las clases gestionan el estado, en forma de las variables que contienen, y comportamientos, en forma de los métodos que contienen. Sin embargo, una clase sólo es una plantilla. Nunca se crea una clase en el código. En su lugar, se crean objetos. Por ejemplo, `CuentaBancaria` es una clase con una variable que contiene el saldo de la cuenta y los métodos `Retirada()`, `Depósito()` y `MostrarSaldo()`.

Los *objetos* son casos concretos de una clase. Los objetos se construyen usando una clase como plantilla. Cuando se crea un objeto, éste gestiona su propio estado. El estado de un objeto puede ser diferente del estado de otro objeto de la misma clase. Imagine una clase que define a una persona. Una clase persona va a tener estado (quizás, una cadena que representa el nombre propio de la persona) y comportamiento, mediante métodos como `IrATrabajar()`, `Comer()` e `IrADormir()`. Se pueden crear dos objetos de la clase persona, cada uno con un estado diferente, como se muestra en la figura 8.1. La figura 8.1 muestra la clase persona y dos objetos persona: uno con el nombre propio "Alice" y otro con el nombre propio "Bob". El estado de cada objeto se almacena en un conjunto de variables diferentes. Vuelva a leer la frase anterior. Esta frase contiene un punto esencial para comprender el funcionamiento de la programación orientada a objetos. Un lenguaje admite objetos cuando no se necesita crear un código especial para disponer de un conjunto de variables cada vez que se crea un objeto diferente.

NOTA: Si un lenguaje admite la gestión automática del estado dentro de objetos pero carece de las otras características descritas en esta sección, suele recibir el nombre de lenguaje basado en objetos. Visual Basic 6 admite objetos, pero no admite la herencia de implementación; por tanto, no se le considera como un auténtico lenguaje orientado a objetos. Auténticos lenguajes orientados a objetos son SmallTalk, Java y C#.

Terminología del diseño de software orientado a objetos

Se encontrará con muchos términos cuando lea libros sobre el desarrollo de software orientado a objetos y probablemente se encuentre con muchos de estos términos cuando trabaje con código C#. Algunos de los términos más usados son:

- Abstracción
- Tipos de datos abstractos
- Encapsulación
- Herencia
- Polimorfismo

Las siguientes secciones definen cada uno de estos términos con detalle.

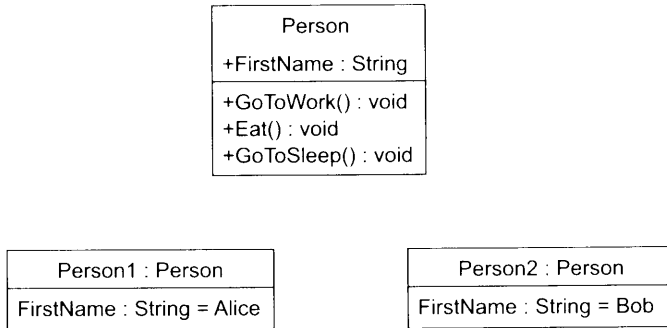


Figura 8.1. Esta clase persona tiene dos objetos persona.

Abstracción

Es importante darse cuenta de que el objetivo de la programación no es reproducir todos los aspectos posibles del mundo real de un concepto dado. Por ejemplo, cuando se programa una clase `Person`, no se intenta modelar todo lo que se conoce sobre una persona. En su lugar, se trabaja dentro del contexto de la aplicación específica. Sólo se modelan los elementos que son necesarios para esa aplicación. Algunas características de una persona, como la nacionalidad, pueden existir en el mundo real, pero se omiten si no son necesarias para la aplicación en cuestión. Una persona en una aplicación bancaria estará interesada en aspectos diferentes de los de, por ejemplo, una persona en un juego de acción. Este concepto recibe el nombre de *abstracción* y es una técnica necesaria para el manejo de los conceptos infinitamente complejos del mundo real. Por tanto, cuando se haga preguntas sobre objetos y clases, tenga siempre en cuenta que debe hacerse estas preguntas en el contexto de una aplicación específica.

Tipos de datos abstractos

Los tipos de datos abstractos fueron el primer intento de determinar el modo en que se usan los datos en programación. Los tipos de datos abstractos se crearon porque en el mundo real los datos no se componen de un conjunto de varia-

bles independientes. El mundo real está compuesto de conjuntos de datos relacionados. El estado de una persona para una aplicación determinada puede consistir en, por ejemplo, nombre, apellidos y edad. Cuando se quiere crear una persona en un programa, lo que se quiere crear es un conjunto de estas variables. Un tipo de datos abstractos permite presentar tres variables (dos cadenas y un número entero) como una unidad y trabajar cómodamente con esta unidad para contener el estado de una persona, como se ve en este ejemplo:

```
struct Person
{
    public String FirstName;
    public String LastName;
    public int Age;
}
```

Cuando se asigna un tipo de datos a una variable del código, se puede usar un tipo de datos *primitivo* o un tipo de datos *abstracto*. Los tipos de datos primitivos son tipos que C# reconoce en cuanto se instala. Tipos como `int`, `long`, `char` y `string` son tipos de datos primitivos de C#.

Los tipos de datos abstractos son tipos que C# no admite cuando se instalan. Antes de poder usar un tipo de datos abstracto hay que declararlo en el código. Los tipos de datos abstractos se definen en el código en lugar de hacerlo en el compilador de C#.

Por ejemplo, imagine la estructura (o clase) `Person`. Si escribe código C# que usa una estructura (o clase) `Person` sin escribir código que le indique al compilador de C# a qué se parece una estructura (o clase) `Person` y cómo se comporta, se obtiene un error de compilación. Observe el siguiente código:

```
class MyClass
{
    static void Main()
    {
        Person MyPerson;

        Person.FirstName = "Malgoska";
    }
}
```

Si se compila este código el compilador de C# genera el siguiente error:

```
error CS0234: El tipo o el nombre del espacio de nombres
'Person' no existe en la clase o el espacio de nombres
'MyClass' (¿falta una referencia de ensamblado?)
```

El problema de este código es que el tipo de datos `Person` no es un tipo de datos primitivo y no está definido por el lenguaje C#. Como no es un tipo primitivo, el compilador de C# considera que se trata de un tipo de datos abstracto y revisa el código buscando la declaración de un tipo de datos `Person`. Sin embargo, como el compilador de C# no puede encontrar información sobre el tipo de

datos abstracto `Person` genera un error. Tras definir un tipo de datos abstracto, puede usarse en el código `C#` exactamente igual que si fuera un tipo de datos primitivo. Las estructuras y las clases de `C#` son ejemplos de tipos de datos abstractos. Una vez que se ha definido una estructura (o clase), se pueden usar las variables de ese tipo dentro de otra estructura (o clase). La estructura `LearningUnit`, por ejemplo, contiene dos variables `Person`:

```
struct LearningUnit
{
    public Person Tutor;
    public Person Student;
}
```

Encapsulación

Mediante la encapsulación, los datos se ocultan, o se *encapsulan*, dentro de una clase y la clase implementa un diseño que permite que otras partes del código accedan a esos datos de forma eficiente. Imagine la encapsulación como un envoltorio protector que rodea a los datos de las clases de `C#`.

Cómo ejemplo de encapsulación, observe la estructura `Point` con la que trabajó en un capítulo anterior.

```
struct Point
{
    public int X;
    public int Y;
}
```

Los miembros de datos de esta estructura son públicos, lo que permite que cualquier parte de código que acceda a la estructura acceda también a los miembros de datos. Como cualquier parte de código puede acceder a los miembros de datos, el código puede asignar a los valores de los miembros de datos cualquier valor que pueda representarse en un valor `int`.

No obstante, puede surgir un problema al permitir que los clientes asignen los valores de los miembros de datos directamente. Supongamos que se usa la estructura `Point` para representar una pantalla de ordenador con una resolución de 800 x 600. En ese caso, sólo tiene sentido permitir al código que asigne a `X` valores entre 0 y 800 y a `Y` valores entre 0 y 600. No obstante, con el acceso público a los miembros de datos, no hay nada que impida al código asignar a `X` el valor 32.000 y a `Y` el valor 38.000. El compilador de `C#` lo permite porque esos valores son posibles en un entero. El problema es que no tiene sentido permitir valores tan elevados.

La encapsulación resuelve este problema. Básicamente, la solución está en marcar los miembros de datos como privados, para que el código no pueda acceder a los miembros de datos directamente. A continuación puede escribir métodos en una clase de punto como `SetX()` y `SetY()`. Los métodos `SetX()` y `SetY()`

pueden asignar los valores y también pueden contener código que genere un error si se trata de llamar a `SetX()` o a `SetY()` con parámetros con valores demasiado grandes. La figura 8.2 muestra el posible aspecto de una clase `Point`.

Point
-X : int
-Y : int
+SetX() : bool
+SetY() : bool
+GetX() : int
+GetY() : int

Figura 8.2. Las variables de miembros en la clase `Point` están encapsuladas.

NOTA: El signo menos delante de los miembros de datos es una notación UML que indica que los miembros tienen una visibilidad privada. El signo más delante de los miembros de datos es una notación UML que indica que los miembros tienen una visibilidad pública.

La técnica consistente en marcar como privados los miembros de datos resuelve el problema de que el código establezca sus valores directamente. Si los miembros de datos son privados, sólo la propia clase puede verlos y cualquier otro código que intente acceder a los miembros de datos genera un error del compilador.

En lugar de acceder directamente a los miembros de datos, la clase declara métodos públicos llamados `SetX()` y `SetY()`. El código que quiere asignar los valores de los puntos `X` e `Y` llama a estos métodos públicos. Estos métodos pueden aceptar el nuevo valor de las coordenadas y un parámetro, pero también pueden comprobar los nuevos valores para asegurarse de que están dentro de los límites adecuados. Si el nuevo valor está fuera de los límites, el método devuelve un error. Si el nuevo valor está dentro de los límites, el método puede establecer un nuevo valor. El siguiente seudo código muestra cómo se puede implementar el método `SetX()`:

```
bool SetX(int NewXValue)
{
    if(NewXValue is out of range)
        return false;
    X = NewXValue;
    return true;
}
```

Este código ha encapsulado el miembro de datos de la coordenada `X` y permite a los invocadores asignar su valor a la vez que impide que le asignen un valor no válido.

Cómo los valores de las coordenadas `X` e `Y` en este diseño son privados, las otras partes de código no pueden examinar sus valores actuales. La accesibilidad

privada en la programación orientada a objetos evita que las partes que realizan la llamada lean el valor actual o guarden el nuevo valor. Para exponer estas variables privadas, se pueden implementar métodos como `GetX()` y `GetY()` que devuelven el valor actual de las coordenadas.

En este diseño, la clase encapsula los valores de X e Y aunque permite que otras partes del código lean y escriban sus valores. La encapsulación proporciona otra ventaja: el método que realiza el acceso impide que se asignen valores absurdos a los miembros de datos.

Herencia

Algunas clases, como la clase `Point`, se diseñan partiendo de cero. El estado y los comportamientos de la clase se definen en ella misma. Sin embargo, otras clases toman prestada su definición de otra clase. En lugar de escribir otra clase de nuevo, se pueden tomar el estado y los comportamientos de otra clase y usarlos como punto de partida para la nueva clase. A la acción de definir una clase usando otra clase como punto de partida se le llama *herencia*.

Herencia simple

Supongamos que estamos escribiendo código usando la clase `Point` y nos damos cuenta de que necesitamos trabajar con puntos tridimensionales en el mismo código. La clase `Point` que ya hemos definido modela un punto en dos dimensiones y no podemos usarlo para definir un punto tridimensional. Decidimos que hace falta escribir un punto tridimensional llamado `Point3D`. Se puede diseñar la clase de una de estas dos formas:

- Se puede escribir la clase `Point3D` partiendo de cero, definiendo miembros de datos llamados X, Y y Z y escribiendo métodos que lean y escriban los miembros de datos.
- Se puede derivar de la clase `Point`, que ya implementa los miembros X e Y. Heredar de la clase `Point` proporciona todo lo necesario para trabajar con los miembros X e Y, por lo que todo lo que hay que hacer en la clase `Point3D` es implementar el miembro Z. La figura 8.3 muestra el aspecto que podría tener la clase `Point3D` en UML.

NOTA: La notación UML que indica que los miembros tienen visibilidad protegida es el símbolo de la libra delante de los miembros de datos. Visibilidad protegida significa que la visibilidad es pública para las clases derivadas y privadas para todas las demás clases.

La figura 8.3 refleja la *herencia simple*. La herencia simple permite que una clase se derive de una sola clase base. Este tipo herencia también es conocido

como derivar una clase de otra. Parte del estado y del comportamiento de la clase `Point3D` se derivan de la clase `Point`.

En este caso, la clase usada como punto de partida recibe el nombre de *clase base* y la nueva clase es la *clase derivada*. En la figura 8.3 la clase base es `Point` y la clase derivada es `Point3D`.

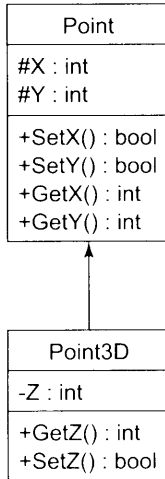


Figura 8.3. La clase `Point3D` hereda los métodos y las variables de la clase `Point`.

Derivar una clase de otra automáticamente permite que los miembros de datos públicos (y protegidos) y a los métodos públicos (y protegidos) de la clase base estén disponibles para la clase derivada. Como los métodos `GetX()`, `GetY()`, `SetX()` y `SetY()` están marcados como públicos en la clase base, están automáticamente disponibles para las clases derivadas. Esto significa que la clase `Point3D` dispone de los métodos públicos `GetX()`, `GetY()`, `SetX()` y `SetY()` ya que se derivaron de la clase base `Point`. Una sección de código puede crear un objeto de tipo `Point3D` y llamar a los métodos `GetX()`, `GetY()`, `SetX()` y `SetY()`, aunque los métodos no estén implementados explícitamente en esa clase. Se derivaron de la clase base `Point` y pueden ser usados en la clase derivada `Point3D`.

Herencia múltiple

Algunos lenguajes orientados a objetos también permiten la *herencias múltiple*, lo que permite que una clase se derive de más de una clase base. `C#` sólo permite herencias simples.

Esta restricción se debe a que el CLR de `.NET` no admite clases con varias clases base, principalmente porque otros lenguajes `.NET`, como `Visual Basic`, no admiten por sí mismos herencias múltiples. Los lenguajes que admiten herencia múltiple, como `C++`, también han evidenciado la dificultad de usar correctamente la herencia múltiple.

NOTA: Si se quiere usar la herencia simple (por ejemplo, para que una clase `RadioReloj` herede las clases `Despertador` y `Radio`) se puede obtener un comportamiento muy parecido al deseado mediante la *contención*. La contención incrusta una variable miembro de la clase de la que se quiere derivar. En este caso, esta técnica solicita añadir una variable `Despertador` y una variable `Radio` para la clase `RadioReloj` y delega la funcionalidad en la variable miembro adecuada. Esta técnica también se puede utilizar en herencias simples, pero es el único medio que hay para simular herencias múltiples en .NET (a menos que Microsoft nos sorprenda añadiendo herencias múltiples en una versión posterior).

Polimorfismo

La herencia permite que una clase derivada redefina el comportamiento especificado para una clase base. Supongamos que creamos una clase base `Animal`. Hacemos esta clase base abstracta porque queremos codificar animales genéricos siempre que sea posible, pero también crear animales específicos, como un gato y un perro. El siguiente fragmento de código muestra cómo declarar la clase con su método abstracto.

```
Abstract class Animal
{
    public abstract void MakeNoise();
}
```

Ahora se pueden derivar animales específicos, como `Cat` y `Dog`, a partir de la clase base abstracta `Animal`:

```
class Cat : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Meow!");
    }
}

class Dog : Animal
{
    public override void MakeNoise()
    {
        Console.WriteLine("Woof!");
    }
}
```

Observe que cada clase tiene su propia implementación del método `MakeNoise()`. Ahora la situación es la indicada para el polimorfismo. Como se aprecia en la figura 8.4, tenemos una clase base con un método que está anula-

do en dos (o más) clases derivadas. El *polimorfismo* es la capacidad que tiene un lenguaje orientado a objetos de llamar correctamente al método anulado en función de qué clase lo esté llamando. Esto suele producirse cuando se almacena una colección de objetos derivados.

El siguiente fragmento de código crea una colección de animales, que recibe el apropiado nombre de zoo. A continuación añade un perro y dos gatos a este zoo.

```
ArrayList Zoo;  
Zoo = new ArrayList(3);  
  
Cat Sasha, Koshka;  
Sasha = new Cat();  
Koshka = new Cat();  
  
Dog Milou;  
Milou = new Dog();
```

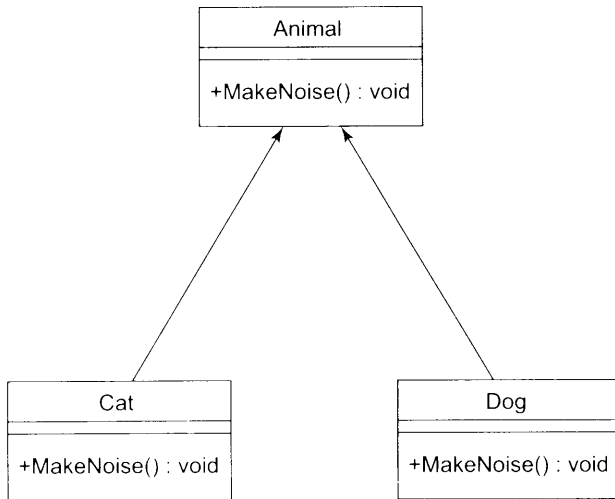


Figura 8.4. Este ejemplo de herencia muestra una clase base y dos clases derivadas.

La colección zoo es una colección polimórfica porque todas sus clases derivan de la clase abstracta *Animal*. Ahora se puede iterar la colección y hacer que cada animal emita el sonido correcto:

```
foreach (Animal a in Zoo)
{
    a.MakeNoise();
}
```

Si ejecuta el código anterior producirá el siguiente resultado:

```
Woof!  
Meow!  
Meow!
```

¿Qué está pasando? Como C# permite el polimorfismo, en el tiempo de ejecución el programa es lo suficientemente inteligente como para llamar a la versión perro de `MakeNoise` cuando se obtiene un perro del zoo y la versión gato de `MakeNoise` cuando se obtiene un gato del zoo.

El listado 8.1 muestra el código C# completo que explica el polimorfismo.

Listado 8.1. Las clases `Cat` y `Dog` evidencia el polimorfismo

```
using System;
using System.Collections;

namespace PolyMorphism
{
    abstract class Animal
    {
        public abstract void MakeNoise();
    }

    class Cat : Animal
    {
        public override void MakeNoise()
        {
            Console.WriteLine("Meow!");
        }
    }

    class Dog : Animal
    {
        public override void MakeNoise()
        {
            Console.WriteLine("Woof!");
        }
    }

    class PolyMorphism
    {
        static int Main(string[] args)
        {
            ArrayList Zoo;
            Zoo = new ArrayList(3);

            Cat Sasha, Koshka;
            Sasha = new Cat();
            Koshka = new Cat();

            Dog Milou;
            Milou = new Dog();

            Zoo.Add(Milou);
            Zoo.Add(Koshka);
            Zoo.Add(Sasha);
        }
    }
}
```

```

foreach (Animal a in Zoo)
{
    a.MakeNoise();
}

// espera a que el usuario acepte los resultados
Console.WriteLine("Hit Enter to terminate...");
Console.Read();
return 0;
}
}
}

```

Resumen

C# es un lenguaje orientado a objetos y los conceptos que se usan en los lenguajes orientados a objetos también se aplican a C#.

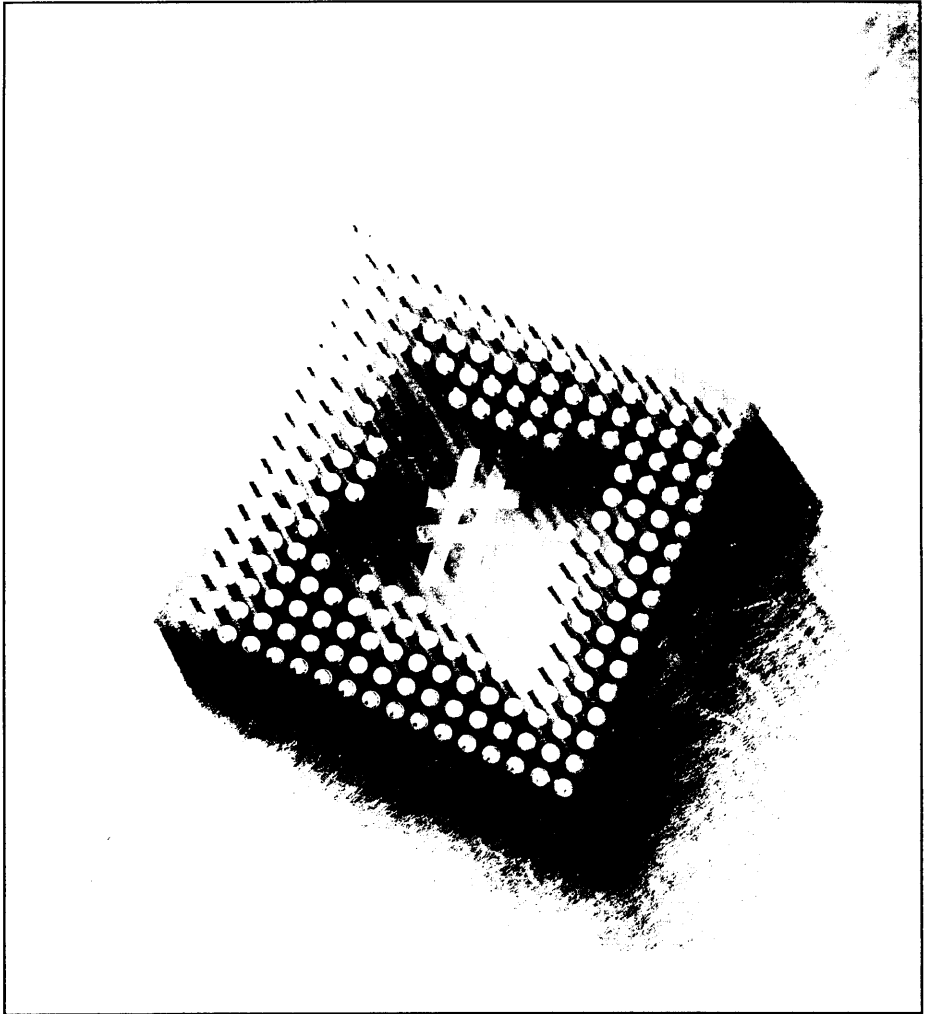
Los *tipos de datos abstractos* son tipos de datos que se definen en el propio código. Los tipos de datos abstractos no forman parte de C#. a diferencia de los tipos de datos primitivos. Se pueden usar tipos de datos abstractos en el código de la misma forma que los tipos de datos primitivos, pero sólo después de haberlos definido en el código.

La *encapsulación* es el acto de diseñar clases que proporcionen un conjunto completo de funcionalidad a los datos de las clases sin exponer directamente esos datos. Si se quiere escribir código que evite que otras partes del código proporcionen valores no válidos a la clase, la encapsulación permite "ocultar" los miembros de datos al hacerlos privados, mientras crea métodos que acceden a ellos y establecen los valores de los miembros de datos como públicos. Otros fragmentos de código pueden llamar a los métodos, que pueden comprobar los valores y emitir un error si los valores no son apropiados.

La *herencia* permite definir una clase por medio de otra. Las clases derivadas heredan de la clase base. Las clases derivadas automáticamente heredan el estado y los comportamientos de la clase base. Se puede usar la herencia para añadir nuevas funciones a clases ya existentes sin necesidad de reescribir desde cero una clase completamente nueva. Algunos lenguajes orientados a objetos permiten tanto herencias simples como herencias múltiples, aunque C# sólo permite la herencia simple.

El *polimorfismo* permite tratar de forma uniforme a una colección de clases derivadas de una sola clase base. Las clases derivadas se recuperan como una clase base y C# automáticamente llama al método correcto en la clase derivada.

Este capítulo trata los conceptos orientados a objetos en general, sin estudiar cómo se usan estos conceptos en el código C#. El resto de los capítulos de esta parte del libro explican cómo se implementan estos conceptos en C#.



9

Clases de C#

El diseño de C# está muy influido por los conceptos del desarrollo de software orientado a objetos. Como la clase es la parte fundamental del software orientado a objetos, no es de sorprender que las clases sean el concepto más importante de C#. Las clases en el mundo del desarrollo de software orientado a objetos contienen código y datos. En C#, los datos se implementan usando miembros de datos y el código se implementa usando métodos. Los *miembros de datos* son cualquier elemento al que se pueda pasar datos dentro y fuera de la clase. Los dos principales tipos de miembros de datos son los campos y las propiedades. En el código C# se pueden escribir tantos miembros de datos y métodos como se desee, pero todos deben incluirse en una o varias clases. El compilador de C# emite un error si se intenta definir alguna variable o se implementa algún método fuera de una definición de clase. Este capítulo trata de los fundamentos de la construcción de clases. Explica cómo crear constructores y destructores, añadir métodos y miembros, además de a usar clases después de crearlos.

Cómo declarar una clase

La acción de declarar una clase resulta similar a declarar una estructura. La principal diferencia reside en que la declaración empieza con la palabra clave de

C# `class` y no con `struct`. Aunque ya hemos visto una definición de clase en anteriores capítulos, vamos a revisar el diseño de una declaración de clase:

- Modificadores de clase opcionales
- La palabra clave `class`
- Un identificador que da nombre a la clase
- Información opcional de la clase base
- El cuerpo de la clase
- Un punto y coma opcional

La declaración de clase mínima en C# es parecida a lo siguiente:

```
class MyClass
{
}
```

Las llaves delimitan el cuerpo de la clase. Los métodos y variables de clase se colocan entre las llaves.

El método Main

Cada aplicación de C# debe contener un método llamado `Main`. Esto es el punto de entrada para que la aplicación se ejecute. Se puede colocar este método dentro de cualquier clase del proyecto porque el compilador es lo bastante inteligente como para buscarlo cuando lo necesite.

El método `Main` debe cumplir dos requisitos especiales para que la aplicación funcione correctamente. En primer lugar, el método debe declararse como `public`. Esto asegura que se pueda acceder al método. En segundo lugar, el método debe declararse como `static`. La palabra clave `static` asegura que sólo se puede abrir una copia del método a la vez.

Teniendo en cuenta estas reglas, observemos este código:

```
class Class1
{
    public static void Main()
    {
    }
}
```

Como puede apreciarse, este ejemplo contiene una clase llamada `Class1`. Esta clase contiene el método `Main` de la aplicación. Es en este método `Main` donde se coloca todo el código necesario para ejecutar su aplicación. Aunque es correcto colocar este método en la misma clase y el mismo archivo que el resto del código de la aplicación, conviene crear una clase y un archivo separados para el

método `Main`. Esta operación sirve de ayuda a los demás programadores que quieran trabajar con este código.

Cómo usar argumentos de línea de comandos

Muchas de las aplicaciones de la plataforma Windows aceptan parámetros de línea de comandos. Para aceptar parámetros de línea de comandos dentro la aplicación C#, es necesario declarar una matriz de cadenas como único parámetro del método `Main`, como se puede apreciar en el siguiente código:

```
class Class1
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
            System.Console.WriteLine(arg);
    }
}
```

Aquí, el método `Main` está contenido en una clase típica. Observe que el método `Main` tiene un parámetro definido, una matriz de cadenas que será almacenada en la variable `args`. Usa el comando `foreach` para recorrer todas las cadenas almacenadas de la matriz `args` y a continuación se escriben esas cadenas en la consola.

NOTA: Si se está usando Visual Studio .NET para programar C#, la matriz de cadenas se añade automáticamente cuando se crea una aplicación de consola.

Si se ejecuta la aplicación anterior sin parámetros, no ocurrirá nada. Si se ejecuta una aplicación como la siguiente:

```
Sampleap.exe parameter1 parameter2
```

el resultado será algo como esto:

```
parameter1
parameter2
```

Los argumentos de línea de comando son un modo excelente de proporcionar modificadores a la aplicación: por ejemplo, para activar un archivo de registro mientras se ejecuta la aplicación.

Cómo devolver valores

Cuando se crea una aplicación, suele ser útil devolver un valor a la aplicación que la inició. Este valor indica al programa que hace la llamada o a la secuencia

de comandos por lotes si el programa se ejecutó con éxito o falló. Para ello, basta con asignar al método `Main` un valor de devolución `int` en lugar de `void`. Cuando el método `Main` está listo para finalizar, sólo hay que usar la palabra clave `return` y un valor que devolver, como se puede apreciar en el siguiente ejemplo:

```
class Class1
{
    public static int Main(string[] args)
    {
        if (args[0] == "fail")
            return 1;
        return 0;
    }
}
```

Esta aplicación acepta un parámetro `fail` o cualquier otra palabra (quizás, `success`). Si la palabra `fail` se pasa a esta aplicación como parámetro, el programa devuelve el valor 1, lo que indica un fallo en el programa. En caso contrario, el programa devuelve 0, lo que indica que el programa finalizó con normalidad. Se puede comprobar el funcionamiento del programa simplemente haciendo que un archivo por lotes ejecute la aplicación y que luego realice algunas acciones dependiendo del código devuelto. El siguiente código es un simple archivo por lotes que realiza esta función:

```
@echo off
retval.exe success

goto answer%errorlevel%

:answer0
echo Program had return code 0 (Success)
goto end

:answer1
echo Program had return code 1 (Failure)
goto end

:end
echo Done!
```

En el código anterior se puede apreciar que se llama al programa con un parámetro `success`. Al ejecutar este programa por lotes, aparece el siguiente mensaje:

```
Program had return code 0 (Success)
```

Si se edita este programa por lotes para que pase la palabra `fail` como parámetro, entonces aparecerá un mensaje que confirma que el programa terminó con un código de salida 1.

El cuerpo de la clase

El cuerpo de la clase puede incluir instrucciones cuya función se incluya en una de estas categorías.

- Constantes
- Campos
- Métodos
- Propiedades
- Eventos
- Indizadores
- Operadores
- Constructores
- Destruidores
- Tipos

Cómo usar constantes

Las *constantes* de clase son variables cuyo valor no cambia. El uso de constantes permite que el código resulte más legible porque pueden incluir identificadores que describan el significado del valor a cualquiera que lea el código. Se puede escribir un código como el siguiente:

```
if(Ratio == 3.14159)
    System.Console.WriteLine("Shape is a circle");
```

El compilador de C# acepta perfectamente este código, pero puede ser un poco difícil de leer, especialmente para alguien que lea código por primera vez y se pregunte qué es el valor de coma flotante. Si se le da un nombre a la constante, por ejemplo pi, se puede escribir el siguiente código:

```
if(Ratio == Pi)
    System.Console.WriteLine("Shape is a circle");
```

El uso de constantes presenta otras ventajas:

- Se le da valor a las constantes cuando se las declara y se usa su nombre, no su valor, en el código. Si por alguna razón es necesario cambiar el valor de la constante, sólo hace falta cambiarlo en un sitio: donde se declara la constante. Si se escribe en las instrucciones de C# el valor real, un cambio en el valor significaría que habría que revisar todo el código y cambiar ese valor.

- Las constantes indican específicamente al compilador de C# que, a diferencia de las variables normales, el valor de la constante no puede cambiar. El compilador de C# emite un error si algún código intenta cambiar el valor de una constante.

La estructura de la declaración de una constante es la siguiente:

- La palabra clave `const`
- Un tipo de dato para la constante
- Un identificador
- Un signo igual
- El valor de la constante

La definición de una constante para pi sería:

```
class MyClass
{
    public const double Pi = 3.1415926535897932384626433832795;

    public static void Main()
    {
    }
}
```

NOTA: Este capítulo usa la palabra clave `public` para indicar que los elementos del cuerpo de clase están disponibles para el resto del código de la aplicación. Posteriormente se examinan algunas alternativas a la palabra clave `public` que pueden usarse cuando se hereda una clase de otra.

C# permite colocar varias definiciones de constantes en la misma línea, siempre que todas las constantes tengan el mismo tipo. Se pueden separar las definiciones de constantes con una coma, como en el siguiente ejemplo:

```
class MyClass
{
    public const int One = 1, Two = 2, Three = 3;

    public static void Main()
    {
    }
}
```

Cómo usar campos

Los *campos* son miembros de datos de una clase. Son variables que pertenecen a una clase y, en términos de programación orientada a objetos, gestionan el

estado de la clase. Se puede acceder a los campos definidos en una clase mediante cualquier método definido en la misma clase.

Un campo se define exactamente igual que cualquier variable. Los campos tienen un tipo y un identificador. También se puede definir un campo sin un valor inicial:

```
class MyClass
{
    public int Field1;

    public static void Main()
    {
    }
}
```

También se puede definir un campo con un valor inicial:

```
class MyClass
{
    public int Field1 = 123;

    public static void Main()
    {
    }
}
```

Para indicar que los campos son de sólo lectura escriba antes de la declaración la palabra clave `readonly`. Si usa la palabra clave `readonly`, deberá escribirla justo antes del tipo del campo:

```
class MyClass
{
    public readonly int Field1;

    public static void Main()
    {
    }
}
```

Se puede establecer el valor de un campo de sólo lectura al declarar el campo o en el constructor de la clase. Su valor no puede establecerse en ningún otro momento. Cualquier intento de cambiar el valor de un campo de sólo lectura es detectado por el compilador de C# y devuelve un error:

```
error CS0191: No se puede asignar un campo de solo lectura
(excepto en un constructor o inicializador de variable)
```

Los campos de sólo lectura se parecen a las constantes en la medida en que se inicializan cuando se crea un objeto de la clase. La principal diferencia entre una constante y un campo de sólo lectura es que el valor de las constantes debe establecerse en tiempo de compilación; en otras palabras, deben recibir su valor

cuando se escribe el código. Los campos de sólo lectura permiten establecer el valor del campo en tiempo de ejecución. Como se puede asignar el valor de un campo de sólo lectura en un constructor de clase, se puede determinar su valor en tiempo de ejecución y establecerlo cuando el código se está ejecutando.

Supongamos, por ejemplo, que estamos escribiendo una clase de C# que gestiona la lectura de un grupo de archivos de una base de datos. Quizás queramos que la clase publique un valor que especifique el número de archivos que hay en la base de datos.

Una constante no es la elección correcta para este valor porque el valor de una constante debe establecerse cuando se escribe el código y no podemos saber cuántos archivos contiene la clase hasta que se ejecute el código. Podemos poner estos datos en un campo cuyo valor puede establecerse después de que la clase haya empezado a ejecutarse y se puede determinar el número de archivos. Como el número de archivos no cambia durante la existencia de la clase, quizás queramos señalar el campo como de sólo lectura para que los otros fragmentos de código no puedan cambiar su valor.

Cómo usar métodos

Los *métodos* son bloques de código con nombre en una clase. Los métodos proporcionan los comportamientos de las clases. Pueden ejecutarse por cualquier otro fragmento de código en la clase y, como se explicará en un capítulo posterior, también pueden ejecutarlos otras clases.

Cómo usar propiedades

En un capítulo anterior se examina la estructura `Point` que puede describir un punto en una pantalla con una resolución de 640 x 480 píxeles. Un modo de implementar esta estructura es definirla con miembros de datos públicos que describan las coordenadas del punto:

```
struct Point
{
    public int X;
    public int Y;
}
```

Al usar una estructura como ésta, los clientes pueden usar la sintaxis `structurename.field` para trabajar con uno de los campos de la estructura:

```
Point MyPoint = new Point();

MyPoint.X = 100;
```

Los clientes pueden escribir fácilmente este código. Los clientes pueden acceder al campo directamente y usar su valor en una expresión.

El problema de este enfoque es que los clientes pueden establecer un campo que C# permite, pero que no es lógico para el código. Por ejemplo, si está gestionando una estructura `Point` que describe un punto en una pantalla de 640 x 480 píxeles, el valor lógico más alto para X debería ser 639 (suponiendo que los valores legales de X están entre 0 y 639). Sin embargo, como se especifica que la coordenada X es un tipo `int`, los clientes pueden establecer como valor cualquier valor autorizado dentro de los límites de los tipos enteros:

```
MyPoint.X = -500;  
  
MyPoint.X = 9000;
```

El compilador C# acepta este código porque los valores que se asignan a X están dentro de los límites válidos para los valores enteros.

Una solución a este problema es hacer que los valores sean privados, con lo que serían inaccesibles para otros fragmentos de código y luego añadir un método público a la estructura que establece los valores de X:

```
struct Point  
{  
    private int X;  
    private int Y;  
  
    public bool SetX(int NewXValue)  
    {  
        if((NewXValue < 0) || (NewXValue > 639))  
            return false;  
        X = NewXValue;  
        return true;  
    }  
}
```

La ventaja de este enfoque es que obliga a los clientes que quieran asignar un valor a X a llamar al método para realizar la tarea:

```
Point MyPoint = new Point();  
  
MyPoint.SetX(100);
```

La ventaja del método es que se puede escribir código que valide el nuevo valor antes de que se almacene realmente en el campo y el código del método puede rechazar el nuevo valor si, por lógica, no es adecuado. Así pues, los clientes llaman al método para establecer un nuevo valor.

Aunque este enfoque funciona, llamar a un método para asignar un valor requiere un poco más de código que asignar un valor directamente. Para el código es más natural asignar un valor a un campo que llamar a un método para que lo asigne.

En el mejor de los casos, querremos lo mejor de los dos elementos: que los clientes puedan leer y escribir directamente los valores de los campos usando

instrucciones de asignación simples, pero también queremos que el código intervenga por anticipado y haga todo lo necesario para obtener el valor de un campo o validar el nuevo valor antes de que se asigne. Afortunadamente, C# ofrece esta posibilidad con un concepto de clase llamado *propiedad*.

Las propiedades son miembros identificados que proporcionan acceso al estado de un objeto. Las propiedades tienen un tipo y un identificador y tienen uno o dos fragmentos de código asociados a ellos: una base de código *get* y una base de código *set*. Estas bases de código reciben el nombre de *descriptores de acceso*. Cuando un cliente accede a una propiedad se ejecuta el descriptor de acceso *get* de la propiedad. Cuando el cliente establece un nuevo valor para la propiedad se ejecuta el descriptor de acceso *set* de la propiedad.

Para mostrar cómo funcionan las propiedades, el listado 9.1 usa una clase *Point* que expone los valores de X e Y como propiedades.

Listado 9.1. Valores Point como propiedades de clase

```
class Point
{
    private int XCoordinate;
    private int YCoordinate;

    public int X
    {
        get
        {
            return XCoordinate;
        }
        set
        {
            if((value >= 0) && (value < 640))
                XCoordinate = value;
        }
    }

    public int Y
    {
        get
        {
            return YCoordinate;
        }
        set
        {
            if((value >= 0) && (value < 480))
                YCoordinate = value;
        }
    }

    public static void Main()
    {
        Point MyPoint = new Point();
    }
}
```

```

    MyPoint.X = 100;
    MyPoint.Y = 200;
    System.Console.WriteLine(MyPoint.X);
    System.Console.WriteLine(MyPoint.Y);
    MyPoint.X = 600;
    MyPoint.Y = 600;
    System.Console.WriteLine(MyPoint.X);
    System.Console.WriteLine(MyPoint.Y);
}
}

```

Este código declara una clase `Point` con los valores `X` e `Y` como propiedades. El método `Main()` crea un nuevo objeto de la clase `Point` y accede a las propiedades de la clase.

La clase `Point` define dos campos privados que contienen los valores de las coordenadas del punto. Como estos campos son privados, el código que se encuentra fuera de la clase `Point` no puede acceder a sus valores. La clase también define dos propiedades públicas que permiten que otros fragmentos de código trabajen con los valores de las coordenadas del punto. Las propiedades son públicas y pueden ser usadas por otras partes del código. La propiedad `X` es la propiedad pública que gestiona el valor del campo privado `XCoordinate` y la propiedad `Y` es la propiedad pública que gestiona el valor del campo privado `YCoordinate`. Las dos propiedades tienen un descriptor de acceso `get` y otro `set`. La figura 9.1 muestra el resultado del listado 9.1.

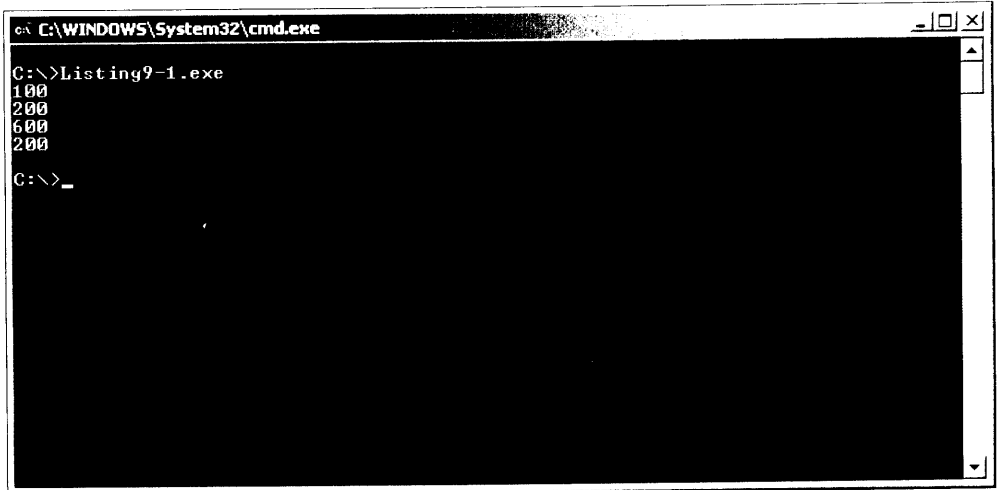


Figura 9.1. Las propiedades de clase ayudan a almacenar los valores del punto.

Descriptores de acceso `get`

Los descriptores de acceso `get` solamente devuelven el valor actual del campo correspondiente: el descriptor de acceso de la propiedad `X` devuelve el valor ac-

tual de `Xcoordinate` y el descriptor de acceso de la propiedad `Y` devuelve el valor actual de `Ycoordinate`.

Cada descriptor de acceso `get` debe devolver un valor que coincida o pueda ser convertido implícitamente al tipo de la propiedad. Si el descriptor de acceso no devuelve un valor, el compilador de `C#` emite el siguiente mensaje de error:

```
error CS0161: 'MyClass.Property.get': no todas las rutas de código devuelven un valor
```

Descriptores de acceso `set`

Los descriptores de acceso `set` del ejemplo son un poco más complicados porque tienen que validar el nuevo valor antes de que sea realmente asignado al campo asociado.

Observe que el descriptor de acceso usa la palabra clave `value`. El valor de este identificador es el valor de la expresión que aparece después del signo de igualdad cuando se llama al descriptor de acceso `set`. Por ejemplo, examine la siguiente instrucción del listado 9.1:

```
MyPoint.X = 100;
```

El compilador de `C#` determina que la instrucción está asignando un nuevo valor a la propiedad `X` del objeto `Point`. Ejecuta el descriptor de acceso `set` de la clase para realizar la asignación. Como se está asignando a la propiedad el valor 100, el valor de la palabra clave `value` del descriptor de acceso `set` será 100.

Los descriptores de acceso `set` del listado 9.1 asignan el valor al campo correspondiente, pero sólo si el valor está dentro de los límites válidos. En `C#`, no está permitido devolver valores de los descriptores de acceso.

NOTA: Como `C#` no permite que los descriptores de acceso `set` devuelvan un valor, no se puede devolver un valor que especifique si la asignación fue o no satisfactoria. Sin embargo, se pueden usar excepciones para informar de cualquier error. Estas excepciones se estudian en un capítulo posterior.

Propiedades de sólo lectura y de sólo escritura

Las propiedades del listado 9.1 pueden leerse usando su descriptor de acceso `get` y puede escribirse en ellas usando su descriptor de acceso `set`. Estas propiedades reciben el nombre de *propiedades de lectura y escritura*. Cuando se diseñan las clases de `C#`, hay que implementar una propiedad de sólo lectura o de sólo escritura. En `C#`, esto es sencillo.

Si hace falta implementar una propiedad de sólo lectura, se especifica una propiedad con un descriptor de acceso `get` pero sin descriptor de acceso `set`:

```

int X
{
    get
    {
        return XCoordinate;
    }
}

```

Si hace falta implementar una propiedad de sólo escritura, se especifica una propiedad con un descriptor de acceso `set` pero sin descriptor de acceso `get`:

```

int X
{
    set
    {
        if((value >= 0) && (value < 640))
            XCoordinate = value;
    }
}

```

Cómo usar eventos

C# permite que las clases informen a otras partes del código cuando se produce una acción sobre ella en la clase. Esta capacidad recibe el nombre de *mecanismo de evento* y permite informar a los elementos que realizaron la llamada si se produce un evento en la clase de C#. Puede diseñar clases de C# que informen a otros fragmentos de código cuando se produzcan determinados eventos en la clase. La clase puede devolver una notificación de evento al fragmento de código original. Quizás quiera usar un evento para informar a otros fragmentos de código de que se ha completado una operación muy larga. Suponga, por ejemplo, que quiere diseñar una clase de C# que lea una base de datos. Si la actividad sobre la base de datos va a requerir mucho tiempo, será mejor que otras partes del código realicen otras acciones mientras se lee la base de datos. Cuando se completa la lectura, la clase de C# puede emitir un evento que indique "la lectura se ha completado". También se puede informar a otras partes del código cuando se emita este evento y el código puede realizar la acción indicada cuando reciba el evento de la clase de C#.

Cómo usar indizadores

Algunas de las clases pueden actuar como contenedores de otros valores.

Supongamos, por ejemplo, que estamos escribiendo una clase de C# llamada `Rainbow` que permite a los clientes acceder a los valores de cadenas que nombran los colores del arco iris en orden. Queremos que los elementos que realizan llamadas puedan retirar los valores de las cadenas, de modo que usamos algunos métodos públicos que permiten a los elementos que realizan llamadas tener acceso a los valores. El listado 9.2 muestra un ejemplo de este tipo de código.

Listado 9.2. Una clase con un conjunto de valores de cadena

```
class Rainbow
{
    public int GetNumberOfColors()
    {
        return 7;
    }

    public bool GetColor(int ColorIndex, out string ColorName)
    {
        bool ReturnValue;

        ReturnValue = true;
        switch(ColorIndex)
        {
            case 0:
                ColorName = "Red";
                break;
            case 1:
                ColorName = "Orange";
                break;
            case 2:
                ColorName = "Yellow";
                break;
            case 3:
                ColorName = "Green";
                break;
            case 4:
                ColorName = "Blue";
                break;
            case 5:
                ColorName = "Indigo";
                break;
            case 6:
                ColorName = "Violet";
                break;
            default:
                ColorName = "";
                ReturnValue = false;
                break;
        }
        return ReturnValue;
    }

    public static void Main()
    {
        int ColorCount;
        int ColorIndex;
        Rainbow MyRainbow = new Rainbow();

        ColorCount = MyRainbow.GetNumberOfColors();
    }
}
```

```

string ColorName;
bool Success;

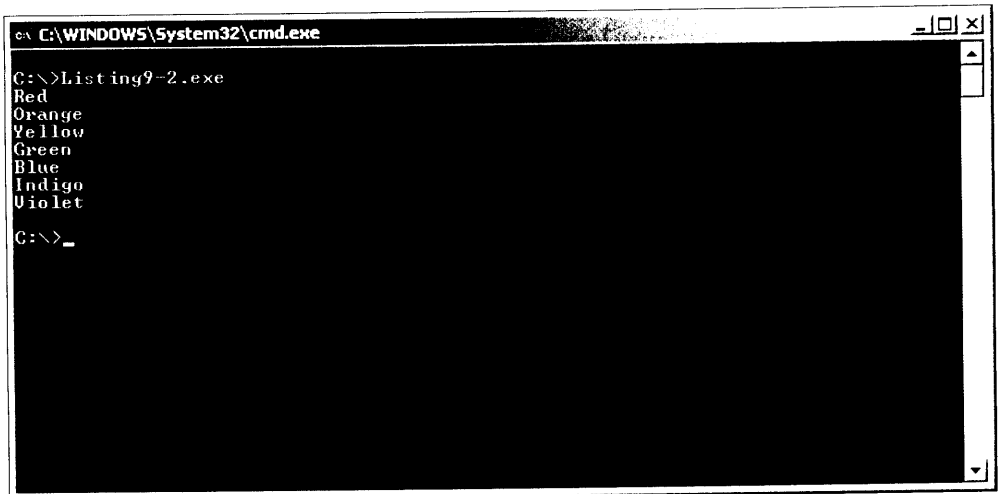
for(ColorIndex = 0; ColorIndex < ColorCount; ColorIndex++)
{
    Success = MyRainbow.GetColor(ColorIndex, out
ColorName);
    if(Success == true)
        System.Console.WriteLine(ColorName);
}
}
}

```

La clase `Rainbow` del listado 9.2 tiene dos métodos públicos:

- `GetColorCount()`. que devuelve el número de colores de la clase
- `GetColor()`. que, basándose en un número de color, devuelve el nombre de uno de los colores de la clase

El método `Main()` crea un nuevo objeto de la clase `Rainbow` y pide al objeto el número de colores que contiene. A continuación se coloca en un bucle `for` y solicita el nombre de cada color. El nombre del color aparece en la consola. El resultado de esta aplicación puede verse en la figura 9.2.



```

C:\WINDOWS\System32\cmd.exe
C:\>Listing9-2.exe
Red
Orange
Yellow
Green
Blue
Indigo
Violet
C:\>_

```

Figura 9.2. Los indizadores recuperan los nombres de los colores.

La clase mantiene una colección de valores del mismo tipo de valor, lo que es muy parecido a una matriz. De hecho, esta clase `Rainbow` también puede implementarse como una matriz y el elemento que realiza la llamada puede usar los corchetes de la sintaxis del descriptor de acceso a la matriz de elementos para recuperar un nombre de color en particular:

```
ColorName = Rainbow[ColorIndex];
```


Los indizadores permiten que se acceda a las clases como si fuera una matriz. Para especificar qué se debe devolver cuando el elemento que hace la llamada usa corchetes para acceder al elemento de la clase, se usa un fragmento de código llamado *descriptor de acceso de indizador*.

Teniendo esto en cuenta, se puede reescribir la clase `Rainbow` para que permita a los elementos que la llaman acceder a los nombres de los colores usando corchetes en su sintaxis. Se elimina el método `GetColor()` y se sustituye por un indizador, y se sustituye el método `GetColorCount()` por una propiedad de sólo lectura llamada `Count`, como se muestra en el listado 9.3.

Listado 9.3. Clase `Rainbow` con un indizador

```
class Rainbow
{
    public int Count
    {
        get
        {
            return 7;
        }
    }

    public string this[int ColorIndex]
    {
        get
        {
            switch(ColorIndex)
            {
                case 0:
                    return "Red";
                case 1:
                    return "Orange";
                case 2:
                    return "Yellow";
                case 3:
                    return "Green";
                case 4:
                    return "Blue";
                case 5:
                    return "Indigo";
                case 6:
                    return "Violet";
                default:
                    return "";
            }
        }
    }

    public static void Main()
    {
        int ColorIndex;
```

```

    Rainbow MyRainbow = new Rainbow();

    string ColorName;

    for(ColorIndex = 0; ColorIndex < MyRainbow.Count;
    ColorIndex++)
    {
        ColorName = MyRainbow[ColorIndex];
        System.Console.WriteLine(ColorName);
    }
}

```

Los indizadores se parecen a las propiedades porque tienen descriptores de acceso `get` y `set`. Si fuera necesario, se puede omitir cualquiera de estos descriptores de acceso. El indizador del listado 9.3 no tiene el descriptor de acceso `set`, lo que significa que la clase se puede leer, pero no se puede escribir en ella.

Los indizadores se estructuran según los siguientes elementos:

- Un tipo que indica el tipo de datos que devuelve el descriptor de acceso.
- La palabra clave `this`.
- Un corchete de apertura.
- Una lista de parámetros, estructurada igual que una lista de parámetros de un método.
- Un corchete de cierre.
- Un cuerpo de código, entre llaves.

El indizador del listado 9.4 recibe un argumento integral y devuelve una cadena, de forma parecida al método `GetColor()` del listado 9.2.

Listado 9.4. Un argumento entero que devuelve una cadena

```

string ColorName;

for(ColorIndex = 0; ColorIndex < MyRainbow.Count; ColorIndex++)
{

    ColorName = MyRainbow[ColorIndex];
    System.Console.WriteLine(ColorName);
}

```

El nuevo código usa los corchetes de la sintaxis del elemento de matriz para obtener el nombre de un color del objeto `MyRainbow`. Esto hace que el compilador de C# llame al código indizador de la clase, que pasa el valor de `ColorIndex` como parámetro. El indizador devuelve una cadena y la cadena se escribe en la consola.

Una clase puede implementar más de un indizador. siempre que los indicadores tengan diferentes listas de parámetros. Las listas de parámetros de los indicadores pueden tener más de un parámetro y pueden ser de cualquier tipo que se pueda usar en el código. No es necesario que los indicadores usen valores enteros como indicadores. Se podría haber implementado igualmente en la clase `Rainbow` un indizador que aceptase un valor `double`:

```
public string this[double ColorIndex]
```

NOTA: Como en las propiedades, C# no permite que los descriptores de acceso `set` devuelvan valores. Sin embargo, pueden usarse excepciones para informar de cualquier error.

Cómo usar operadores

Un operador permite definir el comportamiento de la clase cuando se usa en una expresión con un operador unario o binario. Esto significa que se puede ampliar el comportamiento de operadores predefinidos para que se ajusten a las necesidades de las clases. Por ejemplo, la clase `Point` puede implementar código que especifique que se pueden sumar dos objetos `Point` con el operador `+`. El resultado de la suma será un tercer objeto `Point` cuyo estado es el resultado de sumar los otros dos puntos.

Cómo usar constructores

Los *constructores de estructura* son métodos especiales que se ejecutan cuando se crea una variable del tipo de la estructura. Los constructores suelen usarse para inicializar una estructura con un estado conocido.

Se pueden usar constructores en las clases de C# de la misma forma que las estructuras. Se pueden definir tantos constructores como se desee, siempre que cada constructor tenga una lista de parámetros diferente. Se puede escribir una clase `Point` con constructores de la misma forma que se hizo con la estructura `Point`, como se puede apreciar en el listado 9.5.

Listado 9.5. Una clase `Point` con dos constructores

```
class Point
{
    public int X;
    public int Y;

    public Point()
    {
        X = 0;
    }
}
```

```

        Y = 0;
    }

    public Point(int InitialX, int InitialY)
    {
        X = InitialX;
        Y = InitialY;
    }

    public static void Main()
    {
        Point MyFirstPoint = new Point(100, 200);
        Point MySecondPoint = new Point();
    }
}

```

La clase `Point` del listado 9.5 tiene dos campos públicos: `X` e `Y`. También implementa dos constructores. Uno de ellos no usa ningún parámetro y el otro usa dos parámetros. Los constructores de las clases de `C#` son muy parecidas a las estructuras de `C#`. Los constructores de clase no devuelven valores y su nombre debe coincidir con el nombre de la clase. La principal diferencia entre los constructores de estructura y los constructores de clase estriba en que los constructores de clase pueden implementar un constructor sin parámetros, pero un constructor de estructura no puede hacerlo.

Si se define una clase sin constructores, `C#` proporciona un constructor por defecto. Este constructor por defecto asigna a todos los campos de las clases sus valores por defecto. Si se inicializa con el signo igual cualquier campo de una clase, éste se inicializará antes de que se ejecute el constructor:

```

class Point
{
    public int X = 100;
    public int Y;

    public Point(int InitialY)
    {
        Y = InitialY + X;
    }
}

```

En esta clase `Point`, una declaración de asignación inicializa el campo `X` y el constructor y inicializa el campo `Y`. Al compilar este código, el compilador de `C#` se asegura de que el campo `X` se inicialice en primer lugar.

Cómo usar destructores

Las clases de `C#` pueden definir un destructor, que es un método especial que se ejecuta cuando el CLR (Entorno común de ejecución) destruye los objetos de

la clase. Puede pensar en los destructores como en lo contrario de los constructores: los constructores se ejecutan cuando se crean objetos y los destructores se ejecutan cuando el recolector de objetos no utilizados destruye los objetos. Este proceso tiene lugar de modo oculto sin consecuencias para el programador.

Los destructores son opcionales. Es perfectamente válido escribir una clase de C# sin un destructor (y hasta ahora, es lo que hemos estado haciendo en los ejemplos). Si se escribe un destructor, sólo se puede escribir uno.

NOTA: A diferencia de los constructores, no se puede definir más de un destructor para una clase.

Los destructores se disponen de la siguiente forma:

- El símbolo virgulilla (~).
- El identificador del destructor, que debe corresponder al nombre de la clase.
- Un conjunto de paréntesis.

El listado 9.6 actualiza la clase `Point` del listado 9.5 con un destructor:

Listado 9.6. Una clase `Point` con un destructor

```
class Point
{
    public int X;
    public int Y;

    public Point()
    {
        X = 0;
        Y = 0;
    }

    public Point(int InitialX, int InitialY)
    {
        X = InitialX;
        Y = InitialY;
    }

    ~Point()
    {
        X = 0;
        Y = 0;
    }

    public static void Main()
    {
```

```

        Point MyFirstPoint = new Point(100, 200);
        Point MySecondPoint = new Point();
    }
}

```

Los destructores no devuelven valores: ni tampoco pueden aceptar parámetros. Si se intenta llamar a un destructor en el código, el compilador de C# emite un error.

En muchos lenguajes orientados a objetos, se llama a los destructores de clase cuando la variable ya no puede ser usada. Supongamos, por ejemplo, que escribimos un método y declaramos un objeto `Point` como una variable local del método. Cuando se llama al método, se crea el objeto `Point` y el método puede trabajar con él. Cuando el método llega al final de su bloque de código, el objeto `Point` ya no volverá a usarse. En lenguajes como C++, esto hace que se llame al constructor de la clase cuando acaba el método.

En C#, esto no tiene por qué ocurrir. De hecho, no se puede llamar a un destructor de clase. Recuerde que el CLR implementa una utilidad llamada recolector de objetos no utilizados que destruye objetos que ya no se usan en el código. La recolección puede tener lugar mucho después de que el objeto deje de ser accesible. Observe el método `Main()` en el listado 9.7:

Listado 9.7. Demostración del uso del recolector de objetos no utilizados mediante la estructura `Point`

```

public static void Main()
{
    Point MyFirstPoint = new Point(100, 200);
    Point MySecondPoint = new Point();
}

```

El método `Main()` crea dos objetos locales `Point`. Cuando el método `Main()` termina su ejecución, los objetos locales `Point` ya no pueden volver a ser usados y el CLR los registra como objetos que pueden ser destruidos cuando se ejecute el recolector de objetos no utilizados. Sin embargo, el recolector de objetos no utilizados no siempre se ejecuta inmediatamente, lo que significa que no siempre se llama al destructor del objeto inmediatamente.

NOTA: Se llama a los destructores de las clases de C# cuando se destruye un objeto, no cuando su variable deja de ser accesible.

Por ejemplo, suponga que quiere escribir una clase de C# que gestione un archivo en un disco. Escribiremos una clase llamada `File` con un constructor que abra el archivo y un destructor que cierre el archivo:

```

class File
{

```

```

File(string Name)
{
    // abre el archivo
}
~File()
{
    // cierra el archivo
}
}

```

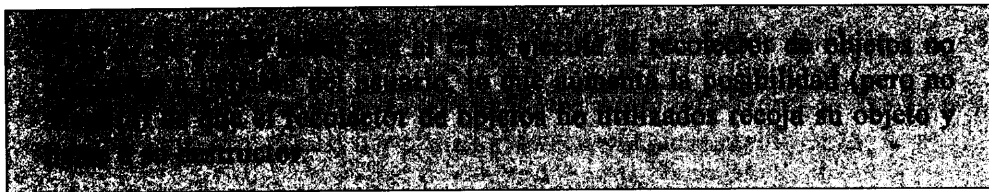
Si queremos que esta clase trabaje en un método:

```

public static void Main()
{
    File MyFile = new File("myfile.txt");
}

```

El destructor de la clase `File` cierra el archivo, pero en realidad el destructor no se ejecuta hasta que el recolector de objetos no utilizados no se activa. Esto significa que el archivo puede todavía estar abierto mucho después de que la variable `MyFile` sea inaccesible. Para asegurarnos de que el archivo se cierra lo antes posible, añadimos a la clase `File` un método `Close()`, que cierra el archivo cuando se le llama.



Cómo usar los tipos de clase

Las clases pueden usar uno de los tipos integrados en C# (`int`, `long` o `char`, por ejemplo) y pueden definir sus propios tipos. Las clases pueden incluir declaraciones de otros elementos, como estructuras o incluso otras clases.

Cómo usar la palabra clave `this` como identificador

En C# se puede emplear la palabra clave `this` para identificar un objeto cuyo código se está ejecutando, lo que a su vez permite hacer referencia a ese objeto.

La palabra clave `this` se puede emplear de varias maneras. Ya hemos visto cómo se usa en un indizador. También se puede usar como prefijo de un identificador de variable para advertir al compilador de que una determinada

expresión debe hacer referencia a un campo de clase. Observe, por ejemplo, la clase `Point` en el listado 9.8.

Listado 9.8. Campos y parámetros con el mismo nombre

```
class Point
{
    public int X;
    public int Y;

    Point(int X, int Y)
    {
        X = X;
        Y = Y;
    }

    public static void Main()
    {
    }
}
```

Este código no tiene el comportamiento esperado porque los identificadores `X` e `Y` se emplean en dos ocasiones: como identificadores de campo y en la lista de parámetros del constructor. El código debe diferenciar el identificador de campo `X` del identificador de la lista de parámetros `X`. Si el código es ambiguo, el compilador de `C#` interpreta que las referencias a `X` e `Y` en las declaraciones del constructor se refieren a los parámetros y el código establece los parámetros con el valor que ya contienen.

Se puede usar la palabra clave `this` para diferenciar el identificador de campo del identificador de parámetro.

El listado 9.9 muestra el código corregido, usando la palabra clave `this` como prefijo del campo de nombre.

Listado 9.9. Cómo usar `this` con campos

```
class Point
{
    public int X;
    public int Y;

    Point(int X, int Y)
    {
        this.X = X;
        this.Y = Y;
    }

    public static void Main()
    {
    }
}
```


El modificador `static`

Cuando se define un campo o un método en una clase, cada objeto de esa clase creado por el código tiene su propia copia de valores de campo y métodos. Mediante la palabra clave `static`, se puede invalidar este comportamiento, lo que permite que varios objetos de la misma clase compartan valores de campo y métodos.

Cómo usar campos estáticos

Retomemos el ejemplo de la clase `Point`. Una clase `Point` puede tener dos campos para las coordenadas `x` e `y` del punto. Todos los objetos creados a partir de la clase `Point` tienen copias de esos campos, pero cada objeto puede tener sus propios valores para las coordenadas `x` e `y`. Asignar un valor a las coordenadas `x` e `y` de un objeto no afecta a los valores de otro objeto:

```
Point MyFirstPoint = new Point(100, 200);
Point MySecondPoint = new Point(150, 250);
```

En este ejemplo se crean dos objetos de la clase `Point`. El primer objeto asigna a su copia de las coordenadas `x` e `y` los valores 100 y 200, respectivamente y el segundo objeto asigna a su copia de las coordenadas `x` e `y` los valores 150 y 250, respectivamente. Cada objeto guarda su propia copia de las coordenadas `x` e `y`.

Si se coloca el modificador `static` antes de una definición de campo se está indicando que todos los objetos de la misma clase compartirán el mismo valor. Si un objeto asigna un valor estático, todos los demás objetos de esa misma clase compartirán ese mismo valor. Observe el listado 9.10.

Listado 9.10. Campos estáticos

```
class Point
{
    public static int XCoordinate;
    public static int YCoordinate;

    public int X
    {
        get
        {
            return XCoordinate;
        }
    }

    public int Y
    {
        get
```

```

    {
        return YCoordinate;
    }
}

public static void Main()
{
    Point MyPoint = new Point();
    System.Console.WriteLine("Before");
    System.Console.WriteLine("=====");
    System.Console.WriteLine(MyPoint.X);
    System.Console.WriteLine(MyPoint.Y);

    Point.XCoordinate = 100;
    Point.YCoordinate = 200;

    System.Console.WriteLine("After");
    System.Console.WriteLine("=====");
    System.Console.WriteLine(MyPoint.X);
    System.Console.WriteLine(MyPoint.Y);
}
}

```

La clase `Point` del listado 9.10 tiene dos campos estáticos enteros llamados `XCoordinate` e `YCoordinate`. También tiene dos propiedades de sólo lectura, llamadas `X` e `Y`, que devuelven los valores de las variables estáticas. El método `Main()` crea un nuevo objeto `Point` y escribe sus coordenadas en la consola. A continuación cambia los valores de los campos estáticos y vuelve a escribir las coordenadas del objeto `Point`. El resultado puede verse en la figura 9.3.

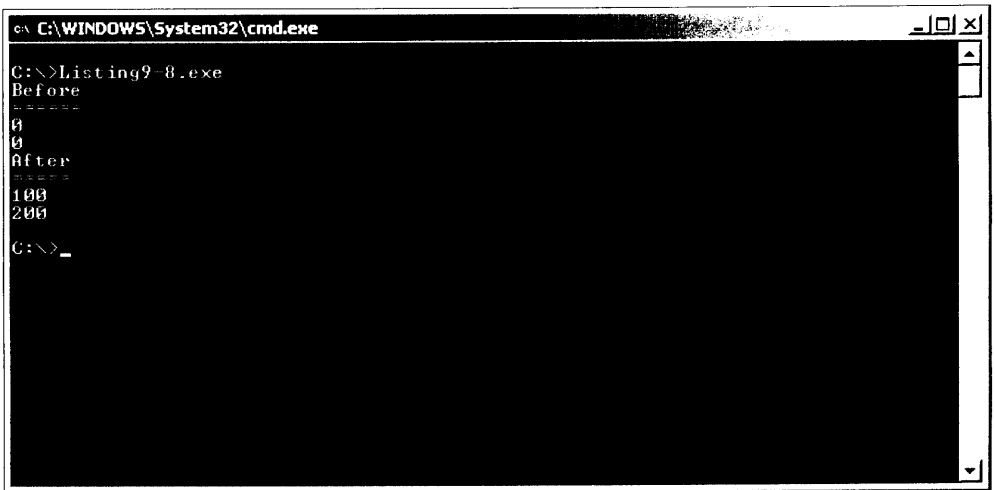


Figura 9.3. El uso de campos estáticos simplifica la codificación.

Hay que tener en cuenta que los valores de las coordenadas del objeto `Point` han cambiado, aunque los valores del propio objeto no hayan cambiado. Esto es

debido a que el objeto `Point` comparte campos estáticos con todos los demás objetos `Point` y cuando los campos estáticos de la clase `Point` cambian, todos los objetos de esa clase se ven afectados.

Los campos estáticos que se usan en expresiones no están prefijados con un identificador de objeto sino con el nombre de la clase que contiene los campos estáticos. La siguiente instrucción es un error porque `MyPoint` hace referencia a un objeto y `XCoordinate` hace referencia a un campo estático:

```
MyPoint.XCoordinate = 100;
```

Este código hace que se produzca el siguiente error del compilador de C#:

```
error CS0176: No se puede obtener acceso al miembro estatico  
'Point.XCoordinate' con una referencia de instancia; utilice un  
nombre de tipo en su lugar
```

El campo estático debe estar prefijado con el nombre de la clase:

```
Point.XCoordinate = 100;
```

Cómo usar constantes estáticas

Las constantes operan de la misma forma que los campos a menos que estén precedidas por la palabra clave `static`; en ese caso, cada objeto de la clase contiene su propia copia de la constante. Sin embargo, hacer que cada objeto de una clase contenga su propia copia de una constante supone desperdiciar memoria. Supongamos que estamos escribiendo una clase llamada `Circle`, que gestiona un círculo. Como estamos trabajando con un círculo, usaremos bastante el valor `pi`. Decidimos que `pi` sea una constante para referirnos siempre a ella con un nombre, en lugar de con un enorme número de coma flotante.

Ahora bien ¿qué ocurre si creamos mil objetos círculo? Por defecto, cada uno de ellos tiene su propia copia de `pi` en memoria. Esto es un desperdicio de memoria, especialmente porque `pi` es una constante y su valor nunca cambia. Es más lógico que cada objeto de la clase `Circle` use una sola copia de la constante `pi`.

Para eso sirve la palabra clave `static`. Si se usa la palabra clave `static` con una constante cada objeto de una clase trabaja con una sola copia del valor de la constante en memoria:

```
const double Pi = 3.1415926535897932384626433832795;
```

En general hay que intentar que todas las constantes sean estáticas de modo que sólo haya una copia del valor de la constante en memoria a la vez.

Cómo usar métodos estáticos

Los métodos que se definen mediante la palabra clave `static` reciben el nombre de *métodos estáticos*. Los métodos que no se definen mediante la palabra

clave `static` reciben el nombre de *métodos de instancia*. Los métodos estáticos están incluidos en una clase, pero no pertenecen a ningún objeto específico. Igual que los campos estáticos y las constantes estáticas, todos los objetos de una clase comparten una copia de un método estático. Los métodos estáticos no pueden hacer referencia a ninguna parte de un objeto que no esté también marcado como estático, como puede apreciarse en el listado 9.11.

Listado 9.11. Métodos estáticos que llaman a un método de instancia de clase

```
class Listing9_9
{
    public static void Main()
    {
        CallMethod();
    }

    void CallMethod()
    {
        System.Console.WriteLine("Hello from CallMethod()");
    }
}
```

El código anterior no se compila y el compilador de C# emite el siguiente error:

```
error CS0120: Se requiere una referencia a objeto para el
campo, método o propiedad no estáticos
'Listing9_9.CallMethod()'
```

El error del código del listado 9.11 es que hay un método estático, `Main()`, que intenta llamar a un método de instancia, `CallMethod()`. Esto no está permitido porque los métodos de instancia son parte de una instancia de objeto y los métodos estáticos no.

Para corregir el código, el método estático `Main()` debe crear otro objeto de la clase y llamar al método de instancia desde el nuevo objeto, como muestra el listado 9.12.

Listado 9.12. Métodos estáticos que llaman a un método de instancia de clase

```
class Listing9_10
{
    public static void Main()
    {
        Listing9_10 MyObject = new Listing9_10();

        MyObject.CallMethod();
    }

    void CallMethod()
    {
```

```

        System.Console.WriteLine("Hello from CallMethod()");
    }
}

```

La figura 9.4 muestra el resultado del listado 9.12.

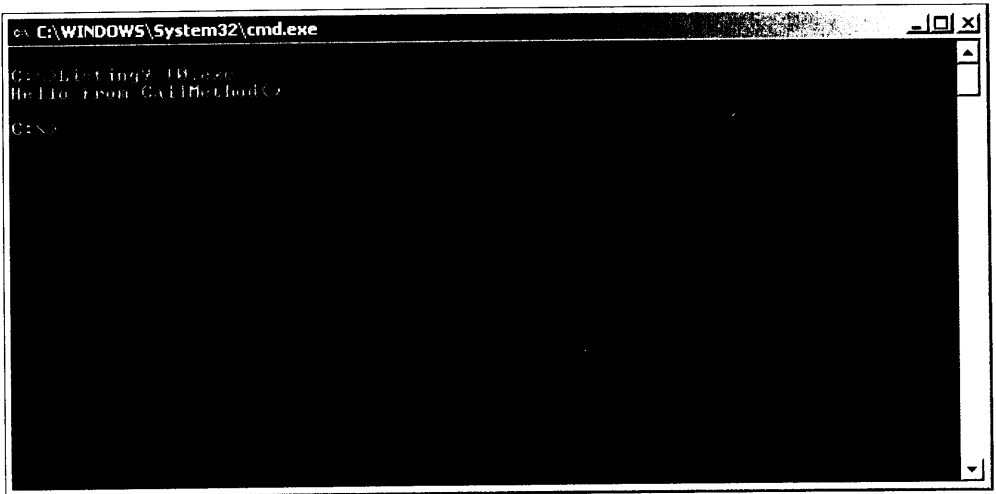


Figura 9.4. Ejemplo de una llamada a un método estático desde dentro de la misma clase

Como todos los elementos de las clases estáticas, los métodos estáticos aparecen sólo una vez en memoria, por lo que se debe marcar el método `Main()` como `static`. Cuando el código .NET se carga en memoria, el CLR empieza a ejecutar el método `Main()`. Recuerde que sólo puede haber un método `Main()` en memoria a la vez. Si una clase tiene varios métodos `Main()`, el CLR no sabría qué método `Main()` ejecutar cuando el código lo necesite. Usar la palabra clave `static` en el método `Main()` hace que sólo haya disponible en memoria una copia del método `Main()`.

NOTA: Mediante el uso de parámetros de líneas de comandos en el compilador de C#, es posible incluir más de un método `Main()` dentro de una aplicación. Esto puede ser muy útil cuando se quiere probar más de un método para depurar el código.

Resumen

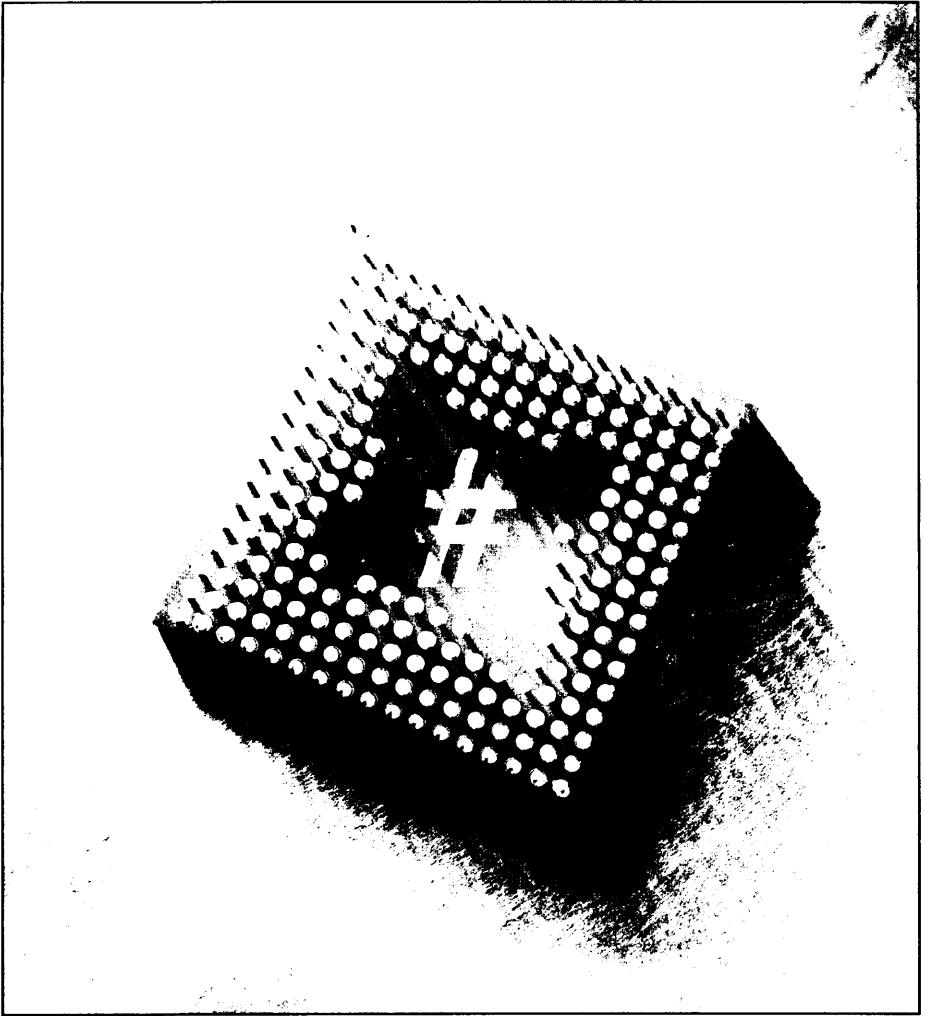
C# es un lenguaje orientado a objetos y los conceptos que se emplean en los lenguajes orientados a objetos se pueden aplicar a C#. Las clases de C# pueden usar varios tipos de miembros de clase:

- Las *constantes* dan nombre a un valor que no cambia en todo el código. El uso de constantes hace que el código sea más legible porque se pueden usar los nombres de las constantes en lugar de los valores literales.
- Los *campos* contienen el estado de las clases. Son variables que están asociadas a un objeto.
- Los *métodos* contienen el comportamiento de la clase. Son fragmentos de código con nombre que realizan una acción determinada para la clase.
- Las *propiedades* permiten que los fragmentos que hacen llamadas tengan acceso al estado de la clase. Los fragmentos que hacen la llamada acceden a las propiedades con la misma sintaxis `object.identifier` que se emplea para acceder a los campos. La ventaja de las propiedades sobre los campos es que se puede escribir código que se ejecuta cuando se consultan o se asignan los valores de la propiedad. Esto permite escribir códigos de validación para evitar que se asignen nuevos valores a las propiedades o para calcular dinámicamente el valor de una propiedad que está siendo consultada. Se pueden implementar propiedades de lectura y escritura, sólo de lectura o sólo de escritura.
- Los *eventos* permiten que la clase informe a las aplicaciones que hacen la llamada cuando se produzcan determinadas acciones en su interior. Las aplicaciones que hacen la llamada pueden suscribirse a los eventos de clase y recibir avisos cuando se produzcan dichos eventos.
- Los *indizadores* permiten que se acceda a la clase como si fuera una matriz. Las aplicaciones que hacen la llamada pueden usar la sintaxis de elemento de matriz de corchetes para ejecutar el código descriptor de acceso del indizador de la clase. Los indizadores se usan cuando una clase contiene una colección de valores y es más lógico considerarla una matriz de elementos.
- Las clases pueden redefinir los operadores, como se vio en un capítulo anterior. Los operadores pueden ayudar a determinar cómo se comporta una clase cuando se usa en una expresión con un operador.
- Los *constructores* son métodos especiales que son ejecutados cuando se crean objetos en la clase. Se puede definir más de un constructor, cada uno con una lista de parámetros diferente. También se puede definir una clase sin constructores. En ese caso, el compilador de C# genera un constructor por defecto que inicializa todos los campos del objeto con el valor cero.
- Los *destructores* son métodos especiales que son ejecutados cuando se destruyen objetos en la clase. Una clase sólo puede tener un destructor. Debido a la interacción con el código .NET y el CLR, los destructores se ejecutan cuando el recolector de objetos no utilizados recoge un objeto, no cuando el código ya no puede acceder al identificador del objeto.

- Las clases pueden definir tipos propios y estos tipos pueden contener definiciones de estructura e incluso definiciones de otras clases. Una vez definidos estos tipos, la clase puede usarlos de la misma forma que los tipos enteros en C#.

La palabra clave `this` hace referencia a la instancia actual de un objeto. Se usa como prefijo para diferenciar a un identificador de campo de un identificador de parámetro con el mismo nombre.

La palabra clave `static` advierte al compilador de C# de que todos los objetos de la clase comparten una sola copia de un campo o de un objeto. Por defecto, cada campo y cada método de una clase de C# mantiene su propia copia de los valores del campo en memoria. Los elementos de la clase que no usan la palabra clave `static` reciben el nombre de *métodos de instancia*. Los elementos de clase que usan la palabra clave `static` reciben el nombre de *métodos estáticos*.



10 Cómo sobrecargar operadores

C# define el comportamiento de los operadores cuando se usan en una expresión que contiene tipos de datos integrados en C#. Por ejemplo, C# define el comportamiento del operador suma calculando la suma de dos operados y ofreciendo la suma como el valor de la expresión.

En C#, se puede definir el comportamiento de la mayoría de los operadores estándar para que puedan usarse en estructuras y clases propias. Se pueden escribir métodos especiales que definen el comportamiento de la clase cuando aparecen en una expresión que usa un operador de C#. Esto permite que las clases se puedan emplear en expresiones que parecería más lógico que escribieran otras partes del código. Supongamos, por ejemplo, que estamos escribiendo una clase que gestiona un conjunto de archivos de una base de datos. Si algún otro fragmento de código tiene dos objetos de esa clase, querrá poder escribir una expresión que sume los archivos y los almacene en un tercer objeto. Esto parece una operación de suma y parece lógico que otros fragmentos del código tengan partes como la siguiente:

```
Records Records1;  
Records Records2;  
Records Records3;  
  
Records3 = Records1 + Records2;
```

La clase `Records` puede incluir un método que especifique cuántos objetos de la clase actuarán de una determinada forma cuando se usen en expresiones con el operador de suma. Estos métodos reciben el nombre de *implementaciones de operadores definidas por el usuario* y la operación orientada a objetos para definir el comportamiento de los operadores de una clase recibe el nombre de *sobrecarga de operador*. Se emplea la palabra "sobrecarga" porque el cuerpo del código sobrecarga el significado del mismo operador y hace que se comporte de forma diferente, dependiendo del contexto en el que se use el operador.

Todos los métodos de sobrecarga de operadores deben declararse con las palabras clave `static` y `public`.

Operadores unarios sobrecargables

C# permite sobrecargar en sus clases y estructuras el comportamiento de estos operadores unarios:

- Unario más
- Unario menos
- Negación lógica
- Operador de complemento bit a bit
- Incremento prefijo
- Decremento prefijo
- La palabra clave `true`
- La palabra clave `false`

Cómo sobrecargar el unario más

Si se quiere sobrecargar el unario más, el unario menos, una negación o un operador de complemento bit a bit en una clase o estructura, hay que definir un método con las siguientes características:

- Un tipo devuelto deseado
- La palabra clave `operator`
- El operador que se quiere sobrecargar
- Una lista de parámetros que especifique un sólo parámetro del tipo o estructura que contiene el método del operador sobrecargado

Retomemos el ejemplo de la clase `Point` utilizada en un capítulo anterior. Supongamos que queremos añadir un operador unario más a la clase que, cuando

se emplee, se asegure de que las dos coordenadas del punto sean positivas. Esto se implementa en el listado 10.1.

Listado 10.1. Cómo sobrecargar el operador unario más

```
class Point
{
    public int X;
    public int Y;

    public static Point operator + (Point RValue)
    {
        Point NewPoint = new Point();

        if(RValue.X < 0)
            NewPoint.X = -(RValue.X);
        else
            NewPoint.X = RValue.X;

        if(RValue.Y < 0)
            NewPoint.Y = -(RValue.Y);
        else
            NewPoint.Y = RValue.Y;

        return NewPoint;
    }

    public static void Main()
    {
        Point MyPoint = new Point();

        MyPoint.X = -100;
        MyPoint.Y = 200;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);

        MyPoint = +MyPoint;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);
    }
}
```

El método `Main()` crea un objeto de tipo `Point` y asigna a sus coordenadas iniciales los valores (100, 200). A continuación aplica el operador unario más al objeto y vuelve a asignar el resultado al mismo punto. Por último, escribe las coordenadas `x` e `y` en la consola. En la figura 10.1 se puede ver el resultado del listado 10.1. Las coordenadas del punto han cambiado de (-100, 200) a (100, 200). El código con el operador sobrecargado se ejecuta cuando se llega a la siguiente instrucción:

```
MyPoint = +MyPoint;
```

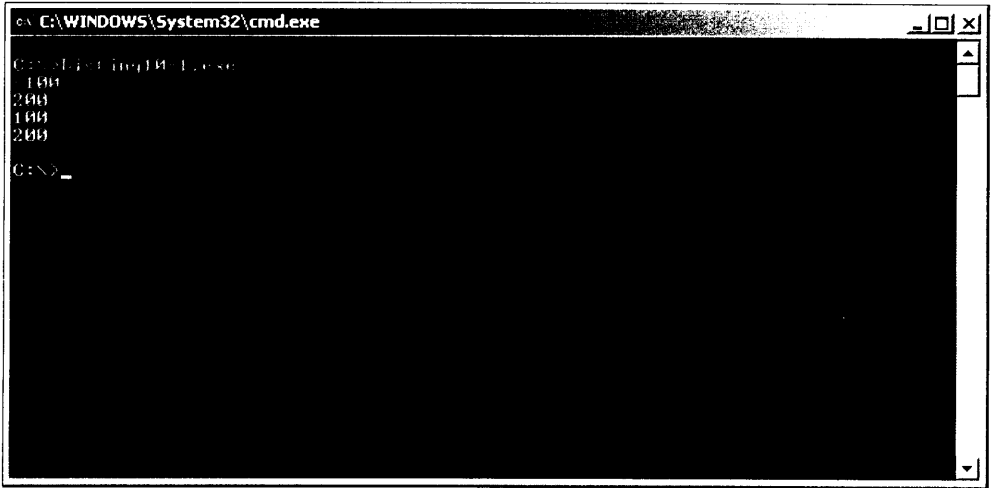


Figura 10.1. Sobrecarga del operador unario

Cuando se llega a esta instrucción, se ejecuta la sobrecarga del operador unario más para la clase `Point`. La expresión situada a la derecha del signo igual se usará como el parámetro del método.

NOTA: La expresión situada a la derecha de un operador de asignación suele ser denominada `rvalue`, que es la abreviatura de "valor derecho". La expresión a la izquierda del operador de asignación suele ser denominada `lvalue`, que es la abreviatura de "valor izquierdo". El uso de `RValue` para nombrar el método de sobrecarga de operadores determina que se está pasando el `rvalue` de la asignación. Esto es sólo una convención de designación y no un requisito. Si lo desea, puede asignar otro nombre a los parámetros usando cualquier identificador válido permitido por C#.

Este método crea un nuevo objeto `Point` y a continuación examina las coordenadas del `rvalue` proporcionado. Si alguno de los parámetros es negativo, sus valores se cambian de signo, volviéndose por tanto valores positivos, y estos nuevos valores positivos se asignan al nuevo punto. Los valores que no son negativos se asignan al nuevo punto sin ninguna conversión. A continuación el método devuelve el nuevo punto. El valor devuelto por el operador se usa como `lvalue` para la declaración original. El tipo de retorno de las sobrecargas del operador para el unario más, el unario menos, la negación o los operadores de complemento bit a bit no tienen el mismo tipo que el `rvalue`. Puede ser cualquier tipo de C# que sea adecuado para el operador.

Cómo sobrecargar el unario menos

Se puede efectuar la sobrecarga del unario menos de la misma manera que se realiza la del unario más.

El listado 10.2 sobrecarga el operador menos para gestionar la clase `Point`.

Listado 10.2. Sobrecarga del unario menos

```
class Point
{
    public int X;
    public int Y;

    public static Point operator - (Point RValue)
    {
        Point NewPoint = new Point();

        if (RValue.X > 0)
            NewPoint.X = -(RValue.X);
        else
            NewPoint.X = RValue.X;

        if (RValue.Y > 0)
            NewPoint.Y = -(RValue.Y);
        else
            NewPoint.Y = RValue.Y;

        return NewPoint;
    }
    public static void Main()
    {
        Point MyPoint = new Point();
        MyPoint.X = -100;
        MyPoint.Y = 200;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);

        MyPoint = -MyPoint;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);
    }
}
```

Tras definir el nuevo operador `Point`, simplemente se define la acción que debe realizar cuando se presente con una variable del tipo `Point`. El listado 10.2 declara la coordenada x como -100 y la coordenada y como 200 . Estos valores se escriben en la consola para una verificación visual y luego se usa el operador sobrecargado. Después de que la aplicación de ejemplo haya realizado la resta de la clase `Point`, los valores resultantes se escriben en la ventana de la consola para indicar que el comportamiento ha sido el esperado. La figura 10.2 muestra el resultado del listado 10.2.

Hasta ahora, en este capítulo hemos estudiado el unario más y el unario menos. Estos operadores efectúan operaciones sobre un valor dado (por eso se lla-

man "unarios"). Los demás operadores matemáticos básicos que pueden usarse sobre un valor se sobrecargan de la misma manera.



Figura 10.2. Sobrecarga del unario menos

En la siguiente sección se describe un operador de otro tipo, el operador de complemento bit a bit.

Cómo sobrecargar complementos bit a bit

El operador de complemento bit a bit sólo tiene definiciones para tipos `int`, `uint`, `long` y `ulong`. El listado 10.3 lo sobrecarga para trabajar con la clase `point`.

Listado 10.3. Sobrecarga del operador de complemento bit a bit

```
class Point
{
    public int X;
    public int Y;

    public static Point operator ~ (Point RValue)
    {
        Point NewPoint = new Point();
        NewPoint.X = ~RValue.X;
        NewPoint.Y = ~RValue.Y;

        return NewPoint;
    }

    public static void Main()
    {
        Point MyPoint = new Point();
```

```

MyPoint.X = 5;
MyPoint.Y = 6;
System.Console.WriteLine(MyPoint.X);
System.Console.WriteLine(MyPoint.Y);
MyPoint = ~MyPoint;
System.Console.WriteLine(MyPoint.X);
System.Console.WriteLine(MyPoint.Y);
}
}

```

El resultado de una operación de complemento bit a bit no se conoce con exactitud hasta que se ven los resultados hexadecimales de la operación. El listado 10.3 genera el complemento de los valores enteros 5 y 6.

El resultado de esta operación (que aparece en la figura 10.3) es -6 y -7, respectivamente.

Cuando se observan los valores hexadecimales de los valores de entrada y salida, es fácil deducir lo que está ocurriendo en realidad.

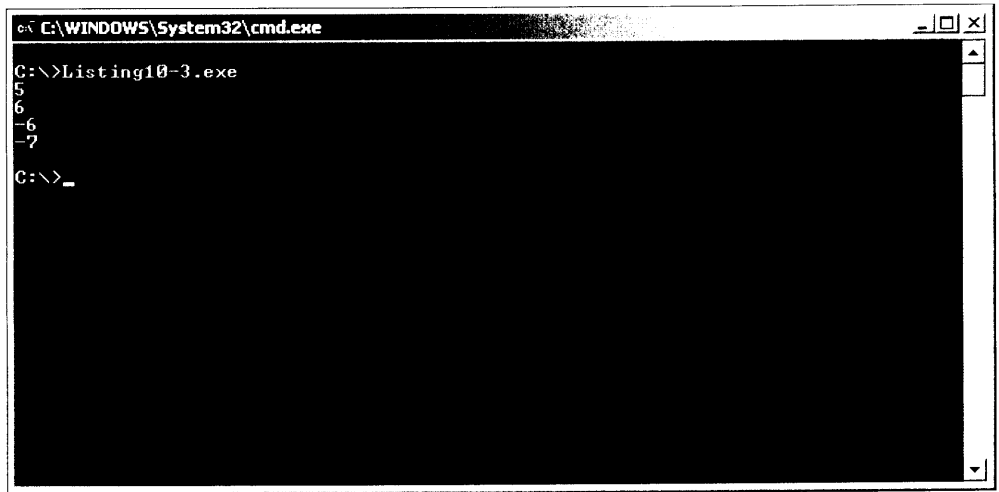


Figura 10.3. Sobrecarga de un complemento bit a bit

Tabla 10.1. Valores de entrada y salida para operaciones de complemento bit a bit

Input	Output
0x0000000000000005	0xffffffffffffA
0x0000000000000006	0xffffffffffff9

Antes de sobrecargar un operador, es necesario entender perfectamente cómo funciona. En caso contrario podría obtener resultados inesperados.

Cómo sobrecargar el incremento prefijo

Para sobrecargar los operadores de incremento prefijo o de decremento prefijo en una clase o estructura, se define un método con las siguientes características:

- Un tipo devuelto que especifica el tipo de clase o estructura que contiene el método del operador sobrecargado
- La palabra clave `operator`
- El operador que se sobrecarga
- Una lista de parámetros que especifica un solo parámetro del tipo de la clase o estructura que contiene el método del operador sobrecargado

Por ejemplo, observe el listado 10.4. Esta clase modifica la clase `Point` para sobrecargar el operador de incremento prefijo. El operador se sobrecarga para aumentar en una unidad las coordenadas `x` e `y`.

Listado 10.4. Cómo sobrecargar el incremento prefijo

```
class Point
{
    public int X;
    public int Y;

    public static Point operator ++ (Point RValue)
    {
        Point NewPoint = new Point();

        NewPoint.X = RValue.X + 1;
        NewPoint.Y = RValue.Y + 1;
        return NewPoint;
    }

    public static void Main()
    {
        Point MyPoint = new Point();

        MyPoint.X = 100;
        MyPoint.Y = 200;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);

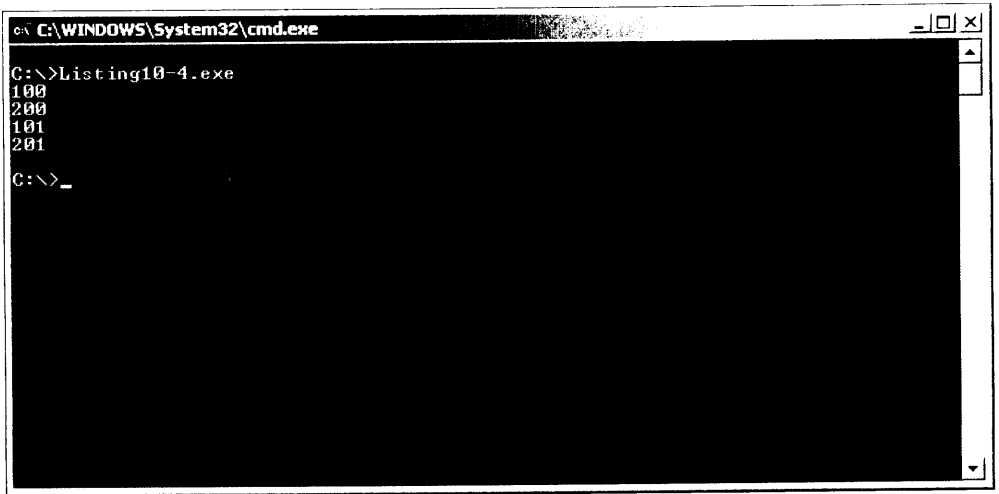
        MyPoint = ++MyPoint;
        System.Console.WriteLine(MyPoint.X);
        System.Console.WriteLine(MyPoint.Y);
    }
}
```

Si se compila y ejecuta el código del listado 10.4 se escribe lo siguiente en la consola:

```
100
200
101
201
```

Cómo sobrecargar el decremento prefijo

Ahora vamos a aprender a sobrecargar el operador de decremento para gestionar la clase `Point`. El listado 10.5 contiene el listado completo para sobrecargar el operador de forma muy parecida a como se hizo con el operador de incremento prefijo que acabamos de estudiar.



```
C:\WINDOWS\System32\cmd.exe
C:\>Listing10-4.exe
100
200
101
201
C:\>_
```

Figura 10.4. Resultado de la ejecución del código compilado del listado 10.4

Listado 10.5. Sobrecarga del operador de decremento prefijo

```
class Point
{
    public int X;
    public int Y;

    public static Point operator - (Point RValue)
    {
        Point NewPoint = new Point();

        NewPoint.X = RValue.X - 1;
        NewPoint.Y = RValue.Y - 1;
        return NewPoint;
    }

    public static void Main()
    {
        Point MyPoint = new Point();
```

```

MyPoint.X = 100;
MyPoint.Y = 200;
System.Console.WriteLine(MyPoint.X);
System.Console.WriteLine(MyPoint.Y);

MyPoint = -MyPoint;
System.Console.WriteLine(MyPoint.X);
System.Console.WriteLine(MyPoint.Y);
}
}

```

De nuevo, se pasa la coordenada *x* con el valor 100 y la coordenada *y* con el valor de 200. La figura 10.5 contiene el resultado de este programa después de que la sobrecarga del operador de decremento haya restado una unidad de *x* y de *y*.



Figura 10.5. Sobrecarga del operador de decremento prefijo

Cuando se sobrecargan operadores siempre hay que estar preparado para lo peor. Siempre hay alguna posibilidad de que los datos que se están pasando sean incorrectos y nos encontraremos con que la función sobrecargada no puede gestionar los datos. Los anteriores ejemplos no mostraban ninguna de las excepciones que pueden aparecer cuando se pasan valores erróneos o inesperados. Es recomendable experimentar con las funciones e intentar forzar errores.

Cómo sobrecargar los operadores true y false

Para sobrecargar los operadores `true` o `false` en una clase o estructura hay que definir un método con las siguientes características:

- Un tipo devuelto `bool`
- La palabra clave `operator`

- El operador que se sobrecarga
- Una lista de parámetros que especifica un solo parámetro del tipo de la clase o estructura que contiene el método del operador sobrecargado

El listado 10.6 es un buen ejemplo. Modifica la clase `Point` para que devuelva `true` si el punto está en el origen o `false` en caso contrario.

Listado 10.6. Sobrecarga de los operadores `true` y `false`

```
class Point
{
    public int X;
    public int Y;

    public static bool operator true (Point RValue)
    {
        if((RValue.X == 0) && (RValue.Y == 0))
            return true;
        return false;
    }

    public static bool operator false (Point RValue)
    {
        if((RValue.X == 0) && (RValue.Y == 0))
            return false;
        return true;
    }

    public static void Main()
    {
        Point MyPoint = new Point();

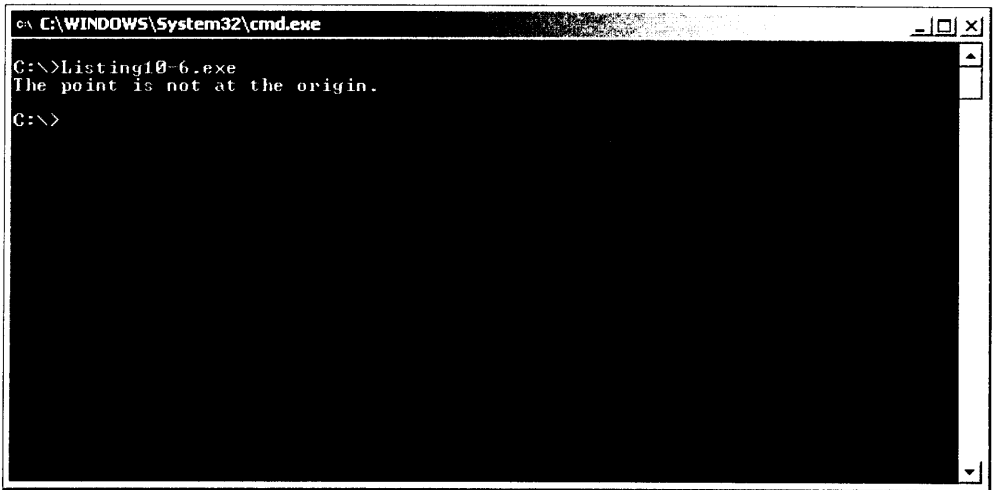
        MyPoint.X = 100;
        MyPoint.Y = 200;

        if(MyPoint)
            System.Console.WriteLine("The point is at the origin.");
        else
            System.Console.WriteLine("The point is not at the origin.");
    }
}
```

La sobrecarga de los operadores `true` y `false` permite que los objetos de la clase `Point` puedan usarse como expresiones booleanas, como en la instrucción `if`. Debido a que el objeto `MyPoint` no está en el origen, el objeto se evalúa como `false`, tal y como aparece en la figura 10.6.

Si se sobrecarga el operador `true` o el operador `false` en una clase o estructura, ambos deben ser sobrecargados. Si se sobrecarga uno de los dos, pero no el otro, el compilador de C# emite un mensaje de error como el siguiente:

error CS0216: El operador 'Point.operator true(Point)' requiere que tambien se defina un operador coincidente 'false'



```
C:\WINDOWS\System32\cmd.exe
C:\>Listing10-6.exe
The point is not at the origin.
C:\>
```

Figura 10.6. Sobrecarga de los operadores true y false

Operadores binarios sobrecargables

Estos son los operadores binarios que se pueden sobrecargar:

- Suma
- Resta
- Multiplicación
- División
- Resto
- AND
- OR
- OR exclusivo
- Desplazamiento a la izquierda
- Desplazamiento a la derecha
- Igualdad
- Desigualdad
- Mayor que
- Menor que

- Mayor o igual que
- Menor o igual que

Si quiere sobrecargar cualquiera de los operadores binarios de una clase o estructura, hay que definir un método con las siguientes características:

- Un tipo devuelto deseado
- La palabra clave `operator`
- El operador que se sobrecarga
- Una lista de parámetros que especifica dos parámetros, al menos uno de los cuales debe ser del tipo de la clase o estructura que contiene el método del operador sobrecargado

La sobrecarga de operadores binarios brinda mucha flexibilidad. Pueden usarse diferentes parámetros para los dos parámetros de la lista, lo que significa que se puede aplicar el operador a dos valores de diferentes tipos si se desea. También se puede usar cualquier tipo disponible como el valor devuelto por el operador sobrecargado. Si se quiere sumar un objeto y un valor de coma flotante para obtener un resultado booleano, se puede escribir un método sobrecargado como el siguiente:

```
static public bool operator + (Point MyPoint, float FloatValue)
```

Se pueden definir varias sobrecargas para el mismo operador, pero sólo si las listas de parámetros usan tipos diferentes:

```
static public bool operator + (Point MyPoint, float FloatValue)
static public bool operator + (Point MyPoint, int IntValue)
static public bool operator + (Point MyPoint, uint UIntValue)
```

El listado 10.7 añade los operadores de igualdad y de desigualdad a la clase `Point`. El operador devuelve resultados booleanos que devuelven `true` si los dos objetos `Point` tienen las mismas coordenadas; en caso contrario, devuelven `false`.

Listado 10.7. Sobrecarga de los operadores de igualdad y desigualdad

```
class Point
{
    public int X;
    public int Y;

    public static bool operator == (Point Point1, Point Point2)
    {
        if(Point1.X != Point2.X)
            return false;
        if(Point1.Y != Point2.Y)
```

```

        return false;
    return true;
}

public override bool Equals(object o)
{
    return true;
}

public override int GetHashCode()
{
    return 0;
}

public static bool operator != (Point Point1, Point Point2)
{
    if(Point1.X != Point2.X)
        return true;
    if(Point2.Y != Point2.Y)
        return true;
    return false;
}

public static void Main()
{
    Point MyFirstPoint = new Point();
    Point MySecondPoint = new Point();
    Point MyThirdPoint = new Point();

    MyFirstPoint.X = 100;
    MyFirstPoint.Y = 200;

    MySecondPoint.X = 500;
    MySecondPoint.Y = 750;

    MyThirdPoint.X = 100;
    MyThirdPoint.Y = 200;

    if(MyFirstPoint == MySecondPoint)
        System.Console.WriteLine("MyFirstPoint and
MySecondPoint are at the same coordinates.");
    else
        System.Console.WriteLine("MyFirstPoint and
MySecondPoint are not at the same coordinates.");

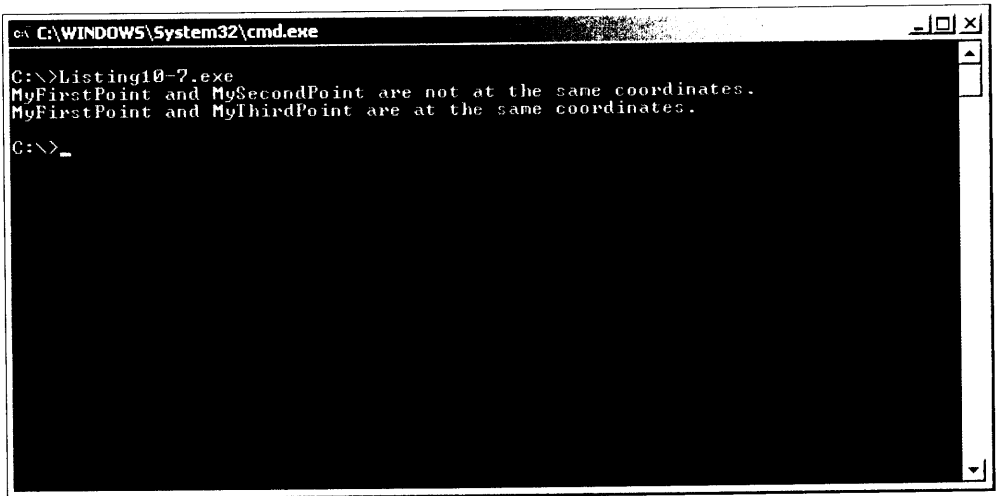
    if(MyFirstPoint == MyThirdPoint)
        System.Console.WriteLine("MyFirstPoint and MyThirdPoint
are at the same coordinates.");
    else
        System.Console.WriteLine("MyFirstPoint and MyThirdPoint
are not at the same coordinates.");
}
}

```

El método `Main()` define tres puntos:

- `MyFirstPoint`, con coordenadas (100, 200)
- `MySecondPoint`, con coordenadas (500, 750)
- `MyThirdPoint`, con coordenadas (100, 200)

A continuación el método usa el operador de igualdad para determinar si los puntos `MyFirstPoint` y `MySecondPoint` hacen referencia a la mismas coordenadas. Entonces usa el operador de igualdad para determinar si los puntos `MyFirstPoint` y `MySecondPoint` hacen referencia a las mismas coordenadas. En la figura 10.7 se muestra el resultado que se obtiene si se compila y ejecuta el código del listado 10.7.



```
ca C:\WINDOWS\System32\cmd.exe
C:\>Listing10-7.exe
MyFirstPoint and MySecondPoint are not at the same coordinates.
MyFirstPoint and MyThirdPoint are at the same coordinates.
C:\>_
```

Figura 10.7. Sobrecarga de los operadores de igualdad y desigualdad

Los siguientes pares de operadores deben ser sobrecargados conjuntamente:

- Igualdad y desigualdad
- Menor y mayor que
- Menor o igual que y mayor o igual que

Si se sobrecarga uno de estos pares pero no el otro, el compilador de C# emite un mensaje como el siguiente:

```
error CS0216: El operador 'Point.operator ==(Point, Point)'
requiere que también se defina un operador coincidente '!='
```

Operadores de conversión sobrecargables

También se pueden escribir métodos de sobrecarga de operadores que conviertan un tipo en otro. El método de sobrecarga también puede definir si el compilador

de C# debe tratar la conversión como implícita o explícita. Si quiere definir un nuevo operador de conversión en una clase o estructura, debe definir un método con las siguientes características:

- La palabra clave `implicit` si la conversión va a considerarse una conversión implícita o la palabra clave `explicit` si la conversión va a considerarse una conversión explícita
- La palabra clave `operator`
- Un tipo que especifica el tipo al que se va a hacer la conversión
- Una lista de parámetros que especifica el tipo original de la conversión

El listado 10.8 define una conversión implícita de un objeto de la clase `Point` a `double`. El tipo `double` especifica la distancia desde el origen hasta el punto, usando el teorema de Pitágoras.

Listado 10.8. Cómo definir una conversión implícita

```
class Point
{
    public int X;
    public int Y;

    public static implicit operator double(Point RValue)
    {
        double Distance;
        double Sum;

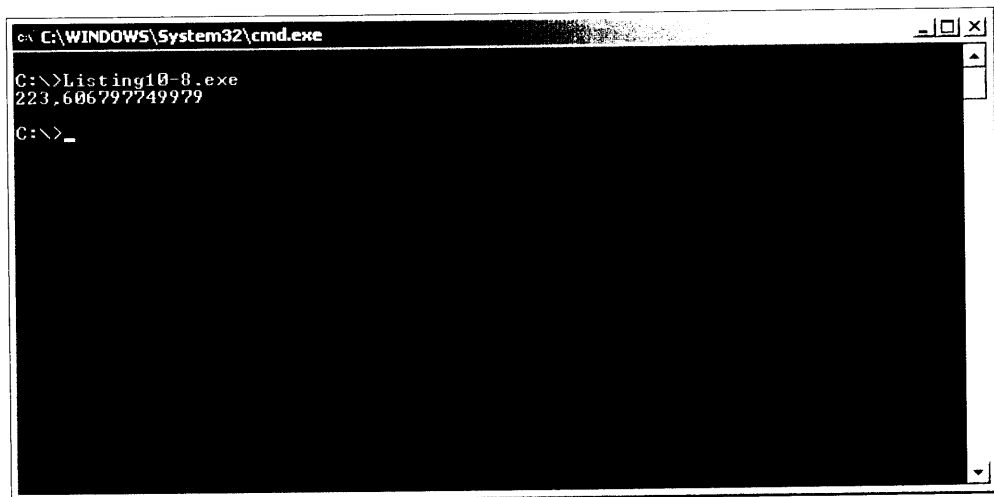
        Sum = (RValue.X * RValue.X) + (RValue.Y * RValue.Y);
        Distance = System.Math.Sqrt(Sum);
        return Distance;
    }

    public static void Main()
    {
        double Distance;
        Point MyPoint = new Point();

        MyPoint.X = 100;
        MyPoint.Y = 200;
        Distance = MyPoint;
        System.Console.WriteLine(Distance);
    }
}
```

NOTA: .NET Framework define el método `System.Math.Sqrt()` que calcula la raíz cuadrada del parámetro proporcionado. El método es estático, por lo que se le puede llamar aunque no se tenga un objeto del tipo `System.Math` para llamarlo.

El método `Main()` declara un objeto del tipo `Point` y asigna a sus coordenadas los valores (100, 200). A continuación asigna el objeto a una variable de tipo `double`. lo que está permitido porque la clase `Point` define un operador de conversión que convierte un objeto `Point` a `double`. Como el operador de conversión está definido como una conversión implícita, no es necesaria una conversión explícita. A continuación el método `Main()` escribe el valor de `double` convertido en la consola. La figura 10.8 muestra el resultado del listado 10.8.



```
ex C:\WINDOWS\System32\cmd.exe
C:\>Listing10-8.exe
223,606797749979
C:\>_
```

Figura 10.8. Definición de una conversión implícita

Operadores que no pueden sobrecargarse

C# no permite redefinir el comportamiento de los operadores de la siguiente lista. Esto se hace principalmente en beneficio de la simplicidad. Los programadores de C# quieren que estos operadores no dejen de ser sencillos y que siempre realicen la misma función; por tanto, no está permitido sobrecargarlos.

- Asignación
- AND condicional
- OR condicional
- Condicional
- Las palabras clave `new`, `typeof`, `sizeof` e `is`

Resumen

C# permite personalizar el comportamiento de varios de los operadores integrados. Las clases y las estructuras pueden incluir métodos llamados *métodos de*

sobrecarga de operador que definen el comportamiento de un operador cuando aparece en una expresión con la clase o estructura.

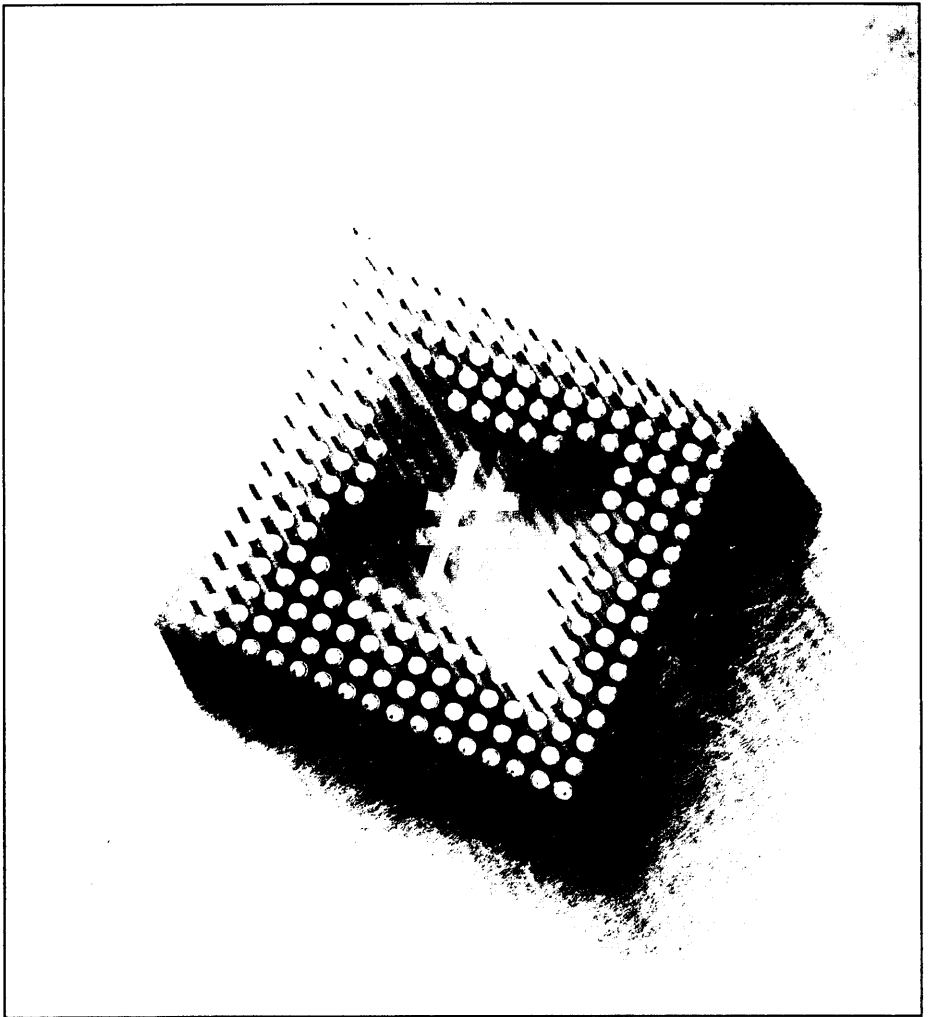
Para sobrecargar los operadores unario más, unario menos, de negación o de complemento bit a bit en una clase o estructura, hay que definir un método con un tipo devuelto deseado, el operador que se está sobrecargando y un solo parámetro del tipo de la clase o estructura que contiene el método de operador sobrecargado.

Para sobrecargar los operadores de incremento o decremento prefijo en una clase o estructura, hay que definir un método con un tipo de devolución que especifica el tipo de clase o estructura que contiene el método del operador sobrecargado. También es necesario definir el operador que se está sobrecargando y un solo parámetro del tipo de clase o estructura que contiene el método del operador sobrecargado.

Para sobrecargar los operadores `true` o `false` en una clase o estructura, hay que definir un método con un tipo de devolución booleano y un solo parámetro del tipo de la clase o estructura que contiene el método de operador sobrecargado.

Para sobrecargar cualquiera de los operadores binarios en una clase o estructura, hay que definir un método con un tipo de devolución, el operador que se está sobrecargando y dos parámetros. Al menos uno de los dos parámetros debe ser del tipo de clase o estructura que contiene el método del operador sobrecargado.

También se pueden definir nuevas conversiones para las clases o estructuras. Se especifica si la conversión se considerará un operador implícito o explícito. El método del operador de conversión especifica el tipo de la variable que se convierte y el tipo al que debe ser convertida.



11 Herencia de clase

Los programas más simples de C# pueden usar una o dos clases. Sin embargo, probablemente se necesiten varias clases en los programas más grandes. Muchas de estas clases pueden tener campos o métodos similares y sería lógico compartir el código común entre el conjunto de clases.

C# incluye el concepto orientado a objetos de *herencia*, que permite que una clase adopte código de otras clases. Las clases de C# pueden derivarse de las clases primarias y las instrucciones heredadas pueden ser usadas en otras clases.

La herencia se usa en el desarrollo de software orientado a objetos para reutilizar los códigos más comunes. Observe, por ejemplo, los cuadros de lista de selección múltiple y de selección simple de Windows. Estos cuadros de lista tienen diferentes funcionalidades; uno permite que se seleccionen varios elementos y el otro lo impide, pero también tienen muchas similitudes. Tienen el mismo aspecto, se comportan de la misma forma cuando el usuario se desplaza por la lista y usan el mismo color para marcar un elemento seleccionado. Si hubiera que escribir estos dos cuadros de lista como clases de C#, se podrían escribir por separado, sin que uno conozca la existencia del otro. Sin embargo, eso sería un desperdicio. La mayor parte del código seguramente sea idéntico. Sería más lógico escribir una clase que contuviera el código común y disponer de clases derivadas de la clase de código común y que implementasen las diferentes funcionalidades. Se puede escribir una clase llamada `ListBox` para que, por

ejemplo, contenga el código común y, a continuación, escribir una clase de C# llamada `SingleSelectionListBox` que herede de `ListBox` y proporcione el código único al cuadro de lista de selección simple. También se puede escribir una clase de C# llamada `MultipleSelectionListBox` que también herede de `ListBox` pero que proporcione el código único al cuadro de lista de selección simple. Otra de las ventajas consiste en que, si encuentra un error en el cuadro de lista, se le puede seguir fácilmente la pista hasta el error en el código común. Si se puede reparar el error en el código común, al volver a compilar el programa se repararán esos errores en las clases de cuadros de lista de selección múltiple y selección simple. Basta reparar un error para solucionar el problema en las dos clases. En terminología orientada a objetos, se habla de herencia en términos de clase base y clase derivada. La clase de la que se hereda recibe el nombre de *clase base* y la clase que hereda de la clase base recibe el nombre de *clase derivada*. En el ejemplo de los cuadros de lista, la clase `ListBox` es la clase base y las clases `SingleSelectionListBox` y `MultipleSelectionListBox` son las clases derivadas.

Cómo compilar con clases múltiples

Trabajar con herencias en C# significa que se va a trabajar con más de una clase de C#. C# no es muy estricto respecto a cómo se relacionan estas clases con los archivos fuente. Se pueden poner todas las clases en un solo archivo fuente o se puede poner cada clase en un archivo fuente diferente.

Obviamente, excepto en los programas más pequeños, implementar todas las clases en un solo archivo no es un buen modo de organizar el código. Hay que tener en cuenta que todas las clases se recompilan cada vez que se hace un cambio en alguna parte del programa. Para compilar un programa que usa archivos fuente separados desde una línea de comandos, tan sólo hay que escribir cada archivo después del nombre del compilador, como se ve en el ejemplo:

```
csc file1.cs file2.cs file3.cs
```

Por defecto, el compilador de C# da al ejecutable resultante el nombre del primer archivo fuente. La anterior línea de comandos produce un ejecutable llamado `file1.exe`. Se puede usar el argumento `/out` para cambiar el nombre del archivo:

```
csc /out:myapp.exe file1.cs file2.cs file3.cs
```

Esta línea de comandos del compilador genera un ejecutable llamado `myapp.exe`.

NOTA: Una, y sólo una, de sus clases debe especificar un método estático `Main()`.

Cómo especificar una clase base en C#

Volvamos al ejemplo de la clase `Point` para ver cómo funciona la herencia en C#. Supongamos que hemos designado una clase llamada `Point2D`, que describe un punto en un espacio bidimensional con las coordenadas X e Y:

```
class Point2D
{
    public int X;
    public int Y;
    // más código
}
```

Ahora supongamos que queremos trabajar con puntos en un espacio tridimensional, pero manteniendo la clase `Point2D`. La herencia nos permite crear una nueva clase que mantiene todo el código de la clase `Point2D` y le añade una coordenada Z.

Para nombrar la clase base de C# se escribe el nombre de la clase derivada seguido por dos puntos y del nombre de la clase base. A continuación, se incluye un ejemplo de cómo se derivaría la clase `Point3D` a partir de la clase `Point2D`:

```
class Point3D : Point2D
{
    public int Z;
    // código para la clase Point3D
}
```

Dependiendo de las reglas de ámbito de la clase base, todos los campos y propiedades de la clase base (`Point2D`) pueden ser empleadas en la clase derivada (`Point3D`). Por ejemplo, cuando una clase se deriva de una clase base, el código de la clase derivada puede acceder a los campos y propiedades de la clase base, si el ámbito lo permite.

Sólo se puede escribir una clase cuando una clase se hereda de otra. Algunos lenguajes orientados a objetos, como C++, permiten especificar más de una clase base para una clase derivada. Este concepto recibe el nombre de *herencia múltiple*. C# admite la herencia simple, pero no la múltiple. En el apartado dedicado a la contención se explica una técnica para simular herencias múltiples en C#.

El listado 11.1 enseña a usar las clases `Point3D` y `Point2D` juntas.

Listado 11.1. Cómo derivar `Point3D` a partir de `Point2D`

```
class Point2D
{
    public int X;
    public int Y;
}

class Point3D : Point2D
```



```

{
    public int Z;
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();
        Point3D My3DPoint = new Point3D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        My3DPoint.X = 150;
        My3DPoint.Y = 250;
        My3DPoint.Z = 350;
    }
}

```

El método `Main()` crea un objeto `Point2D` y otro `Point3D`. El objeto `Point3D` tiene campos para las coordenadas X, Y y Z, aunque la declaración de `Point3D` sólo declare un campo llamado Z. Los campos X e Y se heredan de la clase base `Point2D` y pueden ser usados exactamente igual que si se hubieran declarado directamente en la clase `Point3D`.

Ámbito

Al diseñar la estructura de la herencia de clase, puede decidir qué miembros de la clase base no deben ser visibles para las clases derivadas o para los demás programadores. Por ejemplo, se puede escribir un método en una clase base que ayude a calcular un valor. Si ese cálculo no es de utilidad en una clase derivada, se puede evitar que la clase derivada llame al método.

En terminología de la programación, la visibilidad de una variable o método se conoce como su *ámbito*. Algunas variables o métodos pueden ser declaradas como de ámbito público, otras pueden ser declaradas como de ámbito privado y otras pueden estar entre estos dos casos.

C# define cinco palabras clave que permiten definir el ámbito de cualquier miembro (variable o método) de una clase. El ámbito de un miembro afecta a su visibilidad para las clases derivadas y el código que crea las instancias de la clase. Estas palabras clave, resaltadas en la siguiente lista, se colocan antes de cualquier otra palabra clave en una declaración de miembro.

- Los miembros marcados como `public` son visibles para las clases derivadas y para el código que crea los objetos de la clase. Hasta ahora hemos usado `public`.

- Los miembros marcados como `private` sólo son visibles para la clase en la que se definen. Los miembros privados no son accesibles desde las clases derivadas ni desde el código que crea los objetos de la clase.
- Los miembros marcados como `protected` sólo son visibles para la clase en la que se definen o desde las clases derivadas de esa clase. No se puede acceder a los miembros protegidos desde el código que crea los objetos de su clase.
- Los miembros marcados como `internal` son visibles para cualquier código en el mismo archivo binario, pero no son visibles para otros archivos binarios. Recuerde que .NET Framework acepta el concepto de ensamblados, que son bibliotecas de código ya compiladas que pueden ser usadas por aplicaciones externas. Si se escribe una clase en C# y se compila la clase para obtener un ensamblado, cualquier fragmento de código del ensamblado podrá acceder a los miembros de clase interna. Sin embargo, si otro fragmento de código usa ese ensamblado, no tendrá acceso al miembro, aunque derive una clase de la clase del ensamblado.
- Los miembros marcados como `protected internal` son visibles para todo el código incluido en el mismo archivo binario y para las clases externas que se deriven de su clase. Si se escribe una clase en C# y se compila la clase para obtener un ensamblado, cualquier fragmento de código del ensamblado puede acceder a los miembros de clase interna. Si otro fragmento de código externo usa el ensamblado y deriva una clase de la clase del ensamblado, el miembro interno protegido será accesible para la clase derivada. Sin embargo, el código que trabaja con los objetos de la clase base no tendrá acceso al miembro.

C# permite especificar un miembro de clase sin especificar ninguna palabra clave de ámbito.

Si se declara un miembro de clase sin especificar ninguna palabra clave de ámbito, al miembro se le asigna por defecto accesibilidad privada. Los miembros que se han declarado sin usar ninguna palabra clave de ámbito pueden usarse en otras partes de la clase, pero no pueden ser usados por clases derivadas ni por código que use objetos de la clase.

Cómo reutilizar identificadores de miembros en las clases derivadas

C# permite reutilizar identificadores de clase base en las clases derivadas, pero el compilador de C# emite un aviso cuando lo detecta. Preste atención al código del listado 11.2.

Listado 11.2. Cómo reutilizar identificadores de clase base

```
class Point2D
{
    public int X;
    public int Y;
}

class Point3D : Point2D
{
    public int X;
    public int Y;
    public int Z;
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();
        Point3D My3DPoint = new Point3D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        My3DPoint.X = 150;
        My3DPoint.Y = 250;
        My3DPoint.Z = 350;
    }
}
```

La clase derivada `Point3D` define los campos `X` e `Y` que coinciden con los identificadores usados en la clase base `Point2D`. El compilador de `C#` emite las siguientes advertencias cuando se compila este código:

```
warning CS0108: La palabra clave new es necesaria en
'Point3D.X' porque oculta el miembro heredado 'Point2D.X'
warning CS0108: La palabra clave new es necesaria en
'Point3D.Y' porque oculta el miembro heredado 'Point2D.Y'
```

El compilador de `C#` emite avisos porque los identificadores de la clase derivada ocultan las definiciones que usan el mismo identificador en la clase base. Si se quieren reutilizar los nombres, pero no que el compilador emita avisos, se puede usar el operador `new` al reutilizar los identificadores en la clase derivada. El código del listado 11.3 compila sin emitir avisos.

Listado 11.3. Cómo usar `new` para reutilizar identificadores de clase

```
class Point2D
{
    public int X;
    public int Y;
```

```

}

class Point3D : Point2D
{
    new public int X;
    new public int Y;
    public int Z;
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();
        Point3D My3DPoint = new Point3D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        My3DPoint.X = 150;
        My3DPoint.Y = 250;
        My3DPoint.Z = 350;
    }
}

```

Cómo trabajar con métodos heredados

C# permite que los métodos de la clase base y de las clases derivadas se relacionen de varios modos. C# permite los siguientes métodos:

- Métodos virtuales y de reemplazo
- Métodos abstractos

Métodos virtuales y de reemplazo

Quizás quiera que una clase derivada cambie la implementación de un método en una clase base, pero manteniendo el nombre del método. Suponga, por ejemplo, que la clase `Point2D` implementa un método llamado `PrintToConsole()`, que escribe las coordenadas `X` e `Y` del punto en la consola. Quizás también quiera que la clase derivada `Point3D` proporcione su propia implementación de `PrintToConsole()`. Sin embargo, no puede usar el método `PrintToConsole()` de la clase `Point2D`, porque esa implementación sólo funciona con las coordenadas `X` e `Y` y la clase `Point3D` también tiene una coordenada `Z`. La clase `Point3D` debe facilitar su propia implementación del mismo método `PrintToConsole()`. Los nombres de métodos pueden reutilizarse en clases derivadas si el método de clase base permite que el método pueda volver a ser

implementado. La operación de reimplementar un método de clase base en una clase derivada recibe el nombre de *reemplazar* el método de clase base. Al reemplazar un método de clase base en C# hay que tener en cuenta los dos requisitos:

- El método de clase base debe declararse con la palabra clave `virtual`.
- El método de la clase derivada debe declararse con la palabra clave `override`.

Los métodos de clase base que usan la palabra clave `virtual` reciben el nombre de *métodos virtuales* y los de clase base que usan la palabra clave `override` reciben el nombre de *métodos de reemplazo*. El listado 11.4 demuestra cómo puede implementarse el método `PrintToConsole()` para las clase `Point2D` y `Point3D`.

Listado 11.4. Cómo reemplazar métodos virtuales

```
class Point2D
{
    public int X;
    public int Y;

    public virtual void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}", X, Y);
    }
}

class Point3D : Point2D
{
    public int Z;

    public override void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}, {2}", X, Y, Z);
    }
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();
        Point3D My3DPoint = new Point3D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        My3DPoint.X = 150;
        My3DPoint.Y = 250;
        My3DPoint.Z = 350;
    }
}
```

```

        My2DPoint.PrintToConsole();
        My3DPoint.PrintToConsole();
    }
}

```

NOTA: La sintaxis de las llamadas `WriteLine()` hechas en el listado 11.4 es diferente de la sintaxis usada con anterioridad. Los números entre llaves de la cadena son comodines. Los valores de los otros parámetros se escriben en la consola en lugar del comodín. El comodín `{0}` es reemplazado por el valor del primer parámetro, el comodín `{1}` es reemplazado por el valor del segundo parámetro y así sucesivamente.

El listado 11.4 escribe esto en la consola:

```

(100, 200)
(150, 250, 350)

```

No se puede reemplazar un método de clase base a menos que use la palabra clave `virtual`. Si intenta hacerlo sin usar la palabra clave, el compilador de C# emite el siguiente error:

```

error CS0506: 'Point3D.PrintToConsole()' : no se puede reemplazar el miembro heredado 'Point2D.PrintToConsole()' porque no está marcado como virtual, abstract u override

```

Sin embargo, está permitido reemplazar un método `override`. Si, por alguna extraña razón, se quiere implementar una clase `Point4D` y derivarla de `Point3D`, es posible reemplazar el método de `Point3D` `PrintToConsole()`.

Polimorfismo

El concepto de reemplazo de métodos lleva al concepto de *polimorfismo*. Cuando reemplazamos un método, queremos llamar al método apropiado desde cualquier método que llame a este método reemplazado.

El listado 11.5 presenta este concepto en acción. Se ha añadido a `Point2D` un método `UsePrintToConsole()` que llama al método virtual `PrintToConsole()`. `Point3D` hereda este método de `Point2D`. Cuando se llama a `PrintToConsole()` en esta función, se quiere llamar a la versión que pertenece a la clase apropiada. En otras palabras, en el método `UsePrintToConsole()` que pertenece a la clase `Point2D`, se pretende llamar al método `PrintToConsole()` que pertenece a la clase `Point2D`. En el método `UsePrintToConsole()` que pertenece a la clase `Point3D`, se pretende llamar al método reemplazado `PrintToConsole()` que pertenece a la clase `Point3D`. Como el método `PrintToConsole()` fue declarado como

un método virtual. la detección de la versión que debe ejecutarse tiene lugar automáticamente. El listado 11.5 escribe lo siguiente en la consola:

```
(100, 200)
(150, 250, 350)
```

Listado 11.5. Polimorfismo

```
class Point2D
{
    public int X;
    public int Y;

    public virtual void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}", X, Y);
    }

    public void UsePrintToConsole()
    {
        PrintToConsole();
    }
}

class Point3D : Point2D
{
    public int Z;

    public override void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}, {2}", X, Y, Z);
    }
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();
        Point3D My3DPoint = new Point3D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        My3DPoint.X = 150;
        My3DPoint.Y = 250;
        My3DPoint.Z = 350;

        My2DPoint.UsePrintToConsole();
        My3DPoint.UsePrintToConsole();
    }
}
```

Métodos abstractos

Algunas clases base pueden no ser capaces de proporcionar la implementación de un método, pero puede que queremos que las clases derivadas proporcionen una implementación. Supongamos, por ejemplo, que estamos escribiendo en C# una aplicación de geometría y escribimos clases llamadas `Square` y `Circle`. Decidiremos las funciones comunes que usará cada forma de la aplicación y por lo tanto implementamos una clase base llamada `Shape` y derivamos las clases `Square` y `Circle` de `Shape`:

```
class Shape
{
}

class Circle : Shape
{
}

class Square : Shape
{
}
```

Ahora supongamos que decidimos que todas las formas deben ser capaces de calcular su área, de modo que escribimos un método llamado `GetArea()`. El problema de escribir ese código en la clase base es que la clase base no tiene suficiente información para calcular un área. Cada forma calcula su área usando una fórmula diferente.

Lo que podemos hacer es definir un método abstracto en la clase base `Shape`. Los métodos abstractos no proporcionan una implementación propia sino que proporcionan una firma de método que las clases derivadas deben implementar. Los métodos abstractos dicen "Yo no sé implementar este método, pero mi clase derivada lo hará, de modo que asegúrese de que la implementen con los parámetros y el código devuelto que yo especifico." Los siguientes fragmentos demuestran cómo declarar un método abstracto en la clase `Shape`.

```
abstract class Shape
{
    public abstract double GetArea();
}
```

NOTA: Las clases abstractas usan la palabra clave `abstract`. No tienen cuerpo de método; en su lugar hay un punto y coma después de la lista de parámetros.

Las clases abstractas también son, por definición, métodos virtuales y debe usarse la palabra clave `override` para reemplazarlos por clases derivadas:


```

class Square : Shape
{
    public override double GetArea()
    {
        // implemente el calculo del área
    }
}

```

Las clases que contienen al menos un método abstracto reciben el nombre de clases abstractas y deben incluir la palabra clave `abstract` antes de la palabra clave de la clase. Si no se incluye la palabra clave `abstract` al definir la clase se obtendrá un error del compilador de C#:

```

error CS0513: 'Shape.GetArea ()' es abstract pero está incluida
en la clase nonabstract 'Shape'

```

El compilador de C# no permite crear objetos a partir de clases abstractas. Si se intenta el compilador de C# emite un error:

```

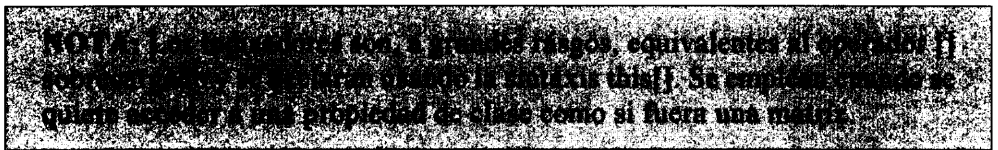
error CS0144: No se puede crear una instancia de la clase o
interfaz abstracta 'Shape'

```

Las clases abstractas suelen usarse para crear una clase base común a un conjunto de clases. Esto permite usar el polimorfismo al almacenar clases derivadas en algún tipo de colección, como se vio en un capítulo anterior.

Clases base: Cómo trabajar con propiedades e indizadores heredados

En C#, se pueden marcar como `virtual`, `override` o `abstract`, propiedades e indizadores de clases base y derivadas, igual que si fueran métodos.



Las propiedades `virtual` y `override` y los indizadores funcionan como las propiedades `virtual` y `override`. Las propiedades y los indizadores pueden marcarse como virtuales en una clase base y como reemplazados en una clase derivada. Las clases base pueden definir propiedades e indizadores, que no tienen implementación propia. Las clases base que contienen al menos una propiedad abstracta o un indizador deben ser marcadas como si fueran una clase abstracta. Las propiedades abstractas y los indizadores deben ser reemplazados en una clase base.

Cómo usar la palabra clave base

C# proporciona la palabra clave `base` para que las clases derivadas puedan acceder a las funciones de su clase base. Se puede usar la palabra clave `base` para llamar a un constructor de clase base cuando se crea un objeto de una clase derivada. Para llamar a un constructor de clase base hay que colocar después del constructor de la clase derivada dos puntos, la palabra clave `base` y a continuación los parámetros que se van a pasar a la clase base. El listado 11.6 demuestra el funcionamiento de todo esto. Añade constructores para las clases `Point2D` y `Point3D` y el constructor `Point3D` llama al constructor de su clase base.

Listado 11.6. Cómo llamar a constructores de clase base

```
class Point2D
{
    public int X;
    public int Y;

    public Point2D(int X, int Y)
    {
        this.X = X;
        this.Y = Y;
    }

    public virtual void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}", X, Y);
    }
}

class Point3D : Point2D
{
    public int Z;

    public Point3D(int X, int Y, int Z) : base(X, Y)
    {
        this.Z = Z;
    }

    public override void PrintToConsole()
    {
        System.Console.WriteLine("{0}, {1}, {2}", X, Y, Z);
    }
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D(100, 200);
        Point3D My3DPoint = new Point3D(150, 250, 350);
    }
}
```

```

        My2DPoint.PrintToConsole();
        My3DPoint.PrintToConsole();
    }
}

```

El constructor de la clase `Point2D` establece los campos `X` e `Y` de la clase mediante los dos enteros que se pasan al constructor. El constructor de la clase `Point3D` admite tres parámetros. Los primeros dos parámetros se pasan al constructor de la clase base usando la palabra clave `base` y el tercero se usa para establecer el valor del campo `Z` de la clase derivada.

Cómo acceder a campos de clase base con la palabra clave base

También se puede usar la palabra clave `base` para acceder a miembros de la clase base. Para trabajar con un miembro de clase base en la clase derivada, hay que anteponer al nombre del miembro la palabra clave `base` y un punto. Se puede acceder a los campos de la clase base mediante la siguiente sintaxis:

```
base.X = 100;
```

También se puede invocar a métodos de clase base con esta otra sintaxis:

```
base.PrintToConsole();
```

Clases selladas

Si no quiere que se derive código de una determinada clase, puede marcar la clase con la palabra clave `sealed`. No se puede derivar una clase de una clase sellada. Se puede especificar una clase sellada escribiendo la palabra clave `sealed` antes de la palabra clave `class`, como en este ejemplo:

```
sealed class MySealedClass
```

Si se intenta derivar una clase de una clase derivada, el compilador de C# emite un error:

```
error CS0509: 'Point3D' : no se puede heredar de la clase sealed 'Point2D'
```

Contención y delegación

Si la herencia es una relación ES-UN, la *contención* es una relación TIENE-UN. Un gato de Birmania ES UN gato (por lo que puede heredar la clase

DeBirmania de la clase genérica Gato): pero un Coche TIENE 4 ruedas (por lo que la clase Coche puede tener cuatro objetos Rueda). El aspecto más interesante de la contención es que se puede emplear como sustituto de la herencia. El principal inconveniente de usar la contención en lugar de la herencia es que se pierden las ventajas del polimorfismo. Sin embargo, se obtienen los beneficios de la reutilización del código.

En C#, hay dos casos comunes en los que prácticamente sólo se puede emplear la contención y no la herencia: cuando se trabaja con herencias múltiples y cuando se trabaja con clases selladas. A continuación se incluye un ejemplo que muestra cómo funciona esta técnica. Además, verá trabajar al polimorfismo.

Supongamos que tenemos una clase `AlarmClock` y una clase `Radio` como las que aparecen en el siguiente fragmento y queremos crear una clase `ClockRadio` que combine estas dos clases. Si C# admitiese herencias múltiples, se podría hacer que `ClockRadio` heredase de las clases `AlarmClock` y `Radio`. A continuación se podría añadir una variable booleana `radioAlarm` que determine si se activa la alarma o la radio y que reemplace `SoundAlarm()` para usar esta variable. Por desgracia, C# no admite herencias múltiples. Por suerte, se puede emplear la contención en lugar de la herencia y obtener todas las ventajas de la reutilización de código. Observe cómo funciona, paso a paso:

```
class Radio
{
    protected bool on_off;

    public void On()
    {
        if (!on_off) Console.WriteLine("Radio is now on!");
        on_off = true;
    }

    public void Off()
    {
        if (on_off) Console.WriteLine("Radio is now off!");
        on_off = false;
    }
}

class AlarmClock
{
    private int currentTime;
    private int alarmTime;

    private void SoundAlarm()
    {
        Console.WriteLine("Buzz!");
    }

    public void Run()
    {
```

```

    for (int currTime = 0; currTime < 43200; currTime++)
    {
        SetCurrentTime(currTime);
        if (GetCurrentTime() == GetAlarmTime())
        {
            Console.WriteLine("Current Time = {0}!", currentTime);
            SoundAlarm();
            break;
        }
    }
}

public int GetCurrentTime()
{
    return currentTime;
}

public void SetCurrentTime(int aTime)
{
    currentTime = aTime;
}

public int GetAlarmTime()
{
    return alarmTime;
}

public void SetAlarmTime(int aTime)
{
    alarmTime = aTime;
}
}

```

Como queremos reemplazar el método `SoundAlarm()` de `AlarmClock`, es recomendable hacer que `ClockRadio` herede de `AlarmClock`. Esto requiere un pequeño cambio en la implementación de `AlarmClock`. Sin embargo, a cambio se consiguen todos los beneficios del polimorfismo. Una vez que se ha seleccionado una clase base, no podemos heredar de `Radio`. En lugar de heredar, crearemos una variable de miembro privada `Radio` dentro de la clase `ClockRadio`. Creamos el miembro privado en el constructor `ClockRadio` y delegamos el trabajo de los métodos `RadioOn()` y `RadioOff()` en este miembro privado. Cada vez que la implementación de la clase `Radio` cambia (por ejemplo, para reparar algún error), la clase `AlarmClock` incorporará automáticamente estos cambios. Un inconveniente del enfoque contención/delegación es que para añadir una nueva funcionalidad de la clase contenida (por ejemplo, añadir nuevos métodos para ajustar el volumen) es necesario hacer cambios en la clase contenida para delegar esta nueva funcionalidad en el miembro privado.

```

class ClockRadio : AlarmClock
{

```

```

private Radio radio;
// Declarar otras variables de miembro...

public ClockRadio()
{
    radio = new Radio();
    // Establecer el valor de otras variables de miembro...
}

//----- Deleagar en Radio -----
public void RadioOn()
{
    radio.On();
}

public void RadioOff()
{
    radio.Off();
}
}

```

Ya hemos implementado completamente la funcionalidad de la radio mediante el patrón contención/delegación. Es hora de añadir la funcionalidad `AlarmClock`. En primer lugar, se añade una variable privada `radioAlarm` que determina si debe sonar la radio o si debe sonar el timbre cuando se dispare la alarma:

```

class ClockRadio : AlarmClock
{
    private bool radioAlarm;
    // Declarar otras variables miembro...
    public ClockRadio()
    {
        radioAlarm = false;
        // Establecer el valor de otras variables miembro...
    }

    //----- Nueva funcionalidad ClockRadio-----
    public void SetRadioAlarm(bool useRadio)
    {
        radioAlarm = useRadio;
    }
}

```

Como queremos reemplazar la función `SoundAlarm()` de `AlarmClock`, necesitamos cambiar la declaración del método `SoundAlarm()` para que sea protegida. Además, como queremos que la función `Run()` tenga un comportamiento polimórfico, tendremos que hacer este método virtual:

```

class AlarmClock
{
    private int currentTime;
    private int alarmTime;
}

```

```

protected virtual void SoundAlarm()
{
    Console.WriteLine("Buzz!");
}

// Otros métodos...
}

```

Reemplazar `SoundAlarm()` en `AlarmClock` es sencillo. Dependiendo de los valores de `radioAlarm`, se enciende la radio o se llama al método `SoundAlarm()` de la clase base que hace sonar el timbre, como sigue:

```

ClockRadio : AlarmClock
{
    private Radio radio;
    private bool radioAlarm;

    //----- AlarmClock Reemplazado -----
    protected override void SoundAlarm()
    {
        if (radioAlarm)
        {
            RadioOn();
        }
        else
        {
            base.SoundAlarm();
        }
    }

    // Otros métodos...
}

```

¡Y en esto consiste básicamente! Algo muy interesante está ocurriendo dentro del método `Run()` de la clase `AlarmClock` (que aparece en el siguiente fragmento de código): el comportamiento polimórfico al que aludíamos. La clase `ClockRadio` hereda este método de su clase base y no lo reemplaza. Por tanto, este método `Run()` puede ser ejecutado desde un objeto de `AlarmClock` o un objeto de `RadioClock`. Como declaramos `SoundAlarm()` de modo que fuese virtual, C# es lo suficientemente inteligente como para llamar al `SoundAlarm()` apropiado dependiendo de qué clase está invocando al método `Run()`.

```

class AlarmClock
{
    private int currentTime;
    private int alarmTime;

    public void Run()
    {
        for (int currTime = 0; currTime < 43200; currTime++)

```

```

        {
            SetCurrentTime(currTime);
            if (GetCurrentTime() == GetAlarmTime())
            {
                Console.WriteLine("Current Time = {0}!", currentTime);
                SoundAlarm();
                break;
            }
        }
    }

    // Otros metodos...
}

```

Este ejemplo resalta uno de los puntos fuertes de la herencia: el polimorfismo. Además, cuando se añaden nuevos métodos públicos (o protegidos) a la clase base, están automáticamente disponibles en la clase derivada. El listado 11.7 es el listado completo con un método `main()` de ejemplo para que pueda experimentar con él.

Listado 11.7. La herencia múltiple puede ser simulada usando la contención

```

using System;

namespace Containment
{
    class Radio
    {
        protected bool on_off;

        public void On()
        {
            if (!on_off) Console.WriteLine("Radio is now on!");
            on_off = true;
        }

        public void Off()
        {
            if (on_off) Console.WriteLine("Radio is now off!");
            on_off = false;
        }
    }

    class AlarmClock
    {
        private int currentTime;
        private int alarmTime;

        protected virtual void SoundAlarm()
        {
            Console.WriteLine("Buzz!");
        }
    }
}

```



```

public void Run()
{
    for (int currTime = 0; currTime < 43200; currTime++)
    {
        SetCurrentTime(currTime);
        if (GetCurrentTime() == GetAlarmTime())
        {
            Console.WriteLine("Current Time = {0}!",
currentTime);
            SoundAlarm();
            break;
        }
    }
}

public int GetCurrentTime()
{
    return currentTime;
}

public void SetCurrentTime(int aTime)
{
    currentTime = aTime;
}

public int GetAlarmTime()
{
    return alarmTime;
}

public void SetAlarmTime(int aTime)
{
    alarmTime = aTime;
}
}

class ClockRadio : AlarmClock
{
    private Radio radio;
    private bool radioAlarm;

    public ClockRadio()
    {
        radio = new Radio();
        radioAlarm = false;
    }

    //----- Delegar en Radio -----
    public void RadioOn()
    {
        radio.On();
    }
}

```

```

public void RadioOff()
{
    radio.Off();
}

//----- AlarmClock Reemplazado -----
protected override void SoundAlarm()
{
    if (radioAlarm)
    {
        RadioOn();
    }
    else
    {
        base.SoundAlarm();
    }
}

//----- Nueva funcionalidad de ClockRadio -----
public void SetRadioAlarm(bool useRadio)
{
    radioAlarm = useRadio;
}

class ContInh
{
    static int Main(string[] args)
    {
        ClockRadio clockRadio;

        clockRadio = new ClockRadio();

        clockRadio.SetRadioAlarm(true);
        clockRadio.SetAlarmTime(100);
        clockRadio.Run();

        // esperar a que el usuario reconozca los resultados
        Console.WriteLine("Hit Enter to terminate...");
        Console.Read();
        return 0;
    }
}
}

```

La clase de objeto .NET

Todas las clases de C# derivan en última instancia de una clase construida en .NET Framework llamada `object`. Si se escribe una clase en C# y no se define

una clase base para ella, el compilador de C# la deriva de `object` sin ningún aviso. Supongamos que escribimos una declaración de clase de C# sin una declaración de clase, como en este ejemplo:

```
class Point2D
```

Esto es equivalente a derivar la clase de la clase base `.NET System.Object`:

```
class Point2D : System.Object
```

La palabra clave `object` puede usarse como si fuera un alias del identificador `System.Object`:

```
class Point2D : object
```

Si se deriva desde una clase base, hay que tener en cuenta que la clase base se deriva desde `object` o desde otra clase base que herede de `object`. Al final, la jerarquía de las clases base siempre incluye la clase `.NET object`.

Gracias a las reglas de herencia de C#, la funcionalidad de la clase `.NET object` está disponible para todas las clases de C#. La clase `.NET object` contiene los siguientes métodos:

- `public virtual bool Equals(object obj)`: Compara dos objetos y devuelve `true` si son iguales y `false` en caso contrario. Este método está marcado como `virtual`, lo que significa que se puede reemplazar en las clases de C#. Quizás quiera reemplazar este método para comparar el estado de dos objetos de la clase. Si los objetos tienen los mismos valores para los campos, puede devolver `true`; si los valores son diferentes puede devolver `false`.
- `public virtual int GetHashCode()`: Calcula un código hash para el objeto. Este método está marcado como `virtual`, lo que significa que se puede reemplazar en las clases de C#. Las colecciones de clase de `.NET` pueden llamar a este método para generar un código hash que ayude en las consultas y clasificaciones y las clases pueden reemplazar a este método para que genere un código hash que tenga sentido para la clase.

NOTA: El código hash es una clave única para el objeto especificado.

- `public Type GetType()`: Devuelve un objeto de una clase `.NET` llamado `Type` que proporciona información sobre la clase actual. Este método no está marcado como `virtual`, lo que significa que no se puede reemplazar en las clases de C#.
- `public virtual string ToString()`: Devuelve una representación de cadena del objeto. Este método está marcado como `virtual`, lo que

significa que se puede reemplazar en las clases de C#. Un método `ToString()` es invocado cuando algún método .NET como `System.Console.WriteLine()` necesita convertir una variable en una cadena. Se puede reemplazar este método para que devuelva una cadena más apropiada para representar el estado de una clase determinada. Quizás quiera, por ejemplo, añadir el signo adecuado a cada divisa junto a la representación de cadena de la clase `Money`.

- `protected virtual void Finalize()`: Puede ser llamado (o puede no serlo) cuando el recolector de objetos no utilizados del entorno de ejecución común destruye el objeto. Este método está marcado como virtual, lo que significa que se puede reemplazar en las clases de C#. También está marcado como protegido, lo que significa que sólo puede ser llamado desde dentro de la clase o desde una clase derivada y no puede ser llamado desde fuera de una jerarquía de clase. La implementación del objeto `Finalize()` de .NET no hace nada, pero puede implementarlo si quiere. También puede escribir un destructor para su clase, lo que produce el mismo efecto (pero tenga cuidado al usarlo). De hecho, el compilador de C# convierte el código destructor en un método reemplazado `Finalize()`.
- `protected object MemberwiseClone()`: Crea una copia idéntica del objeto, asigna al clon el mismo estado que el objeto original y devuelve el objeto copiado. Este método no está marcado como virtual, lo que significa que no se puede reemplazar en las clases de C#. También está marcado como protegido, lo que significa que sólo puede ser llamado desde dentro de la clase o desde una clase derivada y no puede ser llamado desde fuera de una jerarquía de clase.

Las estructuras de C# no pueden tener clases bases definidas explícitamente pero se derivan implícitamente de la clase base `object`. Todo el comportamiento de la clase `object` está disponible para las estructuras y las clases de C#.

Cómo usar boxing y unboxing para convertir a tipo `object` y desde el tipo `object`

Como todas las clases y estructuras derivan en última instancia del tipo `object` de .NET, éste suele usarse a menudo en listas de parámetros cuando el método necesita ser flexible respecto a los datos que recibe.

Observe, por ejemplo, el método `System.Console.WriteLine()` usado en este libro. Este mismo método ha sido usado para escribir cadenas, enteros y tipos dobles en la consola sin usar ningún operador de conversión explícita. En el listado 11.4, se escribe una cadena con comodines y los comodines son sustituidos por los valores de los parámetros que se le proporcionan. ¿Cómo funciona

en realidad? ¿Cómo sabe `System.Console.WriteLine()` qué tipos le va a pasar?

La respuesta es que no puede saberlo. Microsoft construyó el método `System.Console.WriteLine()` mucho antes de que trabajásemos con el listado 11.4, por lo que no podía saber qué tipos de datos le pasaría. Microsoft implementó un método `System.Console.WriteLine()` con la siguiente forma:

```
public static void WriteLine(string format, params object[] arg);
```

El primer parámetro es la cadena que se va a generar y el segundo parámetro es una matriz de parámetros que contiene una cantidad de elementos que se calcula cuando se compila el código. ¿Pero cuál es el tipo de la matriz de parámetros? La matriz de parámetros es del tipo `object`. Observe esta llamada a `WriteLine()`:

```
System.Console.WriteLine("{0}, {1}", X, Y);
```

El compilador de C# convierte los parámetros `X` e `Y` en una matriz de parámetros y llama a `WriteLine()`. Los parámetros `X` e `Y` son de tipo entero, lo que, como ya ha visto, es un alias de una estructura llamada `System.Int32`. Como las estructuras de C# heredan del tipo de `object` de .NET, estas variables heredan del tipo `object` y pueden ser usadas en una matriz de parámetros.

Los literales, que se han estudiado con anterioridad, son algo más complicados. En lugar de usar objetos, puede igualmente escribir el siguiente código:

```
System.Console.WriteLine("{0}, {1}", 100, 200);
```

Este código también funciona correctamente. ¿Cómo sabe C# cómo convertir un valor literal en un objeto para que pueda ser usado en una llamada de método que necesite un objeto? La respuesta está en una técnica llamada *boxing*.

La técnica de *boxing* permite que cualquier tipo de valor, incluso un literal, se pueda convertir en un objeto. Cuando el compilador de C# encuentra un tipo de valor para el que se necesita un tipo de referencia, crea una variable de objeto temporal y la asigna el valor del tipo de valor. Esta técnica "encierra" el valor en un objeto.

Observe nuevamente la anterior llamada `WriteLine()`:

```
System.Console.WriteLine("{0}, {1}", 100, 200);
```

El compilador de C# encuentra los literales y los encierra en objetos. Los objetos se envían a la llamada del método en la matriz de parámetros y a continuación se eliminan los objetos temporales. Observe las siguientes instrucciones:

```
int MyValue = 123;  
object MyObject = MyValue;
```

C# encierra el valor de `MyValue` en el objeto `MyObject`.

C# también permite la técnica de unboxing, que es simplemente el proceso opuesto al boxing. El unboxing reconvierte los tipos de referencia en tipos de valor. Por último, cada tipo es un objeto. El boxing y el unboxing nos ayudan a visualizar esta idea. Como todo es un objeto, todo (incluidos los literales) puede ser tratado como tal y los métodos de la clase `object` pueden llamarlos. El siguiente código funciona gracias a la técnica de boxing del compilador de C#:

```
string MyString;  
MyString = 123.ToString();
```

El compilador de C# aplica la operación boxing al valor literal 123, transformándola en un objeto y llama al método `ToString()` sobre ese objeto.

Resumen

En la terminología de la programación de software orientado a objetos, la herencia se usa para describir una clase que hereda miembros de una clase base. C# admite la herencia simple, en la que una clase puede derivarse de una sola clase base. C# no admite la herencia múltiple, que si es admitida por algunos lenguajes orientados a objetos para permitir que una clase pueda derivarse de más de una clase base.

Las clases base de C# se especifican al declarar una clase. El identificador de clase base sigue al nombre de la clase derivada cuando se declara la clase.

C# permite que los miembros de clase pertenezcan a un atributo de ámbito. El ámbito de un miembro determina su accesibilidad para las clases derivadas y los fragmentos de código que trabajan con los objetos de la clase. Los miembros de clase marcados como `public` son visibles para las clases derivadas y para el código que crea los objetos de la clase. Los miembros de clase marcados como `private` sólo son visibles para la clase en la que están definidos o desde clases derivadas de la clase. Los miembros de clase marcados como `internal` son visibles para todo el código en su mismo archivo binario, pero no son visibles fuera de los archivos binarios. Los miembros de clase marcados como `protected internal` son visibles para cualquier código en su mismo archivo binario y para las clases externas que se deriven de la clase. Los miembros de clase que no tienen ninguna palabra clave de ámbito son, por defecto, privados.

Los métodos y las propiedades de clases base pueden implementarse de nuevo en clases derivadas para proporcionar nuevas implementaciones. Los métodos y propiedades virtuales, que están marcados con la palabra clave de C# `virtual`, pueden implementarse nuevamente en clases derivadas, siempre que la nueva implementación mantenga el mismo identificador, tipo devuelto y lista de parámetros. Las nuevas implementaciones de métodos virtuales reciben el nombre de *reemplazadas* y deben estar marcadas con la palabra clave de C# `override`.

Los *métodos abstractos* son métodos de clases base que no pueden ser implementados. Los métodos abstractos no suelen implementarse en clases base porque la clase base no tiene suficiente información como para ofrecer una implementación completa. Las clases que contienen al menos un método abstracto reciben el nombre de clases abstractas y deben usar la palabra clave `abstract` en la declaración de la clase.

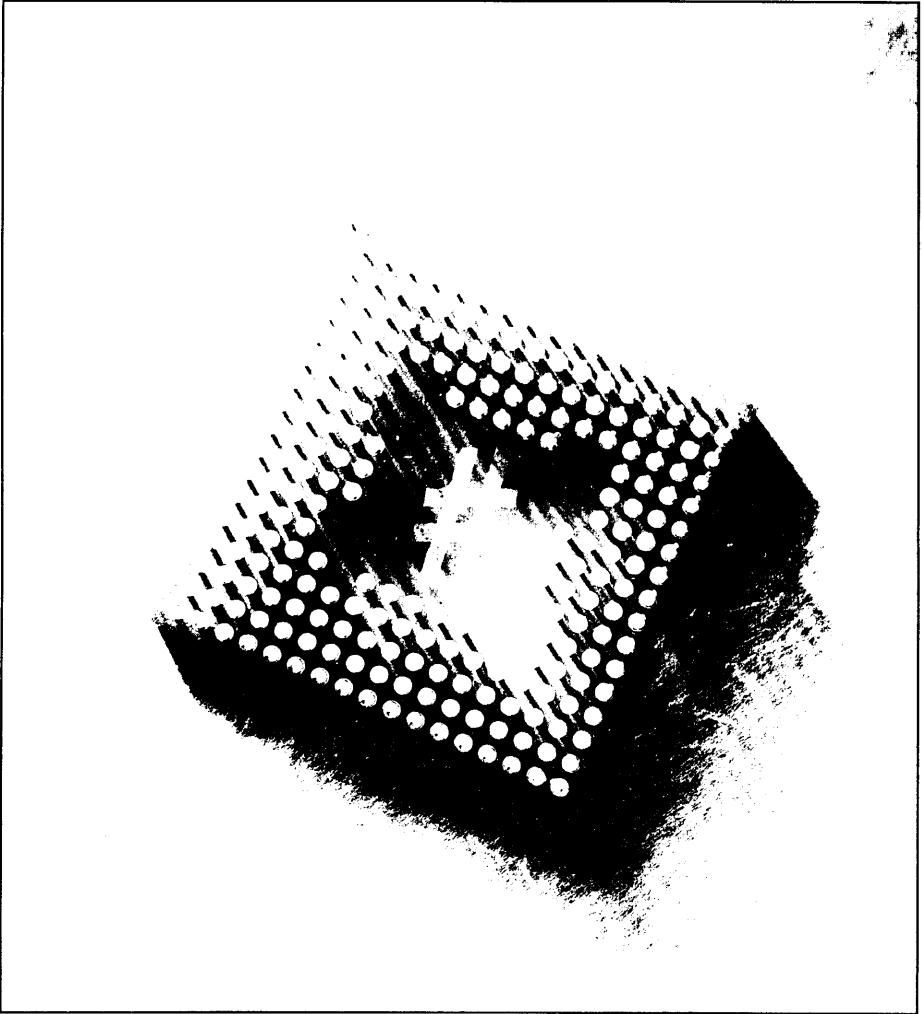
C# dispone de la palabra clave `base` que permite que las clases derivadas accedan a los miembros de una clase base. Se puede anteponer a los identificadores de miembros la palabra clave `base` y se puede usar esta palabra clave para llamar a un constructor de clase base desde un constructor de clase derivada.

Por defecto, se puede usar cualquier clase de C# como clase base y cualquier clase puede derivarse de cualquier otra clase. Puede evitar este comportamiento marcando una clase de C# con la palabra clave `sealed`. Las clases selladas no pueden usarse como clase base y el compilador de C# no permite que se deriven clases a partir de clases selladas.

Todas las clases de C# y, de hecho, cualquier clase implementada en un lenguaje .NET, deriva en última instancia de la clase `.NET System.Object`. La palabra clave de C# `object` es otro nombre para el identificador de clase `System.Object`. La clase `System.Object` contiene algunos métodos que pueden usar las clases derivadas y muchos de estos métodos pueden ser reemplazados. La clase `System.Object` proporciona funcionalidad para definir igualdad de objetos, cálculo de código hash, representaciones de cadenas, finalización de código y clonación de objetos. El tipo `object` puede ser usado como un método o variable de nombre de tipo y cualquier variable de C# puede usarse como un objeto `object`. C# convierte automáticamente algunos tipos de valor, como tipos de valor y literales numéricos y objetos de tipo `object` mediante las técnicas de `boxing` y `unboxing`. La técnica de `Boxing` encierra un valor en un objeto y la técnica de `unboxing` devuelve el valor del objeto a su tipo de valor original.

Parte III

C# avanzado



12 Cómo trabajar con espacios de nombre

Las clases que diseñe las usará en su código y probablemente en el código de otras personas. Las clases de C# pueden ser usadas por una aplicación VB.NET o desde dentro de una página ASP.NET. Además, las clases pueden ser usadas en combinación con otras clases diseñadas por otros programadores de .NET.

El código escrito en un lenguaje .NET hace referencia a las clases por sus nombres y todas estas clases usadas en combinación suscitan un dilema evidente: ¿Qué ocurre si un programador quiere usar dos clases al mismo tiempo?

Supongamos que escribe una clase de C# que lee archivos de una base de datos y llama a esa clase `Recordset`. El código que quiera usar la clase puede crear objetos como el siguiente:

```
Recordset MyRecordset = new Recordset();
```

Ahora supongamos que empaqueta las clases en un ensamblado .NET y distribuye ese ensamblado para que sea usado por otras aplicaciones. Además, suponga que alguien consigue su ensamblado y lo integra en su aplicación. ¿Qué ocurrirá si la misma aplicación también hace uso de otro ensamblado escrito por otra persona y que también contiene una clase llamada `Recordset`? Cuando el código de la aplicación crea un nuevo objeto `Recordset`, ¿qué clase se usa para crear el objeto? ¿La nuestra o la clase del otro ensamblado? Este problema puede resolverse mediante el concepto de C# de los *espacios de nombres*. Los espacios

de nombres organizan las clases mediante un grupo designado y el nombre del espacio de nombre puede ser usado para diferenciar dos clases con el mismo nombre. El código C# debe usar espacios de nombre para posteriormente ayudar a identificar nuestras clases mediante un grupo común, especialmente si está planeando construir un ensamblado para que sea usado por otros programadores. Los espacios de nombre pueden incluso ser útiles en las aplicaciones de C# que construyamos, porque de este modo pueden usar ensamblados externos que usen nombres de clase iguales a los nuestros.

Cómo declarar un espacio de nombre

Un espacio de nombre se declara con la palabra clave de C# `namespace` seguida por un identificador de espacio de nombre y llaves. Las clases que se incluirán en el espacio de nombre deben declararse dentro de las llaves del espacio de nombre, como se puede ver en el siguiente código:

```
namespace MyClasses
{
    class MyFirstClass
    {
    }
}
```

Este fragmento de código declara una clase llamada `MyFirstClass` en un espacio de nombre llamado `MyClasses`. Otro programador también podría escribir otra clase llamada `MyFirstClass`, pero mientras el otro programador use un espacio de nombre diferente, el compilador de C# encontrará la clase adecuada que debe usar para una instrucción particular.

Es posible declarar espacios de nombre dentro de otros espacios de nombre. Basta con encerrar la declaración del segundo espacio de nombre en el interior de la primera declaración:

```
namespace MyClasses
{
    namespace MyInnerNamespace
    {
        class MyFirstClass
        {
        }
    }
}
```

TRUCO: es aconsejable anidar los espacios de nombre cuando se va a ofrecer más de un producto distinto en forma de clase. Por ejemplo, la compañía `Widget Corporation` ofrece un producto de compresión y algunas

rutinas de emulación de terminales. Estos espacios de nombre serían `Widget.Compression` y `Widget.Emulation`, que agrupan los productos de la compañía pero también los mantiene separados mediante el espacio de nombre `Widget`.

Si no quiere anidar los espacios de nombre de esta manera, puede conseguir el mismo efecto declarando las dos declaraciones de espacio de nombre en la misma instrucción y separándolos con un punto, como se indica a continuación:

```
namespace MyClasses.MyInnerNamespace
{
    class MyFirstClass
    {
    }
}
```

Los siguientes tipos de declaraciones pueden aparecer en un espacio de nombre:

- Clases
- Estructuras
- Interfaces
- Enumeraciones
- Delegados

Cualquier declaración de un tipo que no esté en esta lista produce errores del compilador cuando se intenta compilar la aplicación.

Cómo declarar un espacio de nombre en varios archivos fuente

El compilador de C# permite el uso del mismo nombre de espacio de nombre en varios archivos fuente. A continuación crea un archivo binario que combina todas las clases en el mismo espacio de nombre.

Supongamos, por ejemplo, que quiere construir un ensamblado cuyas clases residan en un espacio de nombre llamado `MyAssembly`, que quiere escribir dos clases para incluirlas en ese ensamblado y que quiere definir las clases en archivos separados. Puede simplemente reutilizar el nombre del espacio de nombre en los dos archivos fuente. El primer archivo fuente puede contener la declaración de la primera clase, como en el siguiente ejemplo:

```
namespace MyAssembly
{
```

```

class MyFirstClass
{
}

```

El segundo archivo fuente puede contener la declaración de la segunda clase y puede usar el mismo nombre de espacio de nombre:

```

namespace MyAssembly
{
    class MySecondClass
    {
    }
}

```

Cuando los dos archivos fuente se construyen en un solo ensamblado, el compilador de C# crea un ensamblado con un solo espacio de nombre, `MyAssembly`, con dos clases en el espacio de nombre.

Esto tiene una ventaja para el programador en caso de que quiera separar algunas funcionalidades en distintos archivos o simplemente, si quiere reducir al mínimo la longitud de cada archivo fuente.

Cómo usar clases en un espacio de nombre

Si quiere hacer referencia a una clase en un espacio de nombre específico, anteponga al nombre de la clase el nombre de su espacio de nombre:

```

MyClasses.MyFirstClass MyObject = new MyClasses.MyFirstClass();

```

Esta sintaxis ayuda a distinguir entre las clases de diferentes códigos base con el mismo nombre. El compilador de C# ya tiene suficiente información para encontrar la clase correcta, porque también sabe a qué espacio de nombre debe dirigirse para encontrar las clases que estamos buscando.

Cuando se trabaja con clases declaradas en espacios de nombre anidados, deben aparecer todos los nombres de espacios de nombre cuando se hace referencia a esa clase:

```

Namespace1.Namespace2.MyClass MyObject = new
Namespace1.Namespace2.MyClass();

```

El listado 12.1 ilustra el concepto de espacio de nombre

Listado 12.1. Clases en espacios de nombre diferentes

```

namespace Namespace1
{
    class TestClass
    {

```

```

        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace1.TestClass!");
        }
    }
}

namespace Namespace2
{
    class TestClass
    {
        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace2.TestClass!");
        }
    }
}

class MainClass
{
    public static void Main()
    {
        Namespace1.TestClass Object1 = new Namespace1.TestClass();
        Namespace2.TestClass Object2 = new Namespace2.TestClass();
    }
}

```

El código del listado 12.1 declara dos clases llamadas `TestClass`. Cada una de las declaraciones de clase está en un espacio de nombre diferente y el constructor de cada clase escribe un mensaje en la consola. Los mensajes son ligeramente diferentes de modo que se puede saber cuál es el mensaje que emite cada clase.

El método `Main()` del listado 12.1 crea dos objetos: uno de tipo `Namespace1.TestClass` y otro de tipo `Namespace2.TestClass`. Como los constructores de las clases escriben mensajes en la consola, si se ejecuta el código del listado 12.1 obtendremos como resultado la figura 12.1.

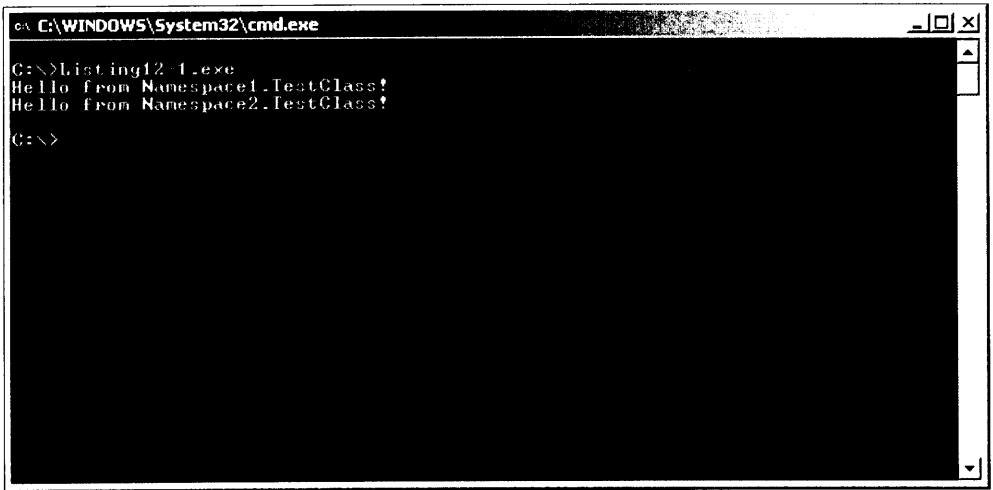
Observe que la clase `MainClass` del listado 12.1 no está encerrada en una declaración de espacio de nombre. Esto es perfectamente válido en C#. No es necesario encerrar las clases en declaraciones de espacio de nombre. Sin embargo, las clases que no están encerradas en espacios de nombre no pueden usar el mismo nombre en otra clase definida sin un espacio de nombre.

Si necesita usar una clase que está declarada en un espacio de nombre, debe usar el nombre de su espacio de nombre al usar el nombre de la clase. Si no hace esto, el compilador de C# emitirá un mensaje de error. Suponga, por ejemplo, que el método `Main()` del listado 12.1 intenta crear un objeto de la clase `TestClass`:

```

TestClass Object1 = new TestClass ();

```



```
e:\C:\WINDOWS\System32\cmd.exe
C:\>Listing12_1.exe
Hello from Namespace1.TestClass!
Hello from Namespace2.TestClass!
C:\>
```

Figura 12.1. Cómo hacer referencia a clases dentro de espacios de nombre

El compilador de C# no puede encontrar una clase llamada `TestClass` definida fuera de un espacio de nombre y emite el siguiente error:

```
error CS0234: El tipo o el nombre del espacio de nombres
'TestClass' no existe en la clase o el espacio de nombres
'MainClass' (¿falta una referencia de ensamblado?)
```

Si revisa los ejemplos de los capítulos anteriores, comprobará que ésta es la sintaxis que hemos estado usando en todas nuestras llamadas a `WriteLine()`, como muestra el siguiente ejemplo:

```
System.Console.WriteLine("Hello from C#!");
```

El método `WriteLine()` está en una clase llamada `Console` y la clase `Console` está definida en un espacio de nombre de .NET llamado `System`.

Cómo ayudar a los espacios de nombre mediante la palabra clave `using`

Hay varias maneras de usar la palabra clave de C# `using` para facilitar el trabajo con los espacios de nombre y ahorrar una buena cantidad de código. A primera vista, la palabra clave `using` parece la típica directiva `#include` de C/C++. No se deje engañar; sus ventajas son mucho más potentes. Las siguientes secciones describen algunas de estas ventajas.

Cómo crear alias de nombres de clase con la palabra clave `using`

Escribir nombres de clase perfectamente válidos y que incluyan nombres de espacio puede ser un poco tedioso, especialmente si los nombres son largos. Pue-

de usar la palabra clave `using` para crear un alias para el identificador de clase completo y, cuando se haya establecido el alias, puede usarlo en lugar del identificador de clase completo. Puede crear un alias mediante una instrucción que tenga la siguiente estructura:

- La palabra clave `using`
- El nombre del alias
- Un signo igual
- El nombre de la clase completa con el identificador de espacio de nombre
- Un punto y coma de fin de instrucción

El listado 12.2 se añade al listado 12.1 creando alias para los nombres de clase y acortando así sus equivalentes. El método `Main()` utiliza los nombres acortados para trabajar con los objetos de las clases.

Listado 12.2. Cómo crear alias de los nombres de clase

```
using Class1 = Namespace1.TestClass;
using Class2 = Namespace2.TestClass;

namespace Namespace1
{
    class TestClass
    {
        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace1.TestClass!");
        }
    }
}

namespace Namespace2
{
    class TestClass
    {
        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace2.TestClass!");
        }
    }
}

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



```

        Class1 Object1 = new Class1();
        Class2 Object2 = new Class2();
    }
}

```

El listado 12.2 escribe los mismos mensajes que el anterior ejemplo. Puede ver estos resultados en la figura 12.2.

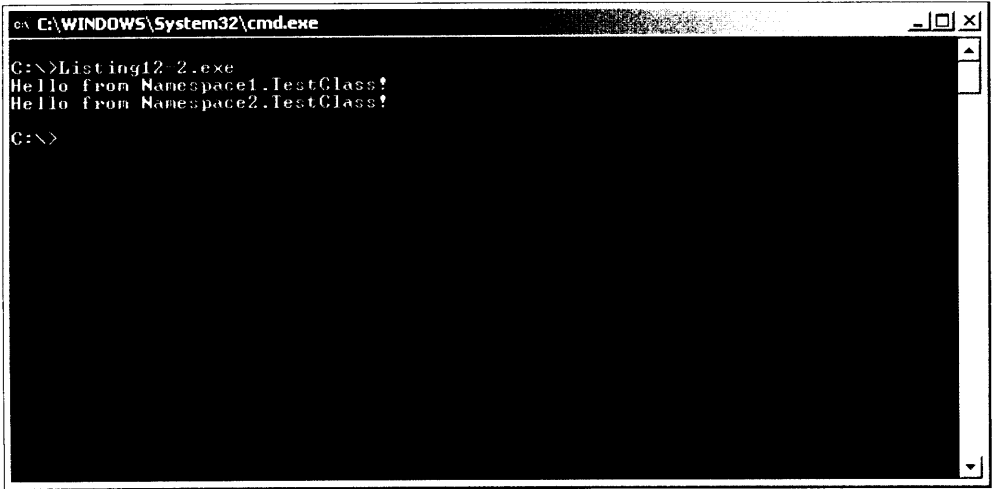


Figura 12.2. Cómo crear alias de los nombres de clase

Las instrucciones `using` deben incluirse en el código fuente antes de que se declaren los espacios de nombres. Si aparecen después de las declaraciones de espacio de nombre, recibirá el siguiente mensaje de error del compilador de C#:

```

error CS1529: Una clausula using debe ir delante de todos los
elementos restantes del espacio de nombres

```

En capítulos anteriores ya vimos que las palabras clave de C# que definen los tipos de variable son en realidad estructuras definidas por .NET Framework. Observe de nuevo la tabla 7.1 y preste atención a lo siguiente:

- Las estructuras de tipo de valor residen en el espacio de nombre de .NET System.
- La palabra clave `using` se usa para crear alias de los nombres de estructuras de .NET con las palabras clave de C# equivalentes. Puede imaginar cómo se implementa la tabla 7.1 en .NET Framework mediante instrucciones de C# como las siguientes:

```

using sbyte = System.SByte;
using byte = System.Byte;
using short = System.Int16;
// ... mas declaraciones ...

```

Puede crear alias de nombres de espacio de nombres así como de clases. como demuestra el listado 12.3.

Listado 12.3. Creación de alias de espacios de nombre

```
using N1 = Namespace1;
using N2 = Namespace2;

namespace Namespace1
{
    class TestClass
    {
        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace1.TestClass!");
        }
    }
}

namespace Namespace2
{
    class TestClass
    {
        public TestClass()
        {
            System.Console.WriteLine("Hello from
Namespace2.TestClass!");
        }
    }
}

class MainClass
{
    public static void Main()
    {
        N1.TestClass Object1 = new N1.TestClass();
        N2.TestClass Object2 = new N2.TestClass();
    }
}
```

Cómo declarar directivas de espacio de nombre con la palabra clave using

Si usa una clase declarada en un espacio de nombre, debe anteponer al nombre de clase el nombre del espacio de nombre, aunque no esté trabajando con ningún otro espacio de nombre que pueda tener una clase con el mismo nombre. Ésta es la razón por la que los ejemplos que hemos usado hasta este momento siempre han llamado a `WriteLine()` con el calificador del espacio de nombre `System`:

```
System.Console.WriteLine("Hello from C#!");
```

Por defecto, si no se usa el nombre de espacio de nombre, el compilador de C# emite un error:

```
error CS0234: El tipo o el nombre del espacio de nombres
'TestClass' no existe en la clase o el espacio de nombres
'MainClass' (¿falta una referencia de ensamblado?)
```

Anteponer cada nombre de clase con nombres de espacios de nombre como `System` es tedioso, especialmente si hay que hacerlo muchas veces. Afortunadamente, se puede usar la palabra clave `using` para reducir el tiempo de codificación.

Al usar la palabra clave `using` con un nombre de espacio de nombre se advierte al compilador de C# que se quiere hacer referencia a clases en el espacio de nombres designado sin anteponer a los nombres de clase el nombre de espacio de nombre. Observe, por ejemplo, la siguiente instrucción:

```
using System;
```

Esto recibe el nombre de *directiva de espacio de nombre*. Las directivas de espacio de nombre avisan al compilador de C# de que el código usará clases del espacio de nombres y que las clases no llevarán antepuesto el nombre del espacio de nombre. El compilador de C# se encarga de encontrar la definición de cada clase en cada espacio de nombre al que se hace referencia en una directiva de espacio de nombre.

El listado 12.4 es una modificación del listado 12.2; incluye una instrucción `using` que hace referencia al espacio de nombre de .NET `System`.

Listado 12.4. Cómo usar una directiva de espacio de nombre

```
using System;

using Class1 = Namespace1.TestClass;
using Class2 = Namespace2.TestClass;

namespace Namespace1
{
    class TestClass
    {
        public TestClass()
        {
            Console.WriteLine("Hello from Namespace1.TestClass!");
        }
    }
}

namespace Namespace2
{
    class TestClass
    {
        public TestClass()
```

```

        {
            Console.WriteLine("Hello from Namespace2.TestClass!");
        }
    }

class MainClass
{
    public static void Main()
    {
        Class1 Object1 = new Class1();
        Class2 Object2 = new Class2();
    }
}

```

La directiva de espacio de nombre `System` del listado 12.4 permite que el código haga referencia a la clase `Console` sin que se le anteponga el espacio de nombre `System`.

Un rápido recorrido por los espacios de nombre de .NET

.NET Framework tiene clases en multitud de espacios de nombre predefinidos que pueden usarse en otros códigos de C#. La siguiente lista describe algunos de ellos:

- El espacio de nombre `System` contiene clases que implementan funcionalidades básicas, como conversiones de tipos de datos, operaciones matemáticas, invocación a programas y gestión del entorno de procesos. El espacio de nombre `System` es el mayor de los proporcionados por .NET. .NET Framework también contiene el espacio de nombre `Microsoft` que brinda compatibilidad con versiones anteriores, además de otros elementos generalmente útiles.
- El espacio de nombre `System.CodeDOM` contiene clases que representan los elementos de un documento de código fuente.
- El espacio de nombre `System.Collections` contiene clases que implementan colecciones de objetos, como listas, colas, matrices, tablas hash y diccionarios.
- El espacio de nombre `System.ComponentModel` contiene clases que se usan para crear componentes y controles durante el tiempo de diseño y ejecución. Este espacio de nombre proporciona interfaces y clases para crear atributos, establecer enlaces a varias fuentes de datos, conceder licencias de componentes, además de para convertidores de tipos.

- El espacio de nombre `System.Data` contiene clases que componen la arquitectura de acceso a datos de ADO.NET. La arquitectura ADO.NET permite construir componentes que pueden gestionar datos de varias fuentes de datos en modo desconectado o conectado.
- El espacio de nombre `System.Diagnostics` contiene clases que ayudan a detectar errores en aplicaciones de .NET y supervisar la ejecución del código. El espacio de nombre `System.Diagnostics` también contiene clases que permiten supervisar la actuación de la aplicación mediante contadores de rendimiento y registros de eventos. Aunque la funcionalidad no se considera realmente un diagnóstico, este espacio de nombre también permite iniciar y detener procesos.
- El espacio de nombre `System.Drawing` contiene clases que implementan funcionalidad de dibujo del Dispositivo de interfaz gráfica (GDI). Este espacio de nombre no está disponible por defecto: hay que crear una referencia a él desde el menú **Proyecto**.
- El espacio de nombre `System.IO` contiene clases que pueden leer y escribir flujos de datos y archivos de disco. Las clases contenidas en este espacio de nombre pueden gestionar la entrada y salida de archivos sincrónica y asincrónica.
- El espacio de nombre `System.Messaging` contiene clases que trabajan con colas de mensajes. Este espacio de nombre no está disponible por defecto: hay que crear una referencia a él desde el menú **Proyecto**.
- El espacio de nombre `System.Net` contiene clases que proporcionan un contenedor de clase para los muchos protocolos que se utilizan actualmente en las redes. Este espacio de nombre consta de clases para gestionar peticiones de DNS, HTTP y peticiones de FTP. Además de las clases generales de acceso a redes, también hay muchas clases de seguridad de redes que tratan los diferentes aspectos de la seguridad, desde accesos a sitios Web hasta accesos de nivel de socket.
- El espacio de nombre `System.Reflection` contiene clases que proporcionan una vista de tipos, métodos y campos disponibles para una aplicación de .NET. Incluso es posible crear e invocar tipos dinámicamente en el tiempo de ejecución usando las clases del espacio de nombre `System.Reflection`.
- El espacio de nombre `System.Resources` proporciona clases que permiten a los programadores crear, almacenar y administrar recursos específicos de las referencias culturales que se utilizan en las aplicaciones.
- El espacio de nombre `System.Runtime` no es muy útil por sí mismo. Sin embargo, dispone de docenas de clases que proporcionan una enorme

funcionalidad. Por ejemplo, `System.Runtime.InteropServices` permite el acceso a objetos COM y a los API nativos desde .NET.

- El espacio de nombre `System.Security` contiene clases que permiten el acceso a la estructura subyacente de seguridad de .NET Framework. El espacio de nombre de seguridad es el punto de partida para otros espacios de nombre más avanzados de muchos servicios de cifrado. Estos servicios incluyen el cifrado y descifrado de datos, generación de hash y generación de números aleatorios.
- El espacio de nombre `System.Text` contiene clases que permiten trabajar con codificaciones de caracteres ASCII, Unicode, UTF-7 y UTF-8.
- El espacio de nombre `System.Threading` contiene clases que permiten implementar varios subprocesos del sistema operativo en las aplicaciones .NET, creando así una auténtica aplicación multiproceso.
- El espacio de nombre `System.Timers` contiene clases que permiten desencadenar un evento en un intervalo de tiempo determinado o en unos plazos más complejos. Estos temporizadores se basan en el servidor. Un temporizador basado en un servidor tiene la capacidad de moverse entre los subprocesos para iniciar el evento, lo que proporciona una flexibilidad mayor que el temporizador típico de Windows.
- El espacio de nombre `System.Web` contiene clases que implementan el protocolo de transmisión del hipertexto (HTTP) que utilizan los clientes Web para acceder a páginas de Internet. Este espacio de nombre no está disponible por defecto; hay que crear una referencia a él desde el menú Proyecto.
- El espacio de nombre `System.Windows.Forms` contiene clases para crear aplicaciones completas para Windows. Las clases del espacio de nombre `System.Windows.Forms` proporcionan un entorno de clase .NET con los controles típicos de Windows como cuadros de diálogo, menús y botones. Este espacio de nombre no está disponible por defecto; hay que crear una referencia a él desde el menú Proyecto.
- El espacio de nombre `System.Xml` contiene clases que pueden procesar datos XML. Este espacio de nombre incluye compatibilidad con espacios de nombre XML 1.0, XML, esquemas XML, XPath, XSL y XSLT, DOM Level 2, y SOAP 1.1.

Aunque no es una lista completa, debería darle una idea de la inmensa cantidad de espacios de nombre ya implementados por .NET Framework. Consulte la documentación del SDK de .NET Framework para conseguir una lista completa de espacios de nombre y clases.

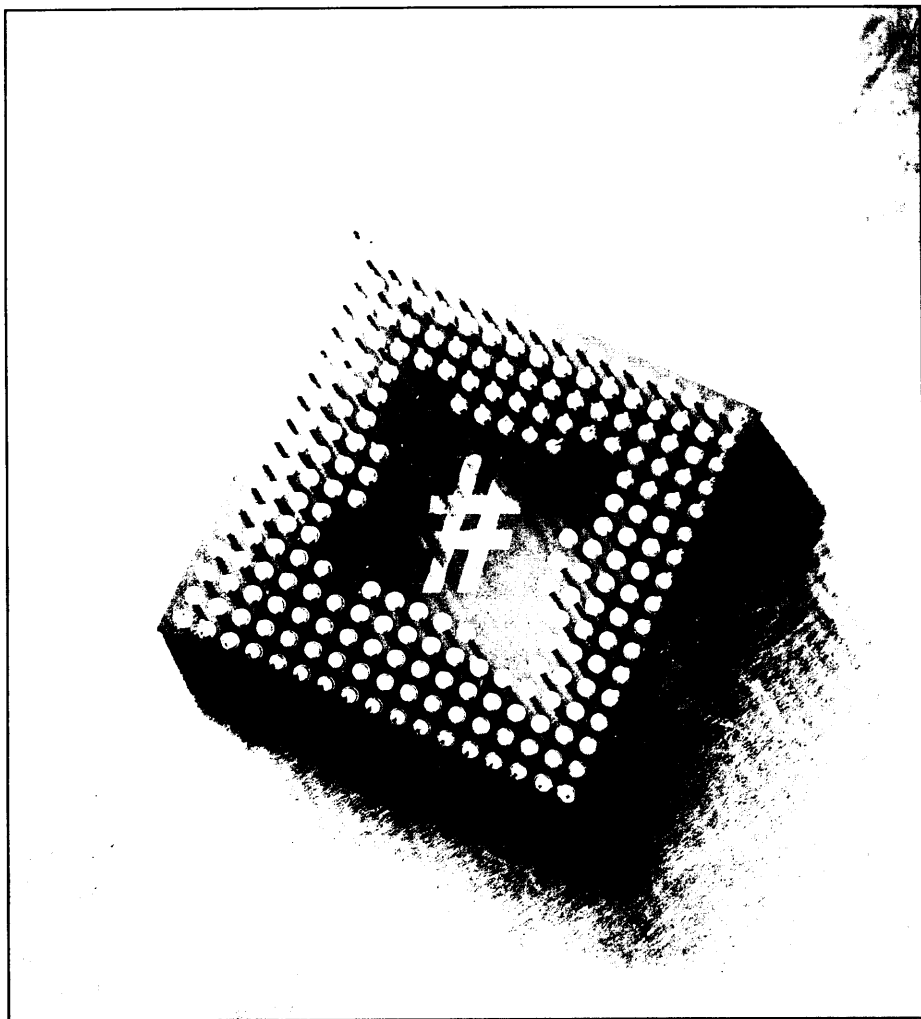
Resumen

Las clases y estructuras que desarrolle se pueden encapsular en los llamados *espacios de nombre*. Los espacios de nombre ayudan a diferenciar unas clases y estructuras de otras que tengan el mismo nombre.

Una clase o estructura completa incluye el nombre del espacio de nombre que alberga a la clase o estructura. Cuando se hace referencia a una clase o estructura en un espacio de nombre, hay que cualificar el nombre anteponiendo el nombre del espacio de nombre y un punto.

Se puede usar la palabra clave `using` para facilitar el trabajo con los nombres de espacios de nombre en el código de C#. La palabra clave `using` puede usarse para proporcionar un alias para una clase particular en un espacio de nombre concreto. También puede usarse como una directiva de espacio de nombre, que avisa al compilador de C# de que nuestro código va a hacer referencia a clases en un espacio de nombre específico y de que el código *no* antepondrá el identificador de espacio de nombre a las clases de ese espacio de nombre.

Puede crear sus propios espacios de nombre y puede usar el código incluido en espacios de nombre desarrollados por otras personas mediante las técnicas reseñadas en este capítulo. .NET Framework incluye una gran cantidad de espacios de nombres llenos de clases, lo que facilita la labor de codificar cualquier cosa, desde aplicaciones Windows hasta procesadores de XML y programas de seguridad. Los espacios de nombre de .NET Framework también proporcionan clases que pueden usarse para crear código C# mediante técnicas avanzadas, como el software de multiproceso y la reflexión.



13 Interfaces

Una interfaz de C# es un conjunto de firmas de métodos, propiedades, eventos o indizadores agrupados con un nombre común. Las interfaces funcionan como conjuntos de funcionalidades definidas que pueden implementarse en una clase o estructura de C#. Las clases o estructuras que implementa una interfaz proporcionan implementaciones para todos los métodos definidos en la interfaz.

Supongamos, por ejemplo, que queremos que las clases de nuestro proyecto puedan guardar los valores de sus campos en una base de datos y recuperarlos más tarde. Al implementar este requisito, podríamos decidir que todas las clases deban implementar un método llamado `Load()` y otro llamado `Save()`. También podríamos definir los métodos en una interfaz llamada `IPersistToDisk` (los nombres de una interfaz comienzan normalmente con la letra I, aunque no es obligatorio en C#) y exigir que nuestras clases implementen esta interfaz.

El código de C# puede consultar un objeto para determinar si admite una interfaz. Consultar a un objeto acerca de una interfaz es básicamente hacer la pregunta "¿Admites esta interfaz?" El objeto responde "Sí" o "No". Si el objeto responde "Sí", se puede llamar al método en la interfaz. Los métodos llamados en una interfaz siempre tienen la misma lista de parámetros y valores devueltos, aunque sus implementaciones pueden ser diferentes.

Imagine una interfaz como un contrato, una promesa de que una clase implementará un conjunto específico de funcionalidades. Si un objeto responde

"Sí, yo admito la interfaz por la que está preguntando", está asegurando que proporciona una implementación para cada uno de los métodos definidos en la interfaz. Las interfaces no proporcionan sus propias implementaciones de métodos. Sólo proporcionan identificadores de métodos, listas de parámetros y códigos devueltos. Las clases que implementan la interfaz son las responsables de proporcionar una implementación. Dos clases que implementan la misma interfaz pueden implementar los métodos de la interfaz de modos muy distintos. Esto es correcto, siempre que las clases sigan definiendo las firmas de los métodos en la definición de la interfaz.

Vamos a usar la interfaz `IPersistToDisk` como ejemplo. Puede tener objetos en su aplicación que necesiten abrir su estado desde un disco y guardar su estado de nuevo en un disco. Podría decidir implementar la interfaz `IPersistToDisk` en estos objetos. Para cada uno de los objetos que implementen `IPersistToDisk`, necesitará escribir código para los métodos `Load()` y `Save()` de la interfaz. Algunos de los objetos pueden tener necesidades de almacenamiento básicas, de modo que esos objetos pueden implementar los métodos `Save()` y `Load()` mediante un simple código de E/S de disco. Otros objetos pueden ser más complicados y necesitan compatibilidad con la E/S transaccional, en la que toda la operación de persistencia debe tener éxito o fracasar como un todo. Para esos objetos, quizás prefiera implementar los métodos `Load()` y `Save()` usando código transaccional, más robusto. La clave está en que el código que usan estos objetos no necesita saber si el objeto usa código simple o transaccional en su implementación. Sólo pregunta a cada objeto "¿Admites el método `IPersistToDisk`?" Para los objetos que responden "sí", el código que usa los objetos puede llamar a `Load()` o `Save()` sin necesidad de saber cómo están implementados realmente esos métodos.

Conceptualmente, las interfaces son muy parecidas a clases base abstractas: las dos proporcionan una lista de métodos que deben ser implementadas por otros fragmentos de código. Sin embargo, hay una diferencia importante: las interfaces pueden ser implementadas sin importar la posición de la clase de implementación en una jerarquía de clases. Si se usan clases base abstractas, todas las clases que quieran implementar la funcionalidad deben derivarse directa o indirectamente de la clase base abstracta. Esto no ocurre con las interfaces: las interfaces pueden implementarse en cualquier clase, sin importar su clase base. Las clases no necesitan derivarse de una clase base específica antes de poder implementar una interfaz.

El concepto de interfaz como modelo de diseño de software no es nueva: sin embargo, el modelo de objetos de componentes (COM) de Microsoft popularizó el concepto. COM trajo la idea de las interfaces (conjuntos específicos de funcionalidad implementados por un objeto) a la vanguardia del desarrollo de software basado en Windows. C# llevó más allá el concepto, promoviendo el concepto como una característica de lenguaje en el nivel de código. Aunque las primeras versiones de C++ o Visual Basic ya tenían interfaces COM integradas, estos lenguajes no eran compatibles con el concepto como rasgo de lenguaje. La

palabra clave de C# `interface` hace que el concepto de programación de interfaz sea compatible con el código fuente y esté disponible para el código, aunque el código no use COM.

Este capítulo le enseña a trabajar con interfaces usando C#. Aprenderá a definir una interfaz usando la palabra clave `interface`. También aprenderá a definir e implementar métodos, propiedades, indizadores y eventos en una interfaz y a acceder a una interfaz implementada por un objeto.

Cómo definir una interfaz

El primer paso al trabajar con interfaces en C# consiste en definir los métodos que componen la interfaz. Las interfaces se definen en C# con la siguiente sintaxis:

- La palabra clave `interface`
- Un identificador de interfaz
- Interfaces base opcionales
- Una llave de apertura
- Una o más declaraciones de miembro de interfaz
- Una llave de cierre

Las interfaces pueden definir métodos, propiedades, indizadores y eventos. Estos constructores de lenguaje trabajan sobre las interfaces del mismo modo que trabajarían con las clases de C#. Los métodos de interfaz definen bloques de código con nombre; las propiedades definen variables que pueden ser validadas mediante código descriptor de acceso y los eventos definen acciones que pueden ocurrir en el código.

Cómo definir métodos de interfaz

La agregación de un método a una interfaz significa que cualquier objeto que desee implementar la interfaz debe proporcionar una implementación del método de interfaz. Esto garantiza al código que los objetos que implementan la interfaz, incluyendo una implementación del método, pueden ser llamados por el código. Para definir un método en una interfaz, realice una declaración de método que proporcione el tipo devuelto por el método, el identificador y la lista de parámetros. La definición en C# de la interfaz `IPersistToDisk` que aparece al principio del capítulo sería:

```
interface IPersistToDisk
{
    bool Load(string FileName);
}
```

```
bool Save(string FileName);
}
```

La interfaz define dos métodos, pero no proporciona ninguna implementación para los métodos. Las declaraciones de método terminan con un punto y coma. Las clases de C# que implementa la interfaz `IPersistToDisk` prometen que proporcionarán una implementación de los métodos `Load()` y `Save()` tal y como se definen en la interfaz.

Cómo definir propiedades de interfaz

Las propiedades definen variables que pueden ser definidas por una interfaz. Al igual que las propiedades de clase, las propiedades de interfaz están asociadas con funciones de descriptores de acceso que definen el código que debe ejecutarse cuando se lea o escriba el valor de la propiedad. Para definir una propiedad en una interfaz hay que indicar el tipo de propiedad, el identificador y las palabras clave descriptoras de acceso seguidas de puntos y comas. Las palabras clave descriptoras de acceso aparecen entre llaves. La clase o estructura que implementa la interfaz es la responsable de proporcionar la implementación de los descriptores de acceso de la propiedad. Para definir una propiedad de lectura/escritura en una interfaz hay que usar las palabras clave descriptoras de acceso `get` y `set`:

```
interface Interfacel
{
    int RecordCount { get; set; }
}
```

Para definir una propiedad de sólo lectura en una interfaz, basta con incluir la palabra clave descriptora de acceso `get`:

```
interface Interfacel
{
    int RecordCount { get; }
}
```

Para definir una propiedad de sólo escritura en una interfaz, basta con incluir la palabra clave descriptora de acceso `set`:

```
interface Interfacel
{
    int RecordCount { set; }
}
```

Cómo definir indizadores de interfaz

Los indizadores son propiedades especiales que permiten al código acceder a datos como si estuvieran almacenados en una matriz. Los indizadores pueden definirse en una interfaz del mismo modo que se definen en una clase. Para definir

un indizador en una interfaz hay que indicar el tipo del indizador, la palabra clave `this`, la lista de parámetros del indizador entre corchetes y las palabras clave descriptoras de acceso seguidas de puntos y comas. Las palabras clave descriptoras de acceso aparecen entre llaves. La clase o estructura que implementa la interfaz es la responsable de proporcionar la implementación de los descriptores de acceso del indizador. Para definir un indizador de lectura/escritura en una interfaz, use las palabras clave descriptoras de acceso `get` y `set`:

```
interface Interfacel
{
    int this [int Index] { get; set; }
}
```

Para definir un indizador de sólo lectura en una interfaz, basta con incluir la palabra clave descriptora de acceso `get`:

```
interface Interfacel
{
    int this [int Index] { get; }
}
```

Para definir un indizador de de sólo escritura en una interfaz, basta con incluir la palabra clave descriptora de acceso `set`:

```
interface Interfacel
{
    int this [int Index] { set; }
}
```

Cómo definir eventos de interfaz

Los eventos pueden definirse en una interfaz del mismo modo que se definen en una clase. Quizás quiera añadir un evento a la interfaz `IPersistToDisk` mencionada al principio del capítulo para que, por ejemplo, se desencadene cuando las implementaciones de `Save()` y `Load()` empiecen a trabajar realmente con el disco para abrir o guardar los datos del objeto. Para definir un evento en una interfaz hay que usar la palabra clave `event`, el tipo del evento y un identificador de evento:

```
interface Interfacel
{
    event EventHandler ClickEvent;
}
```

Cómo derivar a partir de interfaces base

Las interfaces pueden derivarse de interfaces base, igual que las clases derivan de clases base. Las interfaces base se escriben tras los dos puntos que

siguen al nombre de la interfaz derivada. A diferencia de las clases base, las interfaces pueden derivarse de más de una interfaz base. Los diferentes nombres de interfaces base se separan por comas, como se puede apreciar en el siguiente ejemplo:

```
interface Interfacel
{
    void Method1();
}

interface Interface2
{
    void Method2();
}

interface Interface3 : Interfacel, Interface2
{
}
```

Es útil derivar de una interfaz base cuando una interfaz contiene un conjunto de firmas de método, propiedad y evento que deben ser añadidas a una interfaz ya programada. Por ejemplo, .NET Framework define varias interfaces que pueden implementarse y usarse en C#. Como las interfaces ya son una parte de .NET Framework, la lista de métodos, propiedades, indizadores y eventos que admiten se ha consolidado y no puede cambiar. Si quiere usar una interfaz definida y necesita añadir más firmas a la interfaz para uso propio, debería considerar la posibilidad de derivar desde la interfaz ya definida y añadir sus nuevas firmas en la interfaz derivada.

Las clases que implementan una interfaz derivada deben proporcionar implementaciones para todos los métodos definidos por las interfaces base. Cuando una clase implementa una interfaz, debe proporcionar el código para cada uno de los métodos definidos en la interfaz. Si una clase implementa `Interface3`, por usar el ejemplo anterior, la clase debe proporcionar las implementaciones de método para `Method1()` y `Method2()`. No se puede derivar una interfaz de ella misma. Observe la siguiente definición de una interfaz:

```
interface Interfacel : Interfacel
{
    void Method1();
}
```

Este error hace que el compilador de C# produzca el siguiente mensaje de error:

```
error CS0529: La interfaz heredada 'Interfacel' crea un ciclo en la jerarquia de la interfaz 'Interfacel'
```

Este mensaje de error indica que el código está intentando derivar una interfaz de sí misma, lo que no se permite en C#. Las interfaces sólo pueden derivarse de otras interfaces.

Cómo usar la palabra clave `new` para reutilizar identificadores

Se puede usar la palabra clave `new` para redefinir un identificador usado en una clase base. Suponga que está trabajando con una interfaz que define una propiedad llamada `ID`:

```
interface BaseInterface
{
    int ID { get; }
}
```

Ahora suponga que quiere derivar de esa interfaz, pero le gustaría usar el identificador `ID` como nombre de un método:

```
interface DerivedInterface : BaseInterface
{
    int ID();
}
```

Esta construcción hace que el compilador de `C#` emita un aviso sobre la reutilización del identificador `ID`:

```
warning CS0108: La palabra clave new es necesaria en
'DerivedInterface.ID()' porque oculta el miembro heredado
'BaseInterface.ID'
```

El compilador está avisando de que el identificador `ID` se está usando dos veces: una vez en la interfaz base y otra en la interfaz derivada como un nombre de método.

La reutilización de este nombre puede confundir fácilmente a los usuarios de la interfaz. El aviso del compilador significa que el uso de la palabra clave `ID` en la interfaz derivada tiene prioridad sobre el uso de la palabra clave `ID` en la interfaz base. Si un fragmento de código recibe una implementación de la interfaz `DerivedInterface`, el código será incapaz de llamar al método `ID()` pero no puede acceder a la propiedad `ID` de la interfaz base.

La reutilización del identificador `ID` en la interfaz derivada oculta el identificador `ID` en la clase base por lo que los clientes no pueden acceder a la propiedad de la interfaz base.

Para resolver este problema se puede usar la palabra clave `new` al reutilizar el identificador. El uso de la palabra clave `new`, mostrado en el siguiente ejemplo, indica al compilador de `C#` que se quiere dar un nuevo uso al símbolo reutilizado:

```
interface DerivedInterface : BaseInterface
{
    new int ID();
}
```


Cómo implementar interfaces en clases y estructuras

Tras definir una interfaz, se puede implementar esa interfaz en clases y estructuras. Esto indica a los usuarios de la clase o estructura que ésta proporciona implementaciones a los constructores definidos en la interfaz. Por ejemplo, si implementamos en una clase la interfaz `IPersistToDisk` que aparecía en la introducción, informaremos a los usuarios de la clase que ésta proporciona implementaciones de los métodos `Save()` y `Load()` y que se puede llamar a los métodos. Las interfaces que se están implementando se identifican de la misma forma que las clases base, con una lista de nombres tras dos puntos que siguen al identificador de clase o de estructura:

```
interface Interfacel
{
    void Method1();
}

class MyClass : Interfacel
{
    void Method1()
    {
    }
}
```

Este código define una interfaz llamada `Interfacel`. El método `Interfacel` declara un método: un método llamado `Method1()`. El código también declara una clase llamada `MyClass`, que implementa la interfaz `Interfacel`. La clase `MyClass` incluye una implementación para el método `Method1()` definido por la interfaz `Interfacel`.

Aunque la clase sólo puede derivarse de una clase base, puede implementar tantas interfaces como se deseen. Solamente hay que escribir las interfaces tras el identificador de clase y separar cada interfaz con dos puntos:

```
class MyClass : Interfacel, Interface2, Interface3
```

La implementación de interfaz múltiple se usa en todo .NET Framework. Por ejemplo, la clase `System.String` implementa cuatro interfaces definidas por .NET Framework:

- `IComparable`, que compara los valores de dos objetos del mismo tipo.
- `ICloneable`, que crea un nuevo objeto que tiene el mismo estado que otro objeto.
- `IConvertible`, que convierte el valor de un tipo a un valor de otro tipo.
- `IEnumerable`, que permite que el código itere a través de una colección.

Como la clase `System.String` implementa estas cuatro interfaces, la funcionalidad que cada una de las interfaces proporciona es compatible con la clase `System.String`. Esto significa que las cadenas pueden ser comparadas con otras cadenas, pueden ser clonadas, pueden convertirse en otros tipos y se puede iterar a través de sus caracteres como si fuera una colección. El concepto de implementación de interfaz múltiple también está disponible para cualquier programador de C#. C# también permite derivar una clase de una clase base e implementar interfaces al mismo tiempo:

```
class MyDerivedClass : CMyBaseClass, Interface1, Interface2
```

Las clases deben implementar cualquier declaración de evento, método, propiedad o indizador encontrado en una interfaz que implementen. En caso contrario, el compilador de C# emitirá un error. El siguiente código no funciona porque la clase `MyClass` implementa `Interface1` pero no proporciona una implementación del método `Method1()` definido en `Interface1`:

```
interface Interface1
{
    void Method1();
}

class MyClass : Interface1
{
    public static void Main()
    {
    }
}
```

El compilador de C# emite el siguiente mensaje de error al compilar el código:

```
error CS0535: 'MyClass' no implementa el miembro de interfaz
'Interface1.Method1()'
```

La clase debe proporcionar una implementación para la interfaz `Method1()` definida por `Interface1`, dado que la clase implementa `Interface1`. El siguiente ejemplo corrige el error:

```
interface Interface1
{
    void Method1();
}

class MyClass : Interface1
{
    public static void Main()
    {
    }

    public void Method1()
    {
    }
}
```

Cómo implementar métodos de interfaz con el mismo nombre

Debido a que es posible que el nombre de un método aparezca en más de una interfaz y como es posible que una clase de C# implemente más de una interfaz, puede ocurrir que se le pida a una clase de C# que proporcione múltiples implementaciones de diferentes interfaces que tengan el mismo nombre. Observe el método `DoWork()` en el siguiente código:

```
interface Interfacel
{
    void DoWork();
}

interface Interface2
{
    void DoWork();
}

class MyClass : Interfacel, Interface2
{
    void DoWork()
    {
    }
}
```

Este código no se puede compilar. El compilador de C# emite el siguiente mensaje que indica el error que produce:

```
error CS0536: 'MyClass' no implementa el miembro de interfaz
'Interfacel.DoWork()'. MyClass.DoWork()' es estático, no
publico, o tiene un tipo de valor devuelto incorrecto.
error CS0536: 'MyClass' no implementa el miembro de interfaz
'Interface2.DoWork()'. MyClass.DoWork()' es estático, no
publico, o tiene un tipo de valor devuelto incorrecto.
```

En los mensajes de error se muestra la sintaxis interfaz/nombre para recordar la sintaxis correcta de las implementaciones de clase.

El problema es que la clase `MyClass` necesita proporcionar código de implementación para el método `DoWork()` definido por `Interfacel` y del método `DoWork()` definido por `Interface2` y las dos interfaces reutilizan el nombre de método `DoWork()`. Una clase de C# no puede incluir dos métodos con el mismo nombre, de modo que ¿cómo se pueden definir los dos métodos de interfaz?

La solución es anteponer el nombre de la interfaz a la implementación del método y escribir un punto que separe el nombre de la interfaz del nombre de la implementación, como se muestra a continuación:

```

class MyClass : Interfacel, Interface2
{
    void Interfacel.DoWork()
    {
    }

    void Interface2.DoWork()
    {
    }
}

```

Esta clase se compila correctamente ya que contiene dos implementaciones `DoWork()`, una por cada interfaz definida. Como los nombres de método están calificados con los nombres de interfaz, el compilador de C# puede distinguir uno de otro y puede verificar que las dos interfaces han sido implementadas en la clase.

Cómo acceder a miembros de interfaz

Trabajar con clases que implementan interfaces es sencillo en C#. Normalmente se realizan estas operaciones cuando se trabaja con objetos cuyas clases implementan interfaces:

- Consultar un objeto para verificar que es compatible con una interfaz específica.
- Acceder a una interfaz en un objeto.
- Acceder a un miembro de la clase de un objeto definido inicialmente en una interfaz.

Las próximas secciones estudian estas operaciones.

Consultar a un objeto por una interfaz

Dado que diseña e implementa su propio código, ya sabe qué clases se usan en su aplicación y qué interfaces admiten. Sin embargo, cuando se escribe código que puede ser empleado por otras aplicaciones .NET y se reciben objetos de otros códigos, nunca se puede estar realmente seguro de qué interfaces admiten esos objetos. Por ejemplo, si está escribiendo un ensamblado y escribe código que acepta un tipo de objeto genérico, no puede saber si el objeto admite una interfaz dada.

Puede usar la palabra clave `is` para comprobar si un objeto admite o no una interfaz. La palabra clave `is` se emplea como una parte de una expresión booleana que se construye como se indica a continuación:

- Un identificador de objeto.
- La palabra clave `is`.
- Un identificador de interfaz.

La expresión devuelve `True` si el objeto admite la interfaz indicada y `False` en caso contrario. El listado 13.1 muestra el funcionamiento de la palabra clave `is`:

Listado 13.1. Cómo usar la palabra clave `is` para trabajar con una interfaz

```
using System;

public interface IPrintMessage
{
    void Print();
};

class Class1
{
    public void Print()
    {
        Console.WriteLine("Hello from Class1!");
    }
}

class Class2 : IPrintMessage
{
    public void Print()
    {
        Console.WriteLine("Hello from Class2!");
    }
}

class MainClass
{
    public static void Main()
    {
        PrintClassPrintObject = new PrintClass();

        PrintObject.PrintMessages();
    }
}

class PrintClass
{
    public void PrintMessages()
    {
        Class1 Object1 = new Class1();
        Class2 Object2 = new Class2();

        PrintMessageFromObject(Object1);
    }
}
```

```

        PrintMessageFromObject (Object2);
    }

    private void PrintMessageFromObject (object obj)
    {
        if (obj is IPrintMessage)
        {
            IPrintMessage PrintMessage;

            PrintMessage = (IPrintMessage)obj;
            PrintMessage.Print ();
        }
    }
}

```

El listado 13.1 define una interfaz llamada `IPrintMessage`. La interfaz `IPrintMessage` define un método llamado `Print`. Al igual que todos los miembros de interfaz de C#, la interfaz `IPrintMessage` define miembros pero no los implementa.

A continuación el listado implementa dos clases de control llamadas `Class1` y `Class2`. La clase `Class1` implementa un método llamado `Print()`. Como `Class1` no hereda de la interfaz `IPrintMessage`, el método `Print()` implementado por la clase no tiene ninguna relación con el método `Print()` definido por la interfaz `IPrintMessage`. La clase `Class2` implementa un método llamado `Print()`.

Como `Class2` hereda de la interfaz `IPrintMessage`, el compilador de C# considera que el método `Print()` implementado por la clase es una implementación del método `Print()` definido por la interfaz `IPrintMessage`.

A continuación el listado 13.1 define una clase llamada `MainClass`, que implementa el método `Main()` de la aplicación y otra clase llamada `PrintClass`. El método `Main()` del listado 13.1 crea un objeto de la clase `PrintClass` y llama a su método público para que haga el trabajo de verdad.

El listado 13.1 termina declarando una clase llamada `PrintClass`. La clase `PrintClass` implementa un método público llamado `PrintMessages()` y un método de ayuda privado llamado `PrintMessageFromObject()`. El método `PrintMessages()` es el método al que llama el método `Main()`. Como `PrintMessageFromObject()` está marcado como privado, sólo se le puede llamar desde otros fragmentos de código del objeto `PrintClass` y no puede ser llamado desde el código en otras clases. El método `PrintMessages()` crea un objeto de clase `Class1` y un objeto a partir de `Class2` y pasa cada objeto al método privado `PrintMessageFromObject()`. El método privado `PrintMessageFromObject()` acepta un parámetro de tipo `object` como parámetro.

NOTA: Es posible usar un parámetro de tipo `object` gracias a que todos los tipos de variable que el CLR admite derivan en última instancia de

System.Object y la palabra clave de C# object es un alias para el tipo System.Object. Cualquier tipo de variable que pueda ser representada en C# puede ser usada como parámetro para un método que espere un tipo de objeto porque todos los tipos son, en última instancia, objetos de System.Object.

En la siguiente línea del listado 13.1, el método `PrintMessageFromObject()` comienza examinando el objeto para comprobar si implementa la interfaz `IPrintMessage`:

```
if(obj is IPrintMessage)
```

Si el objeto implementa la interfaz, la expresión booleana `obj is IPrintMessage` devuelve `True` y el código situado por debajo de la condición `if` se ejecuta. Si el objeto no implementa la interfaz, la expresión booleana `obj is IPrintMessage` devuelve `False` y el código situado por debajo de la condición `if` no se ejecuta.

Si el objeto admite la interfaz, se puede acceder a la implementación del objeto de la interfaz. Se puede acceder a la implementación de la interfaz de un objeto declarando una variable del tipo de la interfaz y luego convirtiendo explícitamente el objeto al tipo de la interfaz, como se indica a continuación:

```
IPrintMessage PrintMessage;  
PrintMessage = (IPrintMessage)obj;
```

Tras inicializar una variable del tipo de la interfaz, se puede acceder a los miembros de la interfaz usando la habitual notación con punto:

```
PrintMessage.Print();
```

En el listado 13-2, se pasa `Object1` al método `PrintMessageFromObject()` y no se escribe nada en la consola porque el objeto `Object1` es de la clase `Class1` y `Class1` no implementa la interfaz `IPrintMessage`. Cuando se pasa `Object1` al método `PrintMessageFromObject()`, se escribe en la consola el siguiente texto:

```
Hello from Class2!
```

Este mensaje aparece porque el objeto `Object2` es de clase `Class2` y `Class2` implementa la interfaz `IPrintMessage`. Si se llama a la implementación del objeto del método `Print` de la interfaz se escribe el siguiente mensaje en la consola.

Cómo acceder a una interfaz en un objeto

Usar el operador `is` para trabajar con una interfaz requiere que el código acceda a un objeto dos veces:

- Una vez para consultar al objeto y comprobar si el objeto implementa una interfaz.
- Una vez para acceder a la implementación de la interfaz del objeto usando el operador de conversión explícita.

Se pueden combinar estos dos accesos mediante el operador `as`. El operador `as` realiza dos tareas en una sola instrucción. El listado 13.2 es una versión modificada del listado 13.1 que usa la instrucción `as` en lugar de la instrucción `is`:

Listado 13.2. Cómo usar la palabra clave `as` para trabajar con una interfaz

```
using System;

public interface IPrintMessage
{
    void Print();
};

class Class1
{
    public void Print()
    {
        Console.WriteLine("Hello from Class1!");
    }
}

class Class2 : IPrintMessage
{
    public void Print()
    {
        Console.WriteLine("Hello from Class2!");
    }
}

class MainClass
{
    public static void Main()
    {
        PrintClassPrintObject = new PrintClass();

        PrintObject.PrintMessages();
    }
}

class PrintClass
{
    public void PrintMessages()
    {
        Class1 Object1 = new Class1();
        Class2 Object2 = new Class2();
    }
}
```



```

        PrintMessageFromObject (Object1);
        PrintMessageFromObject (Object2);
    }

    private void PrintMessageFromObject (object obj)
    {
        IPrintMessage PrintMessage;

        PrintMessage = obj as IPrintMessage;
        if (PrintMessage != null)
            PrintMessage.Print ();
    }
}

```

El operador `as` se usa como parte de una expresión que se construye como se indica a continuación:

- Un identificador de objeto.
- La palabra clave `as`.
- Un identificador de interfaz.

Si el objeto designado en la expresión implementa la interfaz designada en la expresión, la implementación del objeto de la interfaz se devuelve como el resultado de la expresión. Si el objeto designado en la expresión no implementa la interfaz designada en la expresión, se asigna al resultado de la expresión un valor vacío representado por la palabra clave de C# `null`. La nueva implementación del método privado `PrintMessageFromObject ()` usa el operador `as`. Declara una variable local del tipo de la interfaz `IPrintMessage` y usa el operador `as` para acceder a la implementación del objeto del método.

Una vez que se ha completado la operación `as`, se comprueba la variable de implementación de la interfaz para descubrir si tiene el valor `null`. Si la variable no es `null`, se sabe que el objeto proporcionado implementa la interfaz y se puede llamar al método de la interfaz.

El listado 13.2 es funcionalmente equivalente al listado 13.1 y escribe el siguiente texto en la consola:

```
Hello from Class2!
```

Declaraciones de interfaz y palabras clave de ámbito

Al designar una interfaz, se puede marcar la interfaz como `public`, `protected`, `internal` o `private`. Si decide usar una de estas palabras claves para proporcionar un nivel de ámbito para la interfaz, debe colocarse inmediatamente antes de la palabra clave `interface`.

- Las interfaces marcadas como `public` son visibles para cualquier fragmento de código que tenga acceso al código en el que la definición de la interfaz pueda resolverse en tiempo de ejecución. Si se desarrolla un ensamblado y se implementa una interfaz pública en ese ensamblado, cualquier aplicación de .NET que acceda al ensamblado podrá trabajar con la interfaz.
- Las interfaces marcadas como `private` sólo son visibles para la clase en la que se definen. Sólo las interfaces cuyas definiciones están anidadas en clases pueden marcarse como `private`.
- Las interfaces marcadas como `protected` sólo son visibles para las clases en la que son definidas o desde clases derivadas de la clase. Sólo las interfaces cuyas definiciones están anidadas en clases pueden marcarse como `protected`.
- Las interfaces marcadas como `internal` son visibles para cualquier código en el mismo archivo binario, pero no son visibles para el código que se encuentre en otros archivos binarios. Si se define una interfaz en C# y se compila la clase formando un ensamblado, cualquier fragmento de código del ensamblado puede acceder a las interfaces internas. No obstante, si otro fragmento de código usa ese ensamblado, no tendrá acceso a la interfaz.

C# permite especificar una interfaz sin especificar ninguna palabra clave de ámbito. Si se declara una interfaz sin especificar ninguna palabra clave de ámbito, por defecto se le concede a la interfaz accesibilidad pública.

Cómo implementar interfaces definidas por .NET Framework

.NET Framework define varias interfaces que se pueden implementar en otras clases. En este capítulo ya se ha mencionado que .NET Framework define interfaces, como las interfaces `ICloneable`, `IEnumerable`, `IComparable` e `IConvertible` que implementa la clase `System.String`. Implementar interfaces definidas por .NET Framework puede ayudar a las clases a integrarse en .NET Framework y en el entorno común de ejecución (el CLR, para abreviar). Observe este ejemplo.

Cómo implementar foreach mediante IEnumerable IEnumerator

El listado 9.3 del capítulo 9, implementa una clase llamada `Rainbow`, que incluye un indizador que permite utilizar los contenidos de la clase (cadenas que

nombran los colores del arco iris) como elementos de una matriz, como se muestra en el siguiente ejemplo:

```
Rainbow MyRainbow = new Rainbow();

for(ColorIndex = 0; ColorIndex < MyRainbow.Count; ColorIndex++)
{
    string ColorName;

    ColorName = MyRainbow[ColorIndex];
    System.Console.WriteLine(ColorName);
}
```

Se puede reducir aún más este código usando la palabra clave `foreach` con la clase, como se muestra en el siguiente fragmento de código:

```
Rainbow MyRainbow = new Rainbow();

foreach(string Color in MyRainbow)
    Console.WriteLine(ColorName);
```

Por defecto, las clases no admiten la palabra clave `foreach` y usarla para acceder a los elementos de la clase hace que el compilador de C# genere el siguiente mensaje de error:

```
error CS1579: La instrucción foreach no funciona en variables
del tipo 'Rainbow' porque 'GetEnumerator' no contiene una
definición para 'miembro' o es inaccesible
```

Sin embargo, se puede usar `foreach` en las clases si la clase implementa una interfaz de .NET Framework llamada `IEnumerable`. La interfaz `IEnumerable` contiene métodos que .NET Framework usa para extraer elementos de los objetos. Si la clase contiene una colección de elementos y se quiere que otras partes del código usen la palabra clave `foreach` para iterar cada uno de los elementos de la colección, hay que implementar la interfaz `IEnumerable` en la clase.

La interfaz `IEnumerable` contiene una sola definición de método:

```
IEnumerator GetEnumerator();
```

El método `GetEnumerator()` debe implementarse en la clase y debe devolver un objeto que implementa otra interfaz .NET Framework llamada `IEnumerator`. La interfaz `IEnumerator` es la responsable de implementar el código que devuelven los elementos de clase individuales.

La interfaz `IEnumerator` define una propiedad y dos métodos como sigue:

- `object Current {get;}`
- `bool MoveNext();`
- `void Reset();`

La propiedad `Current` devuelve una referencia al elemento actual de la colección. El método `MoveNext()` se mueve al siguiente elemento de la colección y devuelve `True` si hay otro elemento a continuación o `False` si se ha llegado al final de la colección y no hay otro elemento a continuación. El método `Reset()` devuelve el integrador al principio de la colección.

Cuando se accede a los datos de una clase con el constructor `foreach`, .NET Framework accede a las interfaces de las clases `IEnumerable` e `IEnumerator` con código como el del siguiente pseudo-código:

```
IEnumerator IEnumerableImplementation;
IEnumerator IEnumeratorImplementation;

IEnumerableImplementation = YourClass as IEnumerable;
if (IEnumerableImplementation != null)
{
    IEnumeratorImplementation =
    IEnumerableImplementation.GetEnumerator();
    If (IEnumeratorImplementation != null)
    {
        while (IEnumeratorImplementation.MoveNext() == true)
            CurrentValue = IEnumeratorImplementation.Current;
    }
}
```

Las interfaces `IEnumerable` e `IEnumerator` están definidas en un espacio de nombre de .NET Framework llamado `System.Collections` y hay que hacer referencia a ese espacio de nombre cuando se trabaja con estas interfaces. Se puede hacer referencia a los nombres de espacio explícitamente:

```
class MyClass :
    System.Collections.IEnumerable,
    System.Collections.IEnumerator
```

Si se quiere, se puede usar en su lugar la palabra clave `using` para hacer referencia a los espacios de nombre:

```
using System.Collections;

class MyClass :
    IEnumerable,
    IEnumerator
```

El listado 13.3 remodela el listado 9.3 y usa la clase `Rainbow` para implementar las interfaces `IEnumerable` e `IEnumerator`; también usa el constructor `foreach` del método `Main()` para recorrer los elementos de la clase.

Listado 13.3. Cómo admitir `foreach` mediante `IEnumerable` `IEnumerator`

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

```

class Rainbow : IEnumerable, IEnumerator
{
    private short IteratorIndex = -1;

    public IEnumerator GetEnumerator()
    {
        return this;
    }

    public object Current
    {
        get
        {
            switch(IteratorIndex)
            {
                case 0:
                    return "Red";
                case 1:
                    return "Orange";
                case 2:
                    return "Yellow";
                case 3:
                    return "Green";
                case 4:
                    return "Blue";
                case 5:
                    return "Indigo";
                case 6:
                    return "Violet";
                default:
                    return "*** ERROR ***";
            }
        }
    }

    public bool MoveNext()
    {
        IteratorIndex++;
        if(IteratorIndex == 7)
            return false;
        return true;
    }

    public void Reset()
    {
        IteratorIndex = -1;
    }

    public static void Main()
    {
        Rainbow MyRainbow = new Rainbow();

        foreach(string ColorName in MyRainbow)

```

```

        Console.WriteLine(ColorName);
    }
}

```

Si se ejecuta el código del listado 13.3 se escribe el siguiente texto en la consola:

```

Red
Orange
Yellow
Green
Blue
Indigo
Violet

```

La clase `Rainbow` implementa las interfaces `IEnumerable` e `IEnumerator`. La clase mantiene un campo privado llamado `IteratorIndex` que se usa para seguir el rastro del siguiente elemento que debe devolverse en el bucle `foreach`. Se inicializa a `-1`; veremos por qué cuando examinemos la implementación de `MoveNext()` en las siguientes páginas.

La implementación de clase de `IEnumerable.GetEnumerator()` devuelve una referencia al objeto que se llama con la siguiente instrucción:

```
return this;
```

Recuerde que el método debe devolver una referencia a una clase que implemente el interfaz `IEnumerator`. Como el objeto usado para llamar a `IEnumerable.GetEnumerator()` también implementa la clase `IEnumerator`, se puede devolver el objeto al que se está llamando. Se puede usar la palabra clave `this` como valor devuelto. En este contexto, la palabra clave `this` hace referencia al objeto cuyo código se esté ejecutando en ese momento.

Como el compilador de C# puede determinar en tiempo real que el objeto que se ejecuta en ese momento implementa `IEnumerable`, el código no necesita convertir explícitamente la palabra clave `this` a una variable de tipo `IEnumerator`. El código resultante podría ser como el siguiente:

```
return this as IEnumerator;
```

Sin embargo, esto es redundante porque el compilador de C# ya puede comprobar que el objeto `this` (el objeto que se está ejecutando en ese momento) implementa la interfaz `IEnumerator`. Si se usa este código para devolver una referencia `IEnumerator`, recibirá el siguiente aviso del compilador de C#:

```
warning CS0183: La expresión dada es siempre del tipo
proporcionado ('System.Collections.IEnumerator')
```

El resto de la clase `Rainbow` implementa miembros de la interfaz `IEnumerator`. El primer miembro proporciona la implementación para la propiedad `IEnumerator.Current`. Examina el valor de la propiedad privada de

la clase `IteratorIndex` y devuelve una cadena que representa el color del arco iris en el índice al que hace referencia el valor de la propiedad `IteratorIndex`. La propiedad `Current` devuelve una variable de tipo `object`, pero como las cadenas son objetos igual que todos los otros tipos de datos, el CLR acepta los valores devueltos basados en cadenas.

La implementación del método `IEnumerator.MoveNext()` incrementa el valor de la propiedad privada `IteratorIndex`. Como el arco iris tiene siete colores, la implementación `MoveNext()` da por sentado que los valores válidos de `IteratorIndex` van de 0 a 6. Si el valor llega a 7, la implementación de `MoveNext()` asume que el iterador ha llegado a su límite y devolverá `False`. En caso contrario, la implementación devuelve `True`. La instrucción que incrementa el valor de `IteratorIndex` exige que el valor inicial de `IteratorIndex` sea -1. Cuando se llama a `MoveNext()` por vez primera, la instrucción de incremento aumenta el valor de `IteratorIndex` de -1 a 0, dándole a `IteratorIndex` un valor válido en la primera iteración del bucle.

La implementación del método `IEnumerator.Reset()` simplemente restablece el valor de `IteratorIndex` a -1. A este método se le llama si se llama a más de un constructor `foreach` y .NET Framework necesita devolver el estado de la enumeración a su valor inicial.

Toda esta implementación hace que el método `Main()` sea muy claro. El método puede crear un objeto de la clase `Rainbow` y usar `foreach` para iterar cada nombre de color de la clase.

Cómo implementar limpieza mediante `IDisposable`

El CLR contiene un mecanismo para la eliminación automática de objetos llamada recolección de objetos no utilizados. Es importante comprender cómo funciona este mecanismo, en qué se diferencia de otros sistemas y cómo hacer el código C# lo más compatible posible con este algoritmo para eliminar objetos creados.

En C++, un objeto se crea con la palabra clave `new` y la operación devuelve al objeto un puntero ya que se crea en el montón de memoria de la aplicación. El programador en C++ debe liberar esta memoria invocando al operador `delete` sobre ese mismo puntero cuando el objeto ya no sea necesario. Al invocar al operador `delete` se libera la memoria usada por el objeto y se llama al destructor de la clase para que la clase pueda realizar cualquier operación de limpieza específica de clase. Si no se invoca al operador `delete` sobre un puntero de objeto devuelto por `new`, se produce pérdida de memoria.

En algunos entornos de ejecución, como Visual Basic y COM, se lleva la cuenta de las referencias de los objetos. Los entornos de ejecución llevan la cuenta de los subprocesos asociados a un objeto y libera automáticamente el objeto cuando su contador de referencias llega a cero. Esto permite al programador

olvidarse de tener que llamar a una instrucción de destrucción como `delete` y ayuda a eliminar toda un tipo de errores relacionados con la pérdida de memoria.

CLR usa un esquema de recuperación de memoria llamado recolección de objetos no utilizados. Los objetos no se destruyen cuando se libera su última referencia, como ocurre con los sistemas que cuentan las referencias, como COM y COM+. En su lugar, los objetos se destruyen algo más tarde, cuando el recolector de objetos no utilizados de CLR se ejecuta y destruye los objetos preparados para ser borrados. Los destructores de objetos de C# se ejecutan, no cuando se libera la última referencia del objeto, sino cuando el recolector de objetos no utilizados libera las estructuras de datos internas de CLR usadas para seguir la pista al objeto. Es importante tener en cuenta este diseño de recolección de objetos no utilizados al programar las clases de C#. Las clases que gestionan recursos y necesitan ser explícitamente cerradas cuando se destruye el objeto, como los controladores de conexiones de bases de datos, deben cerrarse tan pronto como el objeto deje de usarse. Insertar código de limpieza en el destructor de la clase significa que los recursos no serán liberados hasta que el recolector de objetos no utilizados destruya el objeto, que puede ser mucho después de que se libere la última referencia al objeto.

.NET Framework admite una interfaz llamada `IDisposable` que las clases pueden implementar para admitir recursos de limpieza de clases. La interfaz `IDisposable` se incluye en el espacio de nombre `System` de .NET Framework. Admite un solo método llamado `Dispose()`, que no toma parámetros y no devuelve nada, como se muestra en el siguiente ejemplo:

```
using System;

public class MyClass : IDisposable
{
    public MyClass()
    {
    }

    ~MyClass()
    {
    }

    public void Dispose()
    {
    }
}
```

Esta clase admite un constructor, al que se llama cuando se crean los objetos de la clase; un destructor, al que se llama cuando el recolector de objetos no utilizados destruye los objetos de la clase; y `Dispose()`, al que se puede llamar cuando el código cliente se deshace del objeto.

El código cliente puede consultar los objetos para ver si son compatibles con la interfaz `IDisposable` y puede llamar a su método `Dispose()` para libe-

rar recursos de clase antes de que el recolector de objetos no utilizados destruya el objeto. El lenguaje C# realiza esta consulta de forma sencilla mediante una sintaxis especial que incluye la palabra clave `using`. La palabra clave `using` puede usarse en una expresión con paréntesis que incluye la creación de un nuevo objeto:

```
using(MyClass MyObject = new MyClass())
{
    // use aqui "MyObject"
}
```

En este ejemplo, la palabra clave `using` se usa para crear un nuevo objeto llamado `MyObject`. El nuevo objeto pertenece a la clase `MyObject`. El objeto puede usarse en cualquier instrucción que esté incluida entre las llaves que siguen a la palabra clave `using`. El objeto es automáticamente destruido cuando la ruta de acceso del código llega a la llave de cierre del bloque `using`. Si la clase del objeto creado en la instrucción `using` admite `IDisposable`, entonces el método `Dispose()` de la clase es invocado automáticamente sin que el cliente deba hacer nada. El listado 13.4 muestra una clase llamada `MyClass` que implementa la interfaz `IDisposable`.

Listado 13.4. `IDisposable` y la palabra clave `using`

```
using System;

public class MyClass : IDisposable
{
    public MyClass()
    {
        Console.WriteLine("constructor");
    }

    ~MyClass()
    {
        Console.WriteLine("destructor");
    }

    public void Dispose()
    {
        Console.WriteLine("implementation of
IDisposable.Dispose()");
    }
}

public class MainClass
{
    static void Main()
    {
        using(MyClass MyObject = new MyClass())
        {
```

```
}  
}  
}
```

Esta aplicación de consola implementa la clase `MyClass` mostrada en el listado 13.4 y contiene instrucciones en su constructor, destructor e implementación `Dispose()` que escriben mensajes en la consola. El listado 13.4 también incluye una instrucción `using` en su método `Main()` que crea un objeto de tipo `MyClass`. Si se ejecuta el listado 13.4 se escribe el siguiente mensaje en la consola:

```
constructor  
implementation of IDisposable.Dispose()  
destructor
```

Observe que la implementación `Dispose()` de la interfaz `IDisposable` es invocada automáticamente sin que el método `Main()` intervenga.

Tenga en cuenta que sólo debe implementar la interfaz `IDisposable` para las clases que tienen recursos que deben ser liberados explícitamente, como conexiones de bases de datos o indicadores de ventana. Si la clase sólo contiene referencias a objetos gestionados por el CLR, entonces no es necesario implementar `IDisposable`. Implementar `IDisposable` significa que el CLR necesita realizar más trabajo para eliminar los objetos y este trabajo adicional puede ralentizar el proceso de recolección de elementos no utilizados. Implemente `IDisposable` cuando sea necesario pero no lo haga a menos que sea necesario.

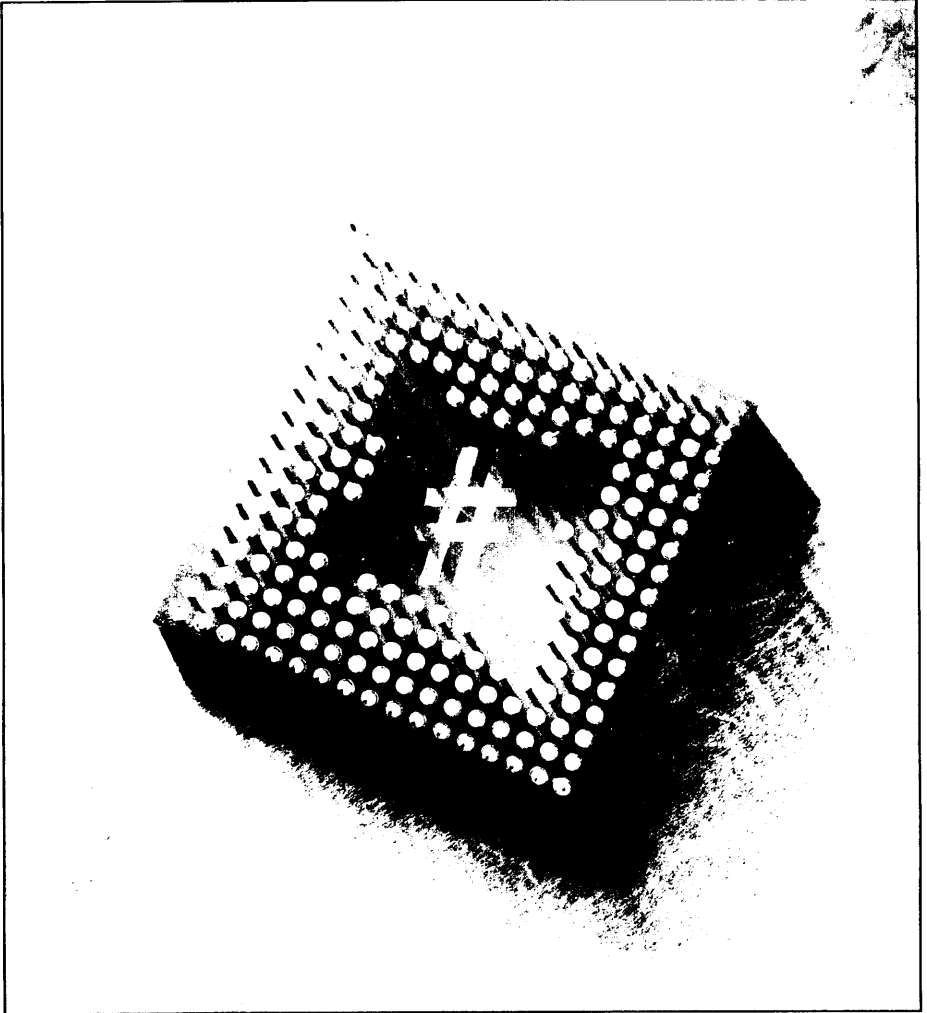
Resumen

Piense en una interfaz como en una promesa de que una clase implementará los métodos, propiedades, indizadores y eventos definidos en la interfaz. Las interfaces proporcionan definiciones de miembros, pero no proporcionan ninguna implementación. Se necesita una clase que implemente una interfaz para proporcionar una implementación de cada uno de los miembros de la interfaz. Una clase puede implementar varias interfaces, aunque sólo puede derivarse de una de las clases base. Las interfaces pueden derivarse de otras interfaces, del mismo modo que las clases pueden derivarse de las clases base.

Las palabras clave de C# `is` y `as` pueden usarse para trabajar con objetos que implementen interfaces. La palabra clave `is` se usa en una expresión booleana que devuelve `True` si un objeto implementa una interfaz y `False` en caso contrario. La palabra clave `as` convierte una variable de objeto en una variable de un tipo de interfaz. Las expresiones que usan la palabra clave `as` devuelven `null` si el objeto no implementa la interfaz designada.

En este capítulo se ve un ejemplo sobre cómo implementar una interfaz definida por .NET Framework. .NET Framework implementa muchas interfaces y es

aconsejable revisar la documentación y estudiarlas todas. Las interfaces de .NET Framework comienzan con la letra *I*. Reviselas todas. Puede usar las interfaces de .NET Framework para implementar cualquier cosa, desde dar formato personalizado a la consola para mecanizarla hasta la semántica de eliminación de la recolección de elementos no utilizados.



Enumeraciones

Algunas de las variables definidas en el código pueden usarse para contener un valor tomado de un conjunto de valores posibles. Por ejemplo, puede necesitar seguir el rastro del estado de un archivo. Podría definir una variable que pudiera describir si un archivo está abierto, cerrado o si no se puede encontrar. Un modo de lograrlo sería escoger algunas constantes para definir las distintas opciones y un entero para que contenga el valor actual, como en el siguiente código:

```
const int FileOpen = 1;
const int FileClosed = 2;
const int FileNotFound = 3;

int FileStatus;

FileStatus = FileClosed;
```

Este código es código C# válido y se compilará perfectamente. Sin embargo, un programador puede asignar a la variable un valor que no esté disponible en el conjunto de constantes definidas. El tipo de datos de `FileStatus` es un número entero y el compilador de C# acepta perfectamente cualquier código que asigne a la variable cualquier valor entero válido, aunque el objetivo es restringir el conjunto de valores válidos al conjunto definido por las constantes. En teoría, asignar a `FileStatus` un valor no definido por las constantes no debería estar

permitido porque lo que se pretendía en un principio era restringir el conjunto de valores posibles al conjunto definido para esa variable.

La situación de desarrollo ideal en casos como éste es que debería ser posible definir una variable y asociar el valor a un conjunto de posibles valores válidos. Además, el compilador de C# debería ser capaz de evitar que los programadores asignen a la variable un valor no definido del conjunto de valores posibles. Como resultado, C# admite un constructor llamado enumeración que se ocupa de este caso concreto.

Las *enumeraciones* son un grupo de constantes definidas con un nombre común. El nombre de la enumeración puede usarse como un tipo de variable una vez que se haya definido la enumeración. Cuando se usa una variable definida como un tipo de enumeración, el compilador de C# se asegura de que los valores asignados a las variables del tipo de la enumeración concuerden con uno de los valores del conjunto de constantes definidas en la definición de la enumeración.

Las enumeraciones son ideales para las situaciones en las que una variable debe asociarse a un conjunto de valores específico. Supongamos, por ejemplo, que está escribiendo una clase de C# que controla una puerta electrónica y decide escribir una propiedad para la clase llamada `DoorState`, que abre o cierra la puerta:

```
public int DoorState
{
    set
    {
        InternalDoorState = value;
    }
}
```

También puede definir algunas constantes que se pueden usar para hacer el código más legible:

```
public const int DoorStateOpen = 1;
public const int DoorStateClosed = 2;
```

La propiedad y las constantes permiten que el código que trabaja con los objetos de la clase pueda escribir código legible como el siguiente:

```
DoorStateObject = new DoorClass();

DoorObject.DoorState = DoorClass.DoorStateOpen;
DoorObject.DoorState = DoorClass.DoorStateClosed;
```

El código anterior se compila y ejecuta sin problemas. Sin embargo, la propiedad `DoorState` se define como un `int` y no hay nada que impida a los invocadores usar valores que no tienen sentido y asignarlos a la propiedad `DoorState`:

```
DoorObject.DoorState = 12345;
```

Este código también es válido porque el literal 12345 está dentro de los límites válidos de un entero de C# y la propiedad `DoorState` está definida como poseedora de un tipo `int`. Aunque este código es válido desde el punto de vista de la compilación de C#, no tiene sentido en el nivel de clase porque el estado de la puerta en realidad sólo debería ser abierta o cerrada.

Podría crear algún código para la verificación de errores en la propiedad `DoorState` para que sólo acepte valores válidos, pero sería incluso mejor hacer que el compilador de C# imponga la restricción por nosotros cuando se crea el código.

Las enumeraciones proporcionan el mecanismo en tiempo de compilación que está buscando. Le permiten agrupar las constantes relacionadas, como las constantes `DoorStateOpen` y `DoorStateClosed`, bajo un nombre de grupo y usar ese nombre de grupo como un tipo de valor. Puede, por ejemplo, agrupar las constantes `DoorStateOpen` y `DoorStateClosed` en una enumeración llamada `LegalDoorStates` y redefinir la propiedad `DoorState` para que trabajen con un tipo de `LegalDoorStates`, en lugar de con un `int`. El compilador de C# puede entonces asegurar que los valores asignados a la propiedad son miembros de la enumeración y producirá un error si el valor no existe en la enumeración.

Cómo declarar una enumeración

Puede declarar una enumeración en C# usando la siguiente sintaxis:

- La palabra clave `enum`.
- Un identificador de enumeración.
- Un tipo base opcional.
- Identificadores de valor de enumeración separados por comas y entre llaves.

La enumeración `LegalDoorStates` de la anterior sección se definiría como se indica a continuación:

```
enum LegalDoorStates
{
    DoorStateOpen,
    DoorStateClosed
}
```

Cada uno de los miembros de las enumeraciones tiene un valor numérico asociado. Por defecto, el valor numérico del primer valor es cero y el valor de cada uno de los otros miembros es una unidad mayor que el valor de la anterior enumeración. Si se usan estas reglas y la enumeración definida anteriormente, el valor

por defecto de `DoorStateOpen` es 0 y el valor de `DoorStateClosed` es 1. Si lo desea, puede invalidar estos valores asignando valores a los miembros cuando se definen usando el operador de asignación:

```
enum LegalDoorStates
{
    DoorStateOpen = 100,
    DoorStateClosed = 150
}
```

Puede asignar los miembros a un valor literal o al resultado de una expresión constante:

```
enum LegalDoorStates
{
    DoorStateOpen = (75 + 25),
    DoorStateClosed = 150
}
```

Si no se asigna un valor a un miembro concreto de la enumeración, se aplican las reglas de asignación de valor por defecto. Observe la siguiente enumeración:

```
enum LegalDoorStates
{
    DoorStateOpen = 100,
    DoorStateClosed
}
```

Usando esta enumeración y las reglas de asignación de valor por defecto, el valor de `DoorStateOpen` es 100 y el valor de `DoorStateClosed` es 101.

C# también permite el uso de un identificador de enumeración para asignar un valor a otro identificador:

```
enum LegalDoorStates
{
    DoorStateOpen = 100,
    DoorStateClosed,
    LastState = DoorStateClosed
}
```

En este ejemplo, el valor de la enumeración `LastState` es igual al valor de la enumeración `DoorStateClosed`, que en este caso es igual a 101. Esto demuestra que dos identificadores en una enumeración tienen el mismo valor.

Las enumeraciones corresponden a un tipo de valor particular. Este tipo correspondiente recibe el nombre de *tipo subyacente* de la enumeración. Las enumeraciones pueden ser convertidas explícitamente a su tipo subyacente. Por defecto, el tipo subyacente de todas las enumeraciones es `int`. Si quiere usar un tipo subyacente diferente, especifique el tipo subyacente después de dos puntos que siguen al identificador de la enumeración:

```
enum LegalDoorStates : short
{
    DoorStateOpen = 100,
    DoorStateClosed
}
```

Todas las asignaciones explícitas de valor deben usar valores que incluidos dentro de los límites válidos del tipo subyacente de la enumeración. Observe el error en la siguiente enumeración:

```
enum Weather : uint
{
    Sunny = -1,
    Cloudy = -2,
    Rain = -3,
    Snow = -4
}
```

Esta declaración de enumeración es un error porque el tipo subyacente es `uint` y las asignaciones usan valores negativos que están fuera de los valores legales de un `uint`. Si se compila la anterior enumeración, el compilador de C# emite los siguientes errores:

```
error CS0031: El valor constante '-1' no se puede convertir a
'uint'
error CS0031: El valor constante '-2' no se puede convertir a
'uint'
error CS0031: El valor constante '-3' no se puede convertir a
'uint'
error CS0031: El valor constante '-4' no se puede convertir a
'uint'
```

Cómo usar una enumeración

Tras haber definido la enumeración, se puede usar el identificador de enumeración como un tipo de variable. El listado 14.1 muestra cómo la clase `DoorController` puede usar una enumeración.

Listado 14.1. Cómo usar una enumeración

```
public enum LegalDoorStates
{
    DoorStateOpen,
    DoorStateClosed
}

class DoorController
{
    private LegalDoorStates CurrentState;
```

```

public LegalDoorStates State
{
    get
    {
        return CurrentState;
    }

    set
    {
        CurrentState = value;
    }
}

class MainClass
{
    public static void Main()
    {
        DoorController Door;

        Door = new DoorController();

        Door.State = LegalDoorStates.DoorStateOpen;
    }
}

```

La enumeración `LegalDoorStates` está definida fuera de una declaración de clase. Esto está permitido en C# y hace que la enumeración sea visible para todas las clases del archivo fuente. Una alternativa es definir las enumeraciones dentro de una declaración de clase usando palabras clave de ámbito (`public`, `protected`, `internal` o `private`) para especificar el modo en que la enumeración es visible para las otras clases.

Tras definir la enumeración `LegalDoorStates`, se puede usar su nombre como un tipo de variable. Se usa como tipo para el campo privado `CurrentState` de la clase `DoorController` y también como tipo de la propiedad pública `State` de la misma clase.

Para referirse a una enumeración del código se usan el nombre de la enumeración y uno de los identificadores de la enumeración. Estos identificadores están separados por un punto, como se muestra en la siguiente instrucción:

```
Door.State = LegalDoorStates.DoorStateOpen;
```

El valor de la expresión `LegalDoorStates.DoorStateOpen` es igual al valor del identificador `DoorStateOpen` de la enumeración `LegalDoorStates`. Este valor se asigna a la propiedad `State`. La ventaja de este diseño basado en enumeraciones es que el compilador puede identificar los lugares donde el código intenta asignar a la propiedad `State` un valor diferente del valor procedente de la enumeración. Observe el error en la siguiente instrucción:

```
Door.State = 12345;
```

El anterior código es un error porque la propiedad `State` está definida como si tomase un valor de tipo `LegalDoorStates` y en su lugar se le asigna un valor entero. El código del anterior ejemplo produce el siguiente error del compilador de C#:

```
error CS0029: No se puede convertir implícitamente el tipo
'int' a 'LegalDoorStates'
```

Cómo usar operadores en valores de enumeración

Debido a que los valores enumerados tienen un tipo subyacente y un valor, lo lógico es que escriba un código que trate con valores subyacentes. Puede usar varios operadores de C# con valores enumerados:

- igualdad
- desigualdad
- menor que
- mayor que
- menor o igual que
- mayor o igual que
- suma
- resta
- AND
- OR exclusivo
- OR inclusivo
- complemento bit a bit
- incremento
- decremento

Por ejemplo, observe el listado 14.2.

Listado 14.2. Cómo usar operadores con enumeraciones

```
using System;

public enum FileAttributes
{
```

```

    AttrNone = 0,
    AttrReadOnly = 1,
    AttrHidden = 2,
    AttrReadyForArchive = 4
}

class MainClass
{
    public static void Main()
    {
        FileAttributes FileAttr;

        FileAttr = FileAttributes.AttrReadOnly |
FileAttributes.AttrHidden;
        Console.WriteLine(FileAttr);
    }
}

```

El código del listado 14.2 define un conjunto de valores enumerados que especifican atributos para un archivo o un disco. Un archivo puede no tener atributos especiales (`FileAttributes.AttrNone`), atributos de sólo lectura (`FileAttributes.AttrReadOnly`), atributos ocultos (`FileAttributes.Hidden`) o atributos listos para ser archivados (`FileAttributes.AttrReadyForArchive`).

El código del método `Main()` especifica una variable local llamada `FileAttr`, que es de tipo `FileAttributes`.

El código asigna el valor a un archivo oculto de sólo lectura mediante una operación `OR` sobre los atributos `FileAttributes.AttrReadOnly` y `FileAttributes.Hidden`. El valor de la variable local se escribe a continuación en la consola. Si se compila y ejecuta el listado 14.2 se escribe lo siguiente en la consola:

3

El listado 14.2 produce el valor 3 porque el valor de la enumeración `FileAttributes.AttrReadOnly`, 1, se unió al valor de la enumeración `FileAttributes.Hidden`, 2, en una operación `OR`. Realizar una operación booleana `OR` con los valores 1 y 2 produce un resultado de 3.

También puede convertir un valor enumerado a un valor con el tipo del tipo subyacente de la enumeración:

```

enum IntEnum
{
    EnumOne = 1,
    EnumTwo,
    EnumThree
}

IntEnum IntEnumValue;
int IntValue;

```

```
IntEnumValue = EnumTwo;
IntValue = (int) IntEnumValue; // el valor es 2
```

El código anterior convierte el valor de una variable de enumeración `IntEnumValue` a su equivalente entero y asigna el entero `IntValue` a ese valor. Como la variable `IntValue` es un entero estándar, se le puede asignar cualquier valor válido para un entero. No está limitado al conjunto de valores definidos por la enumeración, aunque se le asigna un valor que procede de una variable enumerada.

Cómo usar la clase `.NET System.Enum`

El tipo `enum` de `C#` es en realidad un alias de la clase `System.Enum` definida en `.NET Framework`. Podemos usar cualquiera de los miembros de la clase `.NET System.Enum` en las enumeraciones que definamos.

Cómo recuperar nombres de enumeración

El listado 14.3 muestra cómo el código puede trabajar con enumeraciones como objetos `System.Enum`. Se trata de una mejora del listado 14.1 que recupera el estado actual de la puerta y escribe el nombre del estado en la consola.

Listado 14.3. Cómo recuperar una nombre de enumeración mediante `GetName()`

```
using System;

public enum LegalDoorStates
{
    DoorStateOpen,
    DoorStateClosed
}

class DoorController
{
    private LegalDoorStates CurrentState;

    public LegalDoorStates State
    {
        get
        {
            return CurrentState;
        }

        set
        {
            CurrentState = value;
        }
    }
}
```

```

class MainClass
{
    public static void Main()
    {
        DoorController Door;
        string EnumName;

        Door = new DoorController();

        Door.State = LegalDoorStates.DoorStateOpen;
        EnumName = LegalDoorStates.GetName(typeof
(LegalDoorStates), Door.State);
        Console.WriteLine(EnumName);
    }
}

```

El método `Main()` del listado 14.3 usa el método `GetName()` de la clase `System.Enum` para obtener una cadena que represente un valor de enumeración. El primer parámetro es un objeto `Type` que especifica la enumeración a la que se consulta. La expresión `typeof(LegalDoorStates)` devuelve un objeto `.NET Type` para el tipo especificado (en este caso, la enumeración `LegalDoorStates`). El segundo parámetro es el valor de la enumeración actual, de la que debe devolverse la representación de su cadena. La siguiente instrucción muestra cómo puede usarse el método `GetName()` para obtener el nombre de un valor enumerado:

```

EnumName = LegalDoorStates.GetName(typeof(LegalDoorStates),
Door.State);

```

Esta instrucción se interpreta como: "Devuelve una cadena que represente el nombre del valor de la propiedad `Door.State`. Este valor es una parte de la enumeración `LegalDoorStates`."

Si se ejecuta el listado 14.3 se escribe lo siguiente en la consola:

```

DoorStateOpen

```

También se puede usar el método `Format()` para recuperar el nombre de un valor de enumeración, según su valor numérico. La llamada `GetName()` del listado 14.3 puede reemplazarse por la siguiente llamada a `Format()`:

```

EnumName = LegalDoorStates.Format(typeof(LegalDoorStates), 0, "g");

```

El primer parámetro de `Format()` es el mismo que el primer parámetro de `GetNames()`, que es el tipo de enumeración que se usa en la llamada. El segundo parámetro de `Format()` es el valor numérico que debe devolver la llamada. El último parámetro de `Format()` es la cadena que especifica los contenidos de la cadena que debe devolver la llamada. La cadena de formato puede ser una de las siguientes:

- `g`, que especifica que debe devolverse el valor de enumeración con el valor numérico que concuerde con el valor del segundo parámetro.

- `x`, que especifica que el valor del segundo parámetro debe devolverse como una cadena que represente el valor en notación hexadecimal
- `d`, que especifica que el valor del segundo parámetro debe devolverse como una cadena que represente el valor en notación hexadecimal
- `f`, que especifica que el valor debe ser tratado como un conjunto de valores enumerados combinados y que el método debe devolver una lista de valores delimitada por comas como una cadena

El valor de formato `f` se creó para ser usado con enumeraciones que representan valores de bit. Observe la siguiente enumeración:

```
public enum BitsToSet
{
    Bit0Set = 1,
    Bit1Set = 2,
    Bit2Set = 4,
    Bit3Set = 8,
    Bit4Set = 16,
    Bit5Set = 32,
    Bit6Set = 64,
    Bit7Set = 128
}
```

La enumeración anterior representa un conjunto de bits que pueden asignarse a un byte. Se pueden asignar varios bits a una variable mediante el operador booleano `OR`, como en el siguiente ejemplo:

```
BitsToSet Byte;
Byte = BitsToSet.Bit1Set | BitsToSet.Bit3Set | BitsToSet.Bit6Set;
```

Llamar al método `Format()` en la variable `Byte` con el parámetro de formato `f` devuelve una cadena que representa los nombres de los valores enumerados cuyos valores se encuentran en la variable:

```
Bit1Set, Bit3Set, Bit6Set
```

Cómo comparar valores de enumeración

El método `CompareTo()` de la clase `System.Enum` puede comparar una enumeración con otra y devolver un entero que describe la relación entre los dos valores. Observe el listado 14.4, que compara el valor de una variable enumerada con el valor determinado de la misma enumeración:

Listado 14.4. Cómo comparar valores de enumeración con `CompareTo()`

```
using System;

public class MainClass
```



```

{
    public enum Color
    {
        Red = 0,
        Orange,
        Yellow,
        Green,
        Blue,
        Indigo,
        Violet
    }

    static void Main()
    {
        Color MyColor;

        MyColor = Color.Green;
        Console.WriteLine("{0}", MyColor.CompareTo(Color.Red));
        Console.WriteLine("{0}", MyColor.CompareTo(Color.Green));
        Console.WriteLine("{0}", MyColor.CompareTo(Color.Violet));
    }
}

```

El listado 14.4 declara una clase con una enumeración pública llamada `Color`. Sus valores varían de 0 a 6. El método `Main()` declara una variable de tipo `Color` llamada `MyColor` y asigna el valor `Green` a la variable. A continuación invoca a `CompareTo()` para comparar el valor de la variable con otros valores de la enumeración. El método `CompareTo()` devuelve uno de estos tres valores:

- -1 si el valor que se pasa como argumento a `CompareTo()` tiene un valor superior al valor enumerado usado para invocar al método
- 1 si el valor que se pasa como argumento a `CompareTo()` tiene un valor inferior al valor enumerado usado para invocar al método
- 0 si los dos valores son iguales

En el listado 14.4, se llama tres veces al método `CompareTo()`. En la primera llamada, la variable `MyColor` se compara con el valor `Red`. Como `Green`, que tiene el valor 3, tiene un valor superior a `Red`, que tiene el valor 0, `CompareTo()` devuelve 1. En la segunda llamada, la variable `MyColor` se compara con el valor `Green`. Como los valores son iguales, `CompareTo()` devuelve 0. En la última llamada, la variable `MyColor` se compara con el valor `Violet`. Como `Green`, que tiene el valor 3, tiene un valor inferior a `Violet`, que tiene un valor 6, `CompareTo()` devuelve -1.

El argumento usado en la llamada a `CompareTo()` debe ser del mismo tipo que la enumeración usada para llamar al método. Usar cualquier otro tipo, incluso el tipo subyacente de la enumeración, produce un error en tiempo de ejecución.

Cómo descubrir el tipo subyacente en tiempo de ejecución

Descubrir el tipo subyacente de una enumeración en tiempo de ejecución es sencillo con el método `GetUnderlyingType()`. Este método, al que se llama en el tipo de enumeración, en lugar de una variable del tipo, toma un parámetro `Type` que representa el tipo de la enumeración y devuelve otro objeto `Type` que representa el tipo subyacente de la enumeración.

El método `ToString()` puede ser llamado en el objeto `Type` devuelto para obtener un nombre legible para el tipo, como se muestra en el siguiente código:

```
string FormatString;
Type UnderlyingType;

UnderlyingType =
BitsToSet.GetUnderlyingType(typeof(BitsToSet));
Console.WriteLine(UnderlyingType.ToString());
```

Este código recupera el tipo subyacente para una enumeración llamada `BitsToSet` y escribe el nombre del tipo en la consola, que produce una cadena como la siguiente:

```
System.Int32
```

Cómo recuperar todos los valores de enumeración

El método `GetValues()` devuelve una matriz de todos los valores de enumeración ordenados en orden ascendente según su valor numérico, como se puede ver en el siguiente código:

```
Array ValueArray;

ValueArray = Color.GetValues(typeof(Color));
foreach(Color ColorItem in ValueArray)
    Console.WriteLine(ColorItem.ToString());
```

Este código llama a `GetValues()` en la enumeración `Color` definida con anterioridad.

El método `GetValues()` devuelve una matriz y se visita a los elementos de la matriz de uno en uno mediante la palabra clave `foreach`. El nombre de cada elemento de la matriz se escribe en la consola, como se puede ver a continuación:

```
Red
Orange
```

```
Yellow
Green
Blue
Indigo
Violet
```

Análisis de cadenas para recuperar valores de enumeración

La clase `Enum` contiene un método de análisis de cadenas llamado `Parse()`, que acepta una cadena como entrada de datos y devuelve el valor de la enumeración cuyo nombre concuerda con la cadena proporcionada, como se puede ver en el siguiente ejemplo:

```
Color.Parse(typeof(Color), "Blue");
```

Esta llamada devuelve un objeto que representa el valor enumerado llamado `Blue` en una enumeración llamada `Color`. Como muchos otros métodos de enumeración, el método `Parse()` es llamado en el tipo, en lugar de una variable del tipo.

El método `Parse()` devuelve un objeto, que necesita ser convertido explícitamente en un valor del tipo apropiado. El siguiente ejemplo muestra cómo el método `Parse()` puede ser usado como uno de los muchos modos de representar un valor enumerado:

```
Color ColorValue;
object ParsedObject;

ParsedObject = Color.Parse(typeof(Color), "Blue");
Console.WriteLine(ParsedObject.GetType().ToString());
ColorValue = (Color)ParsedObject;
Console.WriteLine(ColorValue.ToString());
Console.WriteLine(Color.Format(typeof(Color), ColorValue, "d"));
```

En este código, se llama a `Parse()` en el tipo de enumeración `Color` y se le otorga una cadena de entrada `Blue`. Esta llamada devuelve un objeto y el código escribe el tipo del objeto en la consola.

El objeto se convierte entonces explícitamente en una variable del tipo `Color` y se escribe en la consola el nombre del valor de la enumeración y el valor decimal:

```
MainClass+Color
Blue
4
```

Este resultado demuestra que el objeto devuelto por el método `Parse()` es del tipo `Color`. La variable convertida, que es una variable de `Color`, tiene un nombre de cadena `Blue` y un valor decimal `4`.

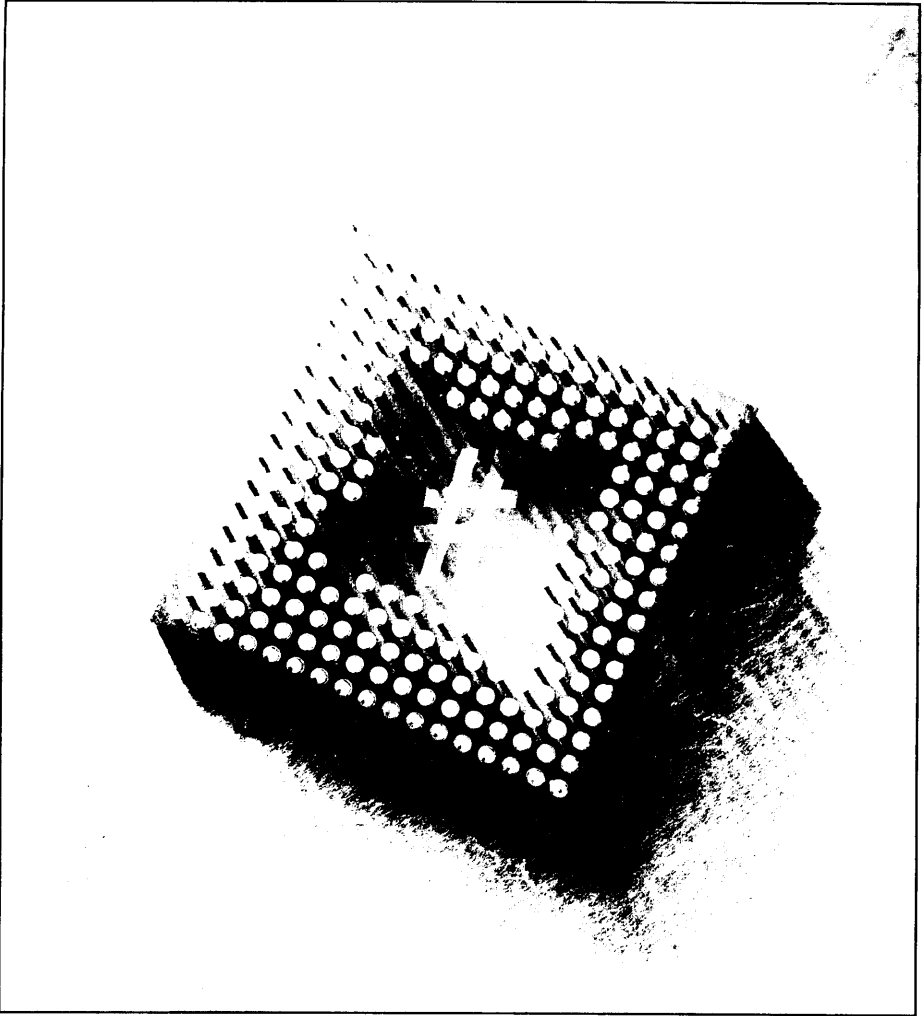
Resumen

Las enumeraciones se emplean para agrupar un conjunto de constantes relacionadas. Al dar a sus enumeraciones un nombre, se puede usar ese nombre en el código como un tipo de variable una vez que se haya definido la enumeración. Las enumeraciones, por defecto, se basan en un conjunto de constantes `int`. Se puede invalidar este valor por defecto especificando un tipo subyacente para la enumeración. Se pueden usar como tipo subyacente de una enumeración muchos de los tipos numéricos de C#. Se deben emplear enumeraciones cuando queramos que el compilador de C# garantice que las constantes con las que trabajamos en el código proceden de un conjunto de valores válidos.

Por defecto, el compilador de C# asigna valores numéricos a los identificadores de las enumeraciones. El primer identificador tiene el valor de cero y las otras enumeraciones aumentan su valor a partir de ahí. Si lo deseamos, se puede usar el operador de asignación para asignar un valor a un identificador de enumeración cuando se define la enumeración.

Un valor en una enumeración se especifica escribiendo el nombre de la enumeración, un punto y el nombre de los identificadores de enumeración. Los identificadores de enumeración pueden convertirse implícitamente al tipo subyacente de la enumeración. Esta conversión implícita también permite el uso de algunos de los operadores de C# para trabajar con los valores de enumeración.

Todas las enumeración de C# derivan de una clase base de .NET llamada `System.Enum`. La clase `System.Enum` contiene algunos métodos útiles que pueden ayudar a obtener las máximas prestaciones de las enumeraciones. Este capítulo ha examinado la mayoría de estos métodos.



15 Eventos y delegados

En el flujo general del típico segmento de software orientado a objetos, un fragmento de código crea un objeto de una clase y llama a métodos de ese objeto. En este contexto, el invocador es el código activo porque es el código el que llama a los métodos. El objeto es pasivo, en el sentido de que espera y realiza una acción sólo cuando se invoca a uno de sus métodos.

Sin embargo, también se puede producir el contexto contrario. Un objeto puede realizar una tarea y avisar al invocador cuando ocurre algo durante el proceso. A este *algo* se le llama un *evento* y la publicación de ese evento del objeto se le conoce como *desencadenar un evento*.

El proceso activado por eventos, en el que fragmentos de código informan a otras piezas de código cuándo se producen eventos relevantes, no es una novedad de .NET. La capa de interfaz de usuario de Windows siempre ha usado una forma de eventos para informar a las aplicaciones Windows cuándo los usuarios trabajan con el ratón, presionan una tecla en el teclado o mueven una ventana. Los controles ActiveX desencadenan eventos para los contenedores de control de ActiveX cuando el usuario realiza una acción que afecta al control.

El lenguaje C# contiene palabras clave especiales que hacen que sea fácil desencadenar, publicar y procesar eventos en el código de C#. Se pueden usar estas palabras clave para permitir que las clases de C# desencadenen y procesen eventos con el mínimo esfuerzo.

Cómo definir delegados

Cuando se programan los eventos que desencadenan las clases de C#, es necesario decidir cómo recibe el evento otros fragmentos de código. Otros fragmentos de código necesitan escribir un método que reciba y procese los eventos que se publican. Suponga, por ejemplo, que su clase implementa un servidor Web y quiere desencadenar un evento siempre que llega una solicitud de una página desde Internet. Otros fragmentos de código quizás quieran realizar alguna acción cuando la clase desencadene este evento `new request` y ese código debería incluir un método que se ejecute cuando se desencadene el evento.

El método que implementan los usuarios de la clase para recibir y procesar los eventos es definido por un concepto de C# llamado *delegado*. Un delegado es una especie de "función patrón" que describe el aspecto que tiene el controlador de eventos del usuario. Un delegado también es una clase que tiene una firma y contiene referencias a métodos. Es como una función puntero, pero puede contener referencias a métodos estáticos y de instancia. Para los métodos de instancia, el delegado almacena una referencia al objeto y al punto de entrada de la función. Un delegado define lo que debe devolver el controlador de eventos del usuario y lo que debe ser la lista de parámetros.

Para definir un delegado en C#, hay que usar la siguiente sintaxis:

- La palabra clave de C# `delegate`.
- El tipo devuelto por el controlador de eventos.
- El identificador de delegado.
- La lista de parámetros del controlador de eventos, entre paréntesis.

Si se declaran delegados en la clase que desencadena el evento, se les puede anteponer las palabras clave `public`, `protected`, `internal` o `private` como se pueden ver en este ejemplo de una definición `delegate`.

```
public delegate void EvenNumberHandler(int Number);
```

En este ejemplo se crea un delegado público llamado `EvenNumberHandler` que no devuelve nada. Este delegado sólo define un parámetro para ser pasado, de tipo `int`. El identificador de delegado, `EvenNumberHandler`, puede ser cualquier nombre que elija mientras no le dé el nombre de una palabra clave de C#.

Cómo definir eventos

Para aclarar lo que es en realidad un evento, vamos a empezar con un ejemplo. Está conduciendo en su coche y aparece en el salpicadero la luz que indica que queda poco combustible. Lo que en realidad ha ocurrido es que un sensor en el

depósito de gasolina ha avisado al ordenador de que el nivel de combustible está bajo. El ordenador entonces desencadena un evento que a su vez enciende la luz del salpicadero para que sepa que tiene que comprar más combustible. En pocas palabras, un evento es un medio que tiene el ordenador de avisarle de una condición.

Para definir un evento que desencadene una clase se usa la palabra clave de C# `event`. En su forma más simple, las declaraciones de eventos de C# usan la siguiente sintaxis:

- La palabra clave de C# `event`.
- El tipo de evento.
- El identificador de evento.

El tipo de evento concuerda con un identificador de delegado, como se muestra en el siguiente ejemplo de servidor Web:

```
public delegate void NewRequestHandler(string URL);

public class WebServer
{
    public event NewRequestHandler NewRequestEvent;
    // ...
}
```

Este ejemplo declara un delegado llamado `NewRequestHandler`. `NewRequestHandler` define un delegado que sirve como una plantilla para los métodos que procesan el evento `new request`. Todos los métodos que necesitan procesar el evento `new request` deben obedecer las convenciones de llamada del delegado: no deben devolver ningún dato y deben tener una sola cadena como lista de parámetros. Las implementaciones de control de eventos pueden tener cualquier nombre de método mientras su tipo devuelto y su lista de parámetros concuerden con el patrón de delegados.

La clase `WebServer` define un evento llamado `NewRequestEvent`. El tipo de este evento es `NewRequestHandler`. Esto significa que sólo los controles de evento escritos para que concuerden con las convenciones de llamada del delegado pueden ser usadas para procesar el evento `NewRequestEvent`.

Cómo instalar eventos

Tras escribir el controlador de eventos, hay que crear una nueva instancia de él e instalarlo en la clase que desencadena el evento. Para crear una nueva instancia de controlador de eventos debe crear una nueva variable del tipo delegado y pasar el nombre del método controlador de eventos como un argumento. Usando el ejemplo del cliente Web, la creación de una nueva instancia controladora de eventos puede tener este aspecto:


```
public void MyNewRequestHandler(string URL)
{
}
```

```
NewRequestHandler HandlerInstance;
```

```
HandlerInstance = new NewRequestHandler(MyNewRequestHandler);
```

Tras crear la nueva instancia controladora de eventos, se usa el operador `+=` para añadirla a la variable de evento:

```
NewRequestEvent += HandlerInstance;
```

Esta instrucción enlaza la instancia de delegado `HandlerInstance`, que admite el método `MyNewRequestMethod`, con el evento `NewRequestEvent`. Mediante el operador `+=`, se pueden enlazar tantas instancias de delegado como quiera para un evento.

Del mismo modo, puede usar el operador `-=` para eliminar una instancia de delegado de un evento:

```
NewRequestEvent -= HandlerInstance;
```

Esta instrucción desenlaza la instancia de delegado `HandlerInstance` del evento `NewRequestEvent`.

Cómo desencadenar eventos

Se puede desencadenar un evento desde una clase usando el identificador del evento (como el nombre del evento) como si fuera un método. La acción de invocar un evento como un método desencadena el evento. Desencadenar el evento `new request` en el ejemplo del servidor Web puede tener el siguiente aspecto:

```
NewRequestEvent(strURLofNewRequest);
```

Los parámetros usados para desencadenar el evento deben coincidir con la lista de parámetros del delegado del evento. El delegado del evento `NewRequestEvent` se definió para aceptar un parámetro de cadena; por tanto, se debe proporcionar una cadena cuando se desencadene el evento desde la clase del cliente Web.

Cómo unirlo todo

El listado 15.1 muestra delegados y eventos en acción. El código implementa una clase que cuenta desde 0 hasta 100 y desencadena un evento cuando encuentra un número impar durante el proceso de contado.

Listado 15.1. Cómo recuperar eventos de número impar

```
using System;

public delegate void EvenNumberHandler(int Number);

class Counter
{
    public event EvenNumberHandler OnEvenNumber;

    public Counter()
    {
        OnEvenNumber = null;
    }

    public void CountTo100()
    {
        int CurrentNumber;

        for(CurrentNumber = 0; CurrentNumber <= 100;
CurrentNumber++)
        {
            if(CurrentNumber % 2 == 0)
            {
                if(OnEvenNumber != null)
                {
                    OnEvenNumber(CurrentNumber);
                }
            }
        }
    }
}

class EvenNumberHandlerClass
{
    public void EvenNumberFound(int EvenNumber)
    {
        Console.WriteLine(EvenNumber);
    }
}

class MainClass
{
    public static void Main()
    {
        Counter MyCounter = new Counter();
        EvenNumberHandlerClass MyEvenNumberHandlerClass = new
EvenNumberHandlerClass();
        MyCounter.OnEvenNumber += new
EvenNumberHandler(MyEvenNumberHandlerClass.EvenNumberFound);
        MyCounter.CountTo100();
    }
}
```

Para compilar esta aplicación hay que crear una nueva aplicación de consola en Visual Studio y copiar en ella el código fuente o simplemente usar el bloc de notas para guardar el archivo y a continuación usar:

```
csc <filename>
```

El listado 15.1 implementa tres clases:

- La clase `Counter` es la clase que realiza el cálculo. Implementa un método público llamado `CountTo100()` y un evento público llamado `OnEvenNumber`. El evento `OnEvenNumber` es del tipo delegado `EvenNumberHandler`.
- La clase `EvenNumberHandlerClass` contiene un método público llamado `EvenNumberFound`. Este método actúa como el controlador de evento para el evento `OnEvenNumber` de clase `Counter`. Imprime en la consola el entero proporcionado como parámetro.
- La clase `MainClass` contiene el método `Main()` de la aplicación.

El método `Main()` crea un objeto de clase `Counter` y da nombre al objeto `MyCounter`. También crea un nuevo objeto de clase `EvenNumberHandlerClass` y llama al objeto `MyEvenNumberHandlerClass`.

El método `Main()` llama al método `CountTo100()` del objeto `MyCounter`, pero no antes de instalar una instancia de delegado en la clase `Counter`. El código crea una nueva instancia de delegado que gestiona el método `EvenNumberFound` del objeto `MyEventNumberHandlerClass` y lo añade al evento `OnEvenNumber` del objeto `MyCounter` usando el operador `+=`.

La implementación del método `CountTo100` usa una variable local para contar desde 0 a 100. Cada vez que pasa por el bucle contador, el código comprueba si el número es par examinando si el número tiene resto al dividirse entre dos. Si el número es par, el código desencadena el evento `OnEvenNumber` y proporciona el número par como el argumento para hacerlo coincidir con la lista de parámetros del delegado de eventos.

Como el método `EvenNumberFound` de `MyEvenNumberHandlerClass` estaba instalado como un controlador de eventos y como ese método escribe el parámetro proporcionado en la consola, si se compila y ejecuta el código del listado 15.1, se escriben en la consola los números pares entre 0 y 100.

Cómo estandarizar un diseño de evento

Aunque C# acepta perfectamente cualquier diseño de delegado que se pueda compilar, .NET Framework prefiere que se adopte un diseño estándar para los delegados. El diseño de delegados preferido usa dos argumentos: por ejemplo, el `System.EventHandler`:

- Una referencia al objeto que desencadenó el evento.
- Un objeto que contiene datos relacionados con el evento.

El segundo parámetro, que contiene todos los datos del evento, debe ser un objeto de una clase que derive de una clase .NET llamada `System.EventArgs`. El listado 15.2 remodela el listado 15.1 usando este diseño preferido.

Listado 15.2. Cómo recuperar números impares con la convención de delegados .NET

```
using System;

public delegate void EvenNumberHandler(object Originator,
OnEvenNumberEventArgs EvenNumberEventArgs);

class Counter
{
    public event EvenNumberHandler OnEvenNumber;

    public Counter()
    {
        OnEvenNumber = null;
    }

    public void CountTo100()
    {
        int CurrentNumber;

        for(CurrentNumber = 0; CurrentNumber <= 100;
CurrentNumber++)
        {
            if(CurrentNumber % 2 == 0)
            {
                if(OnEvenNumber != null)
                {
                    OnEvenNumberEventArgs EventArguments;

                    EventArguments = new
OnEvenNumberEventArgs(CurrentNumber);
                    OnEvenNumber(this, EventArguments);
                }
            }
        }
    }
}

public class OnEvenNumberEventArgs : EventArgs
{
    private int EvenNumber;

    public OnEvenNumberEventArgs(int EvenNumber)
```

```

    {
        this.EvenNumber = EvenNumber;
    }

    public int Number
    {
        get
        {
            return EvenNumber;
        }
    }
}

class EvenNumberHandlerClass
{
    public void EvenNumberFound(object Originator,
OnEvenNumberEventArgs EvenNumberEventArgs)
    {
        Console.WriteLine(EvenNumberEventArgs.Number);
    }
}

class MainClass
{
    public static void Main()
    {
        Counter MyCounter = new Counter();
        EvenNumberHandlerClass MyEvenNumberHandlerClass = new
EvenNumberHandlerClass();
        MyCounter.OnEvenNumber += new
EvenNumberHandler(MyEvenNumberHandlerClass.EvenNumberFound);
        MyCounter.CountTo100();
    }
}

```

El listado 15.2 añade una nueva clase llamada `OnEvenNumberEventArgs` que deriva de la clase `.NET EventArgs`. Implementa un constructor que toma un número entero y lo almacena en una variable privada. También expone una propiedad de sólo lectura llamada `Number`, que devuelve el valor de la variable privada.

La firma del delegado también ha cambiado para cumplir con la nueva convención. Ahora acepta dos parámetros de entrada: una referencia al objeto que desencadena el evento y un objeto de tipo `OnEvenNumberEventArgs`.

Cuando la clase `Counter` se prepara para desencadenar el evento, antes crea un nuevo objeto de tipo `OnEvenNumberEventArgs` y lo inicializa con el número impar. A continuación pasa este objeto al evento como segundo parámetro.

La nueva implementación del método `EvenNumberFound` examina el segundo parámetro, un objeto de clase `OnEvenNumberEventArgs` y escribe el valor de la propiedad `Number` del objeto en la consola.

Cómo usar descriptores de acceso de eventos

En la implementación general de objetos, hay que tener un campo de eventos definido en la clase por cada evento posible que pueda desencadenar la clase. En ejemplos como el listado 15.2, en el que la clase sólo desencadena un evento, definir un campo de eventos para cada evento no es muy costoso. Sin embargo, este sistema es mucho más complicado cuando la clase puede desencadenar uno de varios eventos.

Tome, por ejemplo, una clase de C# que gestiona un componente de la interfaz de usuario de Windows. Los componentes normales de la interfaz de usuario de Windows pueden recibir uno de los muchos mensajes posibles del sistema operativo y podríamos querer diseñar nuestra clase para que se envíen a los usuarios de la clase, mediante un evento de C#, los mensajes que el componente del interfaz de usuario recibe del sistema operativo. Definir un campo de evento en una clase para cada posible mensaje de Windows obligaría a la clase a almacenar una gran cantidad de campos y haría que tuviera un tamaño enorme. C# admite una forma alternativa por la que los eventos pueden definirse como propiedades, en lugar de como campos. Las propiedades de eventos funcionan exactamente igual que las propiedades de clase estándar, que se implementan con código, en lugar de campos de datos. A diferencia de una propiedad estándar, una propiedad de evento usa las palabras clave `add` y `remove` para definir bloques de código:

```
public event EventHandler OnEvenNumber
{
    add
    {
    }
    remove
    {
    }
}
```

El código del bloque de código `add` se invoca cuando un usuario añade un nuevo controlador de eventos al evento usando el operador `+=`.

La ventaja de usar descriptores de acceso de eventos es que tenemos total libertad en lo que respecta al modo en que almacenamos los controladores de eventos. En lugar de definir campos separados para cada evento, podemos almacenar una sola lista enlazada o una matriz de controladores y podemos implementar el descriptor de acceso de eventos `remove` para eliminar un controlador de eventos de la matriz o de la lista.

Como en las propiedades estándar, podemos usar la palabra clave de C# `value` para hacer referencia al controlador de evento que se añade o elimina, como se puede ver en la siguiente instrucción:

```
MyCounter.OnEvenNumber += new
EvenNumberHandler(MyEvenNumberHandlerClass.EvenNumberFound);
```

En esta instrucción, se añade al evento `OnEvenNumber` un nuevo `EvenNumberHandler` objeto. Si implementásemos el evento como una propiedad, el bloque de código `add` podría usar la palabra clave `add` para hacer referencia al nuevo objeto `EvenNumberHandler`:

```
public event EvenNumberHandler OnEvenNumber
{
    add
    {
        AddToList(value);
    }
    remove
    {
        RemoveFromList(value);
    }
}
```

Cuando se usa en descriptores de acceso de eventos, la palabra clave `value` es una variable de tipo `Delegate`.

Cómo usar modificadores de eventos

Se puede anteponer a una declaración de un evento uno de los siguientes modificadores:

- `Static`
- `Virtual`
- `Override`
- `Abstract`

Eventos estáticos

Los eventos modificados con la palabra clave `static` se comportan de forma parecida a los campos estáticos, en el sentido de que, aunque cada copia de una clase contiene copias separadas de todos los campos, sólo puede haber una copia de un miembro estático en un momento dado. Todos los objetos de la clase comparten los eventos estáticos. Cuando se les hace referencia, debe ser a través del nombre de la clase y no mediante el nombre del objeto, como se aprecia a continuación:

```
public class Counter
{
```

```
public static event EventHandler OnEvenNumber;  
// ...  
}  
  
Counter.OnEvenNumber += new  
EventHandler(MyEventHandlerClass.EventNumberFound);
```

Como puede ver, debe hacer referencia al evento estático `OnEvenNumber` especificando el nombre de la clase y el nombre del objeto.

Eventos virtuales

Los eventos modificados con la palabra clave `virtual` marcan cualquier descriptor de acceso `add` o `remove` como virtual. Los descriptores de acceso virtuales en clases derivadas pueden ser reemplazados.

Eventos de reemplazo

Los eventos modificados con la palabra clave `override` marcan cualquier descriptor de acceso `add` o `remove` como eventos de reemplazo `add` o `remove` con el mismo nombre en una clase base.

Eventos abstractos

Los eventos modificados con la palabra clave `abstract` marcan cualquier descriptor de acceso `add` o `remove` como abstracto. Los descriptores de acceso abstractos no proporcionan una implementación propia: en su lugar, un evento de reemplazo de una clase derivada proporciona una implementación.

Resumen

Las clases pueden desencadenar eventos cuando un programa pide a sus clientes que le avisen de las acciones llevadas a cabo por la clase. Sin eventos, los usuarios llaman a un método para realizar una operación, pero en realidad no saben lo avanzada que está la operación. Imagine, por ejemplo, un método que recupera una página Web de un servidor Web. Esa operación consiste en varios pasos:

- Conectar con el servidor Web.
- Solicitar la página Web.
- Recuperar la página Web devuelta.
- Desconectar del servidor Web.

Es posible diseñar una clase como ésta con eventos que se desencadenen cuando comience cada una de estas acciones. Al desencadenar los eventos en los pasos críticos del proceso le da pistas a los usuarios de su clase sobre en qué parte del proceso se encuentra el código.

Los invocadores responden a los eventos registrando métodos llamados *controladores de eventos*. Los controladores de eventos se invocan cuando una clase desencadena algún evento. Estos métodos se corresponden con la lista de parámetros y devuelven un valor de un método patrón especial llamado *delegado*. Un delegado describe el diseño de un controlador de eventos, indicando qué parámetros debe admitir y cómo debe ser su código devuelto.

Los controladores de eventos se instalan usando el operador `+=`. Los eventos se declaran normalmente como campos públicos en una clase y los invocadores añaden sus controladores de eventos a la clase creando un nuevo objeto de la clase delegado y asignando el objeto al evento usando el operador `+=`. C# permite especificar varios controladores de eventos para un solo evento y también permite usar el operador `-=` para eliminar un controlador de eventos de un evento.

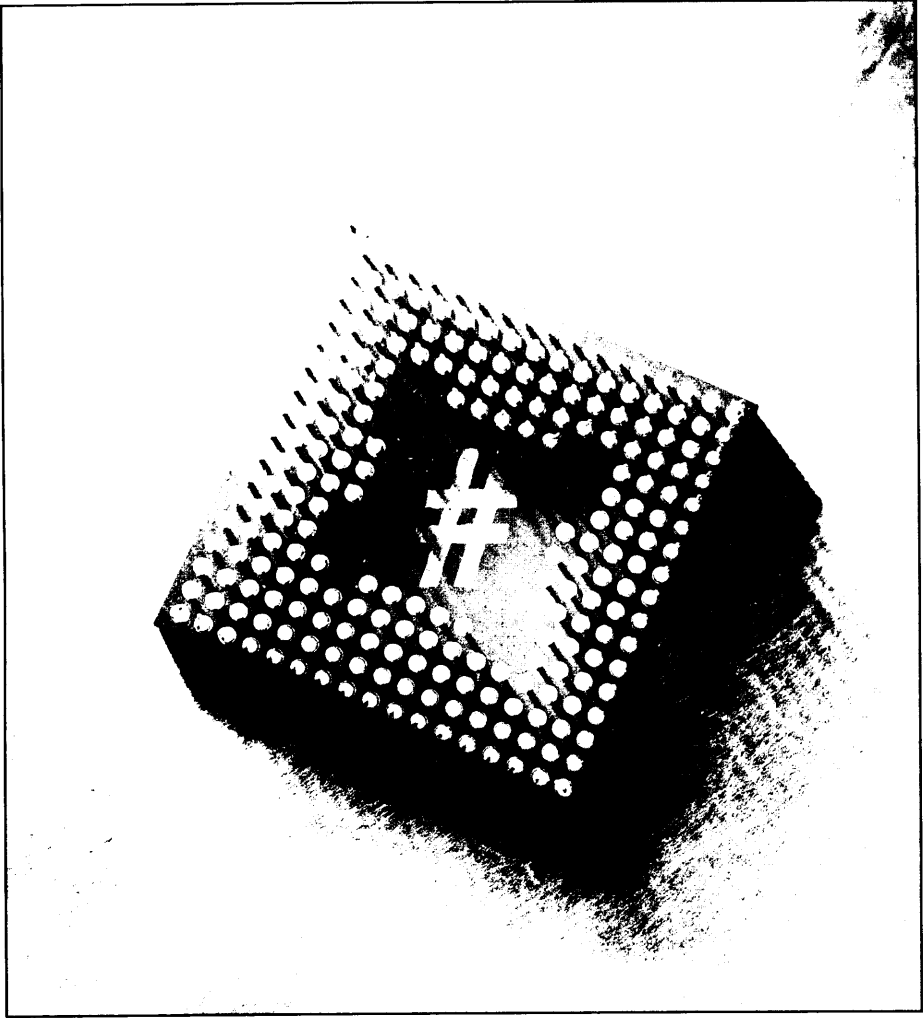
C# no obliga a usar un único patrón de diseño para los delegados, pero .NET Framework recomienda un diseño. Usar el diseño recomendado proporciona un estándar que, cuando se respeta, puede dar a los delegados una lista de parámetros de método: un objeto que especifica los objetos que desencadenan el evento y un objeto de una clase derivada de la clase `System.EventArgs`, que contiene los argumentos para el evento.

C# hace que sea muy sencillo implementar eventos en las clases. En el nivel más básico, cada evento se declara como un campo público y se pueden gestionar tantos eventos como se desee. Si la clase va a gestionar varios eventos y el número de campos públicos dentro de la clase parece tener una cantidad excesiva de código, se pueden escribir descriptores de acceso de eventos que permitan controlar el modo en que la clase gestiona los controladores de eventos. En lugar de definir campos públicos para los eventos, los descriptores de acceso de eventos permiten definir los eventos como propiedades con bloques de código `add` y `remove`. El bloque de código `add` se invoca cuando se añade un controlador de eventos a un evento y el bloque de código `remove` se invoca cuando se elimina un controlador de eventos de un evento. Estos bloques de código se pueden implementar almacenando los controladores de eventos en una matriz o en una lista para ser usados posteriormente.

El concepto de C# de eventos y delegados es un concepto nuevo para la familia de lenguajes C. Los eventos pueden ser desencadenados en C y C++ usando otros mecanismos, pero estos lenguajes no definen palabras clave para hacer que funcionen los eventos. En C#, los eventos y delegados son elementos completamente definidos por sí mismos y tanto el lenguaje como el compilador tienen compatibilidad integrada para el control de eventos.

Otra ventaja de usar eventos en C# es que son completamente compatibles con el CLR, lo que significa que se puede preparar un mecanismo de evento en

C# y desencadenar eventos que se controlen por otro lenguaje .NET. Como el CLR admite eventos en el nivel de tiempo de ejecución, se puede desencadenar un evento en C# y controlarlo y procesarlo con otro lenguaje, como Visual Basic .NET.



16 Control de excepciones

Buscar errores y manejarlos adecuadamente es un principio fundamental para diseñar software correctamente. En teoría, escribimos el código y cada línea funciona como pretendemos y los recursos que empleamos siempre están presentes. Sin embargo, éste no siempre es el caso en el mundo real. Otros programadores (por supuesto, no nosotros) pueden cometer errores, las conexiones de red pueden interrumpirse, los servidores de bases de datos pueden dejar de funcionar y los archivos de disco pueden no tener los contenidos que las aplicaciones creen que contienen. En pocas palabras, el código que se escribe tiene que ser capaz de detectar errores como éstos y responder adecuadamente.

Los mecanismos para informar de errores son tan diversos como los propios errores. Algunos métodos pueden estar diseñados para devolver un valor booleano que indican el éxito o el fracaso de un método. Otros métodos pueden escribir errores en un fichero de registro o una base de datos de algún tipo. La variedad de modelos de presentación de errores nos indica que el código que escribamos para controlar los errores debe ser bastante consistente. Cada método puede informar de un error de un modo distinto, lo que significa que la aplicación estará repleta de gran cantidad de código necesario para detectar los diferentes tipos de errores de las diferentes llamadas al método.

.NET Framework proporciona un mecanismo estándar, llamado *control de excepciones estructurado* (SEH), para informar de errores. Este mecanismo de-

pende de las excepciones para indicar los fallos. Las excepciones son clases que describen un error. .NET Framework las usa para informar de errores y podemos utilizarlas en nuestro código. Puede escribir código que busque excepciones generadas por cualquier fragmento de código, tanto si procede del CLR como si procede de nuestro propio código y podemos ocuparnos de la excepción generada adecuadamente. Usando SEH, sólo necesitamos crear un diseño de control de errores para nuestro código.

Esta metodología unificada del proceso de errores también es crucial para permitir la programación .NET multilingüe. Al diseñar todo nuestro código usando SEH, podemos mezclar y comparar código (por ejemplo C#, C++ o VB.NET), sin peligro alguno y fácilmente. Como premio por seguir las reglas del SEH, .NET Framework garantiza que todos los errores serán expuestos y controlados convenientemente en los diferentes lenguajes.

El proceso de detección y gestión de excepciones en el código de C# es sencillo. Se deben identificar tres bloques de código cuando se trabaja con excepciones:

- El bloque de código que debe usar el procesamiento de excepciones
- Un bloque de código que se ejecuta si se encuentra una excepción mientras se procesa el primer bloque de código
- Un bloque de código opcional que se ejecuta después de que se procese la excepción

En C#, la generación de una excepción recibe el nombre de *iniciación de una excepción*. El proceso de informar de que se ha iniciado una excepción recibe el nombre de *capturar* una excepción. El fragmento de código que se ejecuta después de que se haya procesado la excepción es el bloque *finally*. En este capítulo veremos cómo se usan estos constructores en C#. También estudiaremos a los miembros de la jerarquía de las excepciones.

NOTA: Un debate largo y recurrente de la comunidad de usuarios de software orientado a objetos es si las excepciones deberían usarse en todos los errores (incluyendo los errores que uno espera que ocurran frecuentemente) o sólo para los errores graves (los conocidos como errores de excepción, que sólo ocurren cuando un recurso falla inesperadamente). El punto crucial de este debate es el relativamente importante encabezado necesario para iniciar y atrapar excepciones, encabezado que podemos evitar mediante el uso de otro método de control, como los códigos de devolución. La respuesta de .NET Framework a este conflicto es el uso del control de excepciones estructurado para todos los errores porque permite garantizar que todos los recursos se liberan adecuadamente cuando se produce un error. Esto es propio del consenso actual de este reñido debate. La investigación exhaus-

tiva (principalmente por parte de la comunidad de usuarios de C++) ha llegado a la conclusión de que evitar la pérdida de recursos sin excepciones es prácticamente imposible. Por supuesto, C# evita la pérdida de memoria con la recogida de elementos no utilizados, pero todavía necesitamos un mecanismo para evitar las pérdidas de recursos de los varios tipos de punteros de operaciones del sistema.

Como en todas las instrucciones de diseño, usar excepciones indiscriminadamente para todos los errores supone un uso excesivo de recursos. Cuando el error es local para un bloque de código, puede ser más apropiado usar códigos de devolución de error. Con frecuencia vemos este enfoque cuando se implementa una validación de formularios. Éste es un cambio aceptable porque los errores de validación suelen estar localizados en el formulario que recoge la entrada. En otras palabras, cuando ocurre un error de validación, se presenta en pantalla un mensaje y pedimos al usuario que vuelva a introducir correctamente la información requerida. Como el error y el código controlador están en el mismo bloque, controlar la pérdida de recursos es sencillo. Otro ejemplo es controlar una condición de fin de archivo cuando se está leyendo un archivo. Esta condición puede ser controlada fácilmente sin usar el encabezado de excepciones requerido. De nuevo, la condición de error se controla completamente dentro del bloque de código donde ocurre el error. Cuando observe que se realizan llamadas al código fuera del bloque de código donde ocurre el error, debe tender a procesar los errores usando SEH.

Cómo especificar el procesamiento de excepciones

La palabra clave de C# `try` especifica que hay un bloque de código que debe buscar cualquier excepción iniciada mientras se está ejecutando el código. Trabajar con la palabra clave `try` es sencillo. Use la palabra clave `try` seguida de una llave de apertura, de las instrucciones en las que se deben buscar excepciones mientras se ejecutan y termine con una llave de cierre:

```
try
{
    // coloque aquí las instrucciones
}
```

Si se inicia una instrucción mientras se está ejecutando cualquiera de las instrucciones del bloque `try`, se puede capturar la excepción en el código y ocuparse de ella adecuadamente.

Cómo capturar excepciones

Si se usa la palabra clave `try` para especificar que desea ser informado sobre las excepciones iniciadas, es necesario escribir código que atrape la excepción y se ocupe de los errores que envía el código.

Para indicar el código que debe ejecutarse cuando se atrape una excepción se usa la palabra clave de C# `catch` después de un bloque `try`. La palabra clave `catch` funciona de forma parecida a la palabra clave `try`.

Cómo usar la palabra clave `try`

La forma más simple del bloque de código `catch` atrapa todas las excepciones iniciadas por el código en el anterior bloque `try`. El bloque `catch` tiene la misma estructura que el bloque `try`, como se puede apreciar en el siguiente ejemplo:

```
try
{
    // coloque aquí las instrucciones
}
catch
{
    // coloque aquí las instrucciones
}
```

Las instrucciones del bloque `catch` se ejecutan si se inicia una excepción desde el bloque `try`.

Si ninguna de las instrucciones del bloque `try` inicia una excepción, entonces no se ejecuta nada del código del bloque `catch`.

Cómo atrapar clases específicas de excepciones

También se puede escribir un bloque `catch` que controle una clase específica de excepción iniciada por una de las instrucciones del bloque `try`. Esta forma del bloque de código `catch` usa la siguiente sintaxis:

- La palabra clave `catch`
- Un paréntesis de apertura
- La clase de excepción que desea controlar
- Un identificador de variable para la excepción
- Un paréntesis de cierre
- Una llave de apertura

- Las instrucciones que deben ejecutarse cuando se inicia una excepción del tipo especificado desde el anterior bloque `try`
- Una llave de cierre

Observe el siguiente código:

```
try
{
    // coloque aqui las instrucciones
}
catch(Exception thrownException)
{
    // coloque aqui las instrucciones
}
```

El bloque `catch` de este ejemplo atrapa excepciones de tipo `Exception` que inician el anterior bloque `try`. Define una variable del tipo `Exception` llamada `ThrownException`. La variable `ThrownException` puede usarse en el código del bloque `catch` para obtener más información sobre la excepción iniciada.

El código del bloque `try` puede iniciar diferentes clases de excepciones y queremos controlar cada una de las diferentes clases. C# permite especificar varios bloques `catch`, cada uno de los cuales controla una clase de error específica:

```
try
{
    // coloque aqui las instrucciones
}
catch(Exception ThrownException)
{
    // Bloque catch 1
}
catch(Exception ThrownException2)
{
    // Bloque catch 2
}
```

En este ejemplo, se revisa el código del bloque `try` en busca de excepciones iniciadas. Si el CLR descubre que el código del bloque `try` inicia alguna excepción, examina la clase de la excepción y ejecuta el bloque `catch` adecuado. Si la excepción iniciada es un objeto de clase `Exception`, se ejecuta el código del Bloque `catch 1`. Si la excepción iniciada es un objeto de alguna otra clase, no se ejecuta ninguno de los bloques.

También se puede añadir un bloque `catch` genérico a la lista de bloques de código `catch`, como en el siguiente ejemplo:

```
try
{
    // coloque aqui las instrucciones
```



```

}
catch(Exception ThrownException)
{
    // Bloque catch 1
}
catch
{
    // Bloque catch 2
}

```

En este caso, se revisa el código del bloque `try` en busca de excepciones. Si la excepción iniciada es un objeto de clase `Exception`, se ejecuta el código de `Bloque catch 1`. Si la excepción iniciada es un objeto de alguna otra clase, se ejecuta el código del bloque genérico `catch (Bloque catch 2)`.

Cómo liberar recursos después de una excepción

Los bloques `catch` pueden ir seguidos por bloque de código. Este bloque de código se ejecuta después de que se procese una excepción y cuando no ocurre ninguna excepción. Si se quiere ejecutar este código, se puede escribir un bloque `finally`. La palabra clave de C# `finally` especifica que hay un bloque de código que debe ejecutarse después de que se ejecute un bloque de código `try`. Un bloque de código `finally` tiene el mismo formato que los bloques `try`:

```

finally
{
    // coloque aqui las instrucciones
}

```

El bloque de código `finally` es un buen sitio donde liberar recursos a los que se había colocado en el método con anterioridad. Suponga, por ejemplo, que estamos escribiendo un método que abre tres archivos. Si encerramos el código de acceso a archivos en un bloque `try`, podremos atrapar excepciones relacionadas con la apertura, lectura o escritura de esos archivos. Sin embargo, al final del código, queremos cerrar los tres archivos, aunque se haya iniciado una excepción. Probablemente queramos colocar las instrucciones de cierre de archivos en un bloque `finally` y podremos estructurar el código como se indica a continuación:

```

try
{
    // abrir archivos
    // leer archivos
}
catch
{
    // atrapar excepciones
}

```

```

}
finally
{
    // cerrar archivos
}

```

El compilador de C# permite definir un bloque `finally` sin ningún bloque `catch`. Se puede escribir un bloque `finally` inmediatamente después de un bloque `try`.

La clase exception

Todas las excepciones iniciadas por .NET Framework son clases derivadas de la clase `System.Exception`. La tabla 16.1 describe algunos miembros útiles de esta clase.

Tabla 16.1. Miembros de la clase `System.Exception Class`

Miembro	Descripción
HelpLink	Un vínculo al archivo de ayuda que proporciona más información sobre la excepción
Message	El texto que se ha proporcionado, normalmente como parte del constructor de la excepción, para describir la condición del error
Source	El nombre de la aplicación u objeto que provocó la excepción
StackTrace	Una lista de las llamadas al método en la pila
TargetSite	El nombre del método que inició la excepción

Introducción a las excepciones definidas por .NET Framework

.NET Framework define varias excepciones que pueden iniciarse cuando se encuentran ciertos errores en el código de C# o en los métodos que se pueden invocar. Todas estas excepciones son excepciones estándar de .NET y pueden ser atrapadas usando un bloque `catch` de C#.

Cada una de las excepciones de .NET se define en el espacio de nombre `System` de .NET. Los siguientes apartados describen algunas de las excepciones más comunes. Estas excepciones son sólo una pequeña parte de todas las que hay definidas en la biblioteca de clases base de .NET Framework.

OutOfMemoryException

El CLR inicia la excepción `OutOfMemoryException` cuando agota su memoria. Si el código intenta crear un objeto usando el operador `new` y el CLR no dispone de suficiente memoria para ello, el CLR inicia la excepción `OutOfMemoryException`, mostrada en el listado 16.1.

Listado 16.1. Excepción `OutOfMemoryException`

```
using System;

class MainClass
{
    public static void Main()
    {
        int [] LargeArray;

        try
        {
            LargeArray = new int [2000000000];
        }
        catch(OutOfMemoryException)
        {
            Console.WriteLine("The CLR is out of memory.");
        }
    }
}
```

El código del listado 16.1 intenta asignar un espacio a una matriz de dos mil millones de números enteros. Dado que un número entero requiere cuatro bytes de memoria, se necesitan ocho mil millones de bytes para contener una matriz de este tamaño. Es bastante probable que su ordenador no disponga de esta cantidad de memoria y de que la asignación falle. El código encierra a la asignación en un bloque `try` y define un bloque `catch` para que controle cualquier excepción `OutOfMemoryException` iniciada por el CLR.

NOTA: El código del listado 16.1 no escribe un identificador para la excepción del bloque `catch`. Esta sintaxis (en la que se especifica la clase de una excepción pero no se le da nombre) es válida. Funciona perfectamente cuando se quiere atrapar una clase de excepciones pero no se necesita información del propio objeto específico de la excepción.

StackOverflowException

El CLR inicia la excepción `StackOverflowException` cuando agota el espacio de la pila. El CLR gestiona una estructura de datos llamada *stack*, que

registra los métodos que han sido llamados y el orden en el que fueron llamados. El CLR tiene una cantidad limitada de espacio de pila disponible y si se llena, se inicia la excepción. El listado 16.2 muestra la excepción `StackOverflowException`.

Listado 16.2. Excepción `StackOverflowException`

```
using System;

class MainClass
{
    public static void Main()
    {
        try
        {
            Recursive();
        }
        catch(StackOverflowException)
        {
            Console.WriteLine("The CLR is out of stack space.");
        }
    }

    public static void Recursive()
    {
        Recursive();
    }
}
```

El código del listado 16.2 implementa un método llamado `Recursive()`, que se llama a sí mismo antes de regresar. Este método es llamado por el método `Main()` y con el tiempo, hace que el CLR agote su espacio de pila porque el método `Recursive()` nunca regresa. El método `Main()` llama a `Recursive()`, que a su vez llama a `Recursive()`, que a su vez llama a `Recursive()` y así sucesivamente. A la larga, el CLR se quedará sin espacio de pila e iniciará la excepción `StackOverflowException`.

NullReferenceException

En este ejemplo, el compilador atrapa un intento de eliminar la referencia de un objeto null. El listado 16.3 muestra la excepción `NullReferenceException`.

Listado 16.3. Excepción `NullReferenceException`

```
using System;

class MyClass
{
    public int Value;
```

```

}

class MainClass
{
    public static void Main()
    {
        try
        {
            MyObject = new MyClass();
            MyObject = null;

            MyObject.Value = 123;

            // espere a que el usuario reconozca los resultados
            Console.WriteLine("Hit Enter to terminate...");
            Console.Read();
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("Cannot reference a null object.");

            //espere a que el usuario reconozca los resultados
            Console.Read();
        }
    }
}

```

El código del listado 16.3 declara una variable de objeto de tipo `MyClass` y asigna a la variable el valor de `null`. (Si no se usa la instrucción `new`, sino que sólo se declara una variable de objeto de tipo `MyClass`, el compilador emitirá el siguiente mensaje de error cuando se compile. "Uso de la variable local no asignada `MyObject`.") A continuación intentará trabajar con el campo público `Value` del objeto, lo que no está permitido porque no se puede hacer referencia a objetos `null`. El CLR atrapa este error e inicia la excepción `NullReferenceException`.

TypeInitializationException

El CLR inicia la excepción `TypeInitializationException` cuando una clase define un constructor estático y el constructor inicia una excepción. Si no hay bloques `catch` en el constructor para atrapar la excepción, el CLR inicia una excepción `TypeInitializationException`.

InvalidCastException

El CLR inicia la excepción `InvalidCastException` si falla una conversión explícita. Esto puede ocurrir en contextos de interfaz. El listado 16.4 muestra una excepción `InvalidCastException`.

Listado 16.4. Excepción `InvalidCastException`

```
using System;

class MainClass
{
    public static void Main()
    {
        try
        {
            MainClass MyObject = new MainClass();
            IFormattable Formattable;

            Formattable = (IFormattable)MyObject;

            // espere a que el usuario reconozca los resultados
            Console.WriteLine("Hit Enter to terminate...");
            Console.Read();
        }
        catch(InvalidCastException)
        {
            Console.WriteLine("MyObject does not implement the
                IFormattable interface.");

            // espere a que el usuario reconozca los resultados
            Console.Read();
        }
    }
}
```

El código del listado 16.4 usa un operador de conversión de tipo explícito para obtener una referencia a una interfaz .NET llamada `IFormattable`. Como la clase `MainClass` no implementa la interfaz `IFormattable`, la operación de conversión explícita falla y el CLR inicia la excepción `InvalidCastException`.

ArrayTypeMismatchException

El CLR inicia la excepción `ArrayTypeMismatchException` cuando el código intenta almacenar un elemento en una matriz cuyo tipo no coincide con el tipo del elemento.

IndexOutOfRangeException

El CLR inicia la excepción `IndexOutOfRangeException` cuando el código intenta almacenar un elemento en una matriz empleando un índice de elemento que está fuera del rango de la matriz. El listado 16.5 describe la excepción `IndexOutOfRangeException`.

Listado 16.5. Excepción `IndexOutOfRangeException`

```
using System;

class MainClass
{
    public static void Main()
    {
        try
        {
            int [] IntegerArray = new int [5];

            IntegerArray[10] = 123;

            // espere a que el usuario reconozca los resultados
            Console.WriteLine("Hit Enter to terminate...");
            Console.Read();
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("An invalid element index access was
attempted.");

            // espere a que el usuario reconozca los resultados
            Console.Read();
        }
    }
}
```

El código del listado 16.5 crea una matriz con cinco elementos y a continuación intenta asignar un valor al elemento 10 de la matriz. Como el índice 10 está fuera del rango de la matriz de números enteros, el CLR inicia la excepción `IndexOutOfRangeException`.

DivideByZeroException

El CLR inicia la excepción `DivideByZeroException` cuando el código intenta realizar una operación que da como resultado una división entre cero.

OverflowException

El CLR inicia la excepción `OverflowException` cuando una operación matemática guardada por el operador de C# `checked` da como resultado un desbordamiento. El listado 16.6 muestra la excepción `OverflowException`.

Listado 16.6. Excepción `OverflowException`

```
using System;

class MainClass
```

```

{
    public static void Main()
    {
        try
        {
            checked
            {
                int Integer1;
                int Integer2;
                int Sum;

                Integer1 = 2000000000;
                Integer2 = 2000000000;
                Sum = Integer1 + Integer2;
            }

            // espere a que el usuario reconozca los resultados
            Console.WriteLine("Hit Enter to terminate...");
            Console.Read();
        }
        catch(OverflowException)
        {
            Console.WriteLine("A mathematical operation caused an
overflow.");

            // espere a que el usuario reconozca los resultados
            Console.Read();
        }
    }
}

```

El código del listado 16.6 suma dos números enteros, cada uno con un valor de dos mil millones. El resultado, cuatro mil millones, se asigna a un tercer número entero.

El problema es que el resultado de la suma es mayor que el valor máximo que se puede asignar a un número entero de C# y se inicia una excepción de desbordamiento matemático.

Cómo trabajar con sus propias excepciones

Puede definir sus propias excepciones y usarlas en su código del mismo modo que haría con una excepción definida por .NET Framework. Esta consistencia en el diseño le permite escribir bloques `catch` que funcionen con cualquier excepción que pueda iniciar cualquier fragmento de código, tanto si el código pertenece a .NET Framework, como si pertenece a una de sus propias clases o a un ensamblado que se ejecuta en tiempo de ejecución.

Cómo definir sus propias excepciones

.NET Framework declara una clase llamada `System.Exception`, que sirve como clase base para todas las excepciones de .NET Framework. Las clases predefinidas del entorno común de ejecución se derivan de `System.Exception`, que a su vez deriva de `System.Exception`. La excepción a esta regla son las excepciones `DivideByZeroException`, `NotFiniteNumberException` y `OverflowException`, que derivan de una clase llamada `System.ArithmeticException`, que deriva de `System.SystemException`. Cualquier clase de excepción que defina debe derivar de `System.ApplicationException`, que también deriva de `System.Exception`.

La clase `System.Exception` contiene cuatro propiedades de sólo lectura que el código de los bloques `catch` puede usar para obtener más información sobre la excepción que se ha iniciado:

- La propiedad `Message` contiene una descripción de la causa de la excepción.
- La propiedad `InnerException` contiene la excepción que ha provocado que se inicie la excepción actual. Esta propiedad puede ser `null`, lo que indicaría que no hay ninguna excepción interior disponible. Si `InnerException` no es `null`, hace referencia al objeto de excepción que se ha iniciado y que provocó que se iniciase la excepción actual. Un bloque `catch` puede atrapar una excepción e iniciar otra diferente. En ese caso, la propiedad `InnerException` puede contener una referencia al objeto de excepción original atrapado por el bloque `catch`.
- La propiedad `StackTrace` contiene una cadena que muestra la pila de llamadas de método que estaba en vías de ejecución cuando se inició la excepción. En última instancia, este rastro de pila puede contener todo el recorrido hasta la llamada al método `Main()` de la aplicación del CLR.
- La propiedad `TargetSite` contiene el método que ha iniciado la excepción.

Algunas de estas propiedades pueden especificarse en uno de los constructores de la clase `System.Exception`:

```
public Exception(string message);  
public Exception(string message, Exception innerException);
```

Las excepciones definidas por el usuario pueden llamar al constructor de clase base en su constructor de modo que se pueden asignar valores a las propiedades, como muestra el siguiente código:

```
using System;

class MyException : ApplicationException
{
    public MyException() : base("This is my exception message.")
    {
    }
}

```

Este código define una clase llamada `MyException`, que deriva de la clase `ApplicationException`. Su constructor usa la palabra clave `base` para llamar al constructor de la clase base. La propiedad `Message` de la clase recibe el valor `This is my exception message`.

Cómo iniciar sus excepciones

Puede iniciar sus propias excepciones mediante la palabra clave de C# `throw`. La palabra clave `throw` debe ir seguida por una expresión que evalúa un objeto de clase `System.Exception` o una clase derivada de `System.Exception`. Observe el código del listado 16.7.

Listado 16.7. Cómo iniciar sus propias excepciones

```
using System;

class MyException : ApplicationException
{
    public MyException() : base("This is my exception message.")
    {
    }
}

class MainClass
{
    public static void Main()
    {
        try
        {
            MainClass MyObject = new MainClass();

            MyObject.ThrowException();

            // espere a que el usuario reconozca los resultados
            Console.WriteLine("Hit Enter to terminate...");
            Console.Read();
        }
        catch(MyException CaughtException)
        {
            Console.WriteLine(CaughtException.Message);
        }
    }
}

```

```

        // espere a que el usuario reconozca los resultados
        Console.Read();
    }
}

public void ThrowException()
{
    throw new MyException();
}
}

```

El código del listado 16-7 declara una clase `new` llamada `MyException`, que deriva de la clase base `ApplicationException` definida por .NET Framework.

La clase `MainClass` contiene un método llamado `ThrowException`, que inicia un nuevo objeto de tipo `MyException`. El método es invocado por el método `Main()`, que encierra la llamada en un bloque `try`. El método `Main()` también contiene un bloque `catch`, cuya implementación escribe el mensaje de la excepción en la consola. Como el mensaje se estableció cuando se construyó el objeto de la clase `MyException`, está disponible y se puede escribir. Si se compila y ejecuta el listado 16.7 se escribe lo siguiente en la consola:

```
This is my exception message.
```

Cómo usar excepciones en constructores y propiedades

Algunos constructores de C# contienen código que se puede ejecutar, pero no pueden devolver un valor que indique el éxito o fracaso del código que se está ejecutando. Los constructores de clase y los descriptores de acceso de propiedades `set` son un ejemplo claro. Iniciar excepciones es un buen modo de informar de errores de bloques de código como éstos.

En un capítulo anterior examinamos una clase que implementaba un punto en la pantalla. La clase tenía propiedades que representaban las coordenadas x e y del punto y los descriptores de acceso `set` para las propiedades garantizaban que el valor era válido antes de que se almacenara realmente. El problema con el código del listado 9.1 es que no hay un informe de errores en caso de que se proporcione un valor que esté fuera de los límites permitidos. El listado 16.8 es una versión mejorada del listado 9.1 porque añade control de excepciones para informar de coordenadas que están fuera de los límites permitidos.

Listado 16.8. Cómo iniciar excepciones desde descriptores de acceso de propiedades

```

using System;

public class CoordinateOutOfRangeException :
    ApplicationException

```

```

{
    public CoordinateOutOfRangeException() : base("The supplied
coordinate is out of range.")
    {
    }
}

public class Point
{
    private int XCoordinate;
    private int YCoordinate;

    public int X
    {
        get
        {
            return XCoordinate;
        }
        set
        {
            if((value >= 0) && (value < 640))
                XCoordinate = value;
            else
                throw new CoordinateOutOfRangeException();
        }
    }

    public int Y
    {
        get
        {
            return YCoordinate;
        }
        set
        {
            if((value >= 0) && (value < 480))
                YCoordinate = value;
            else
                throw new CoordinateOutOfRangeException();
        }
    }

    public static void Main()
    {
        Point MyPoint = new Point();

        try
        {
            MyPoint.X = 100;
            MyPoint.Y = 200;
            Console.WriteLine("{0}, {1}", MyPoint.X, MyPoint.Y);
            MyPoint.X = 1500;
            MyPoint.Y = 600;
        }
    }
}

```

```

        Console.WriteLine("{0}, {1})", MyPoint.X, MyPoint.Y);

        // espere a que el usuario reconozca los resultados
        Console.WriteLine("Hit Enter to terminate...");
        Console.Pead();
    }
    catch(CoordinateOutOfRangeException CaughtException)
    {
        Console.WriteLine(CaughtException.Message);

        // espere a que el usuario reconozca los resultados
        Console.Pead();
    }
    catch
    {
        Console.WriteLine("An unexpected exception was
                            caught.");

        // espere a que el usuario reconozca los resultados
        Console.Pead();
    }
}
}

```

El código del listado 16.8 comprueba el valor de los descriptores de acceso de la propiedad `set` para garantizar que el valor proporcionado está dentro de los límites válidos. En caso contrario, se inicia una excepción. La asignación del primer punto tiene éxito ya que los dos valores están dentro de los límites válidos. Sin embargo, el segundo punto no tiene éxito, ya que la coordenada `x` está fuera de los límites válidos. Este valor fuera de los límites válidos hace que se inicie un objeto de clase `CoordinateOutOfRangeException`.

Si se compila y ejecuta el listado 16.8 se escribe lo siguiente en la consola:

```

(100, 200)
The supplied coordinate is out of range.

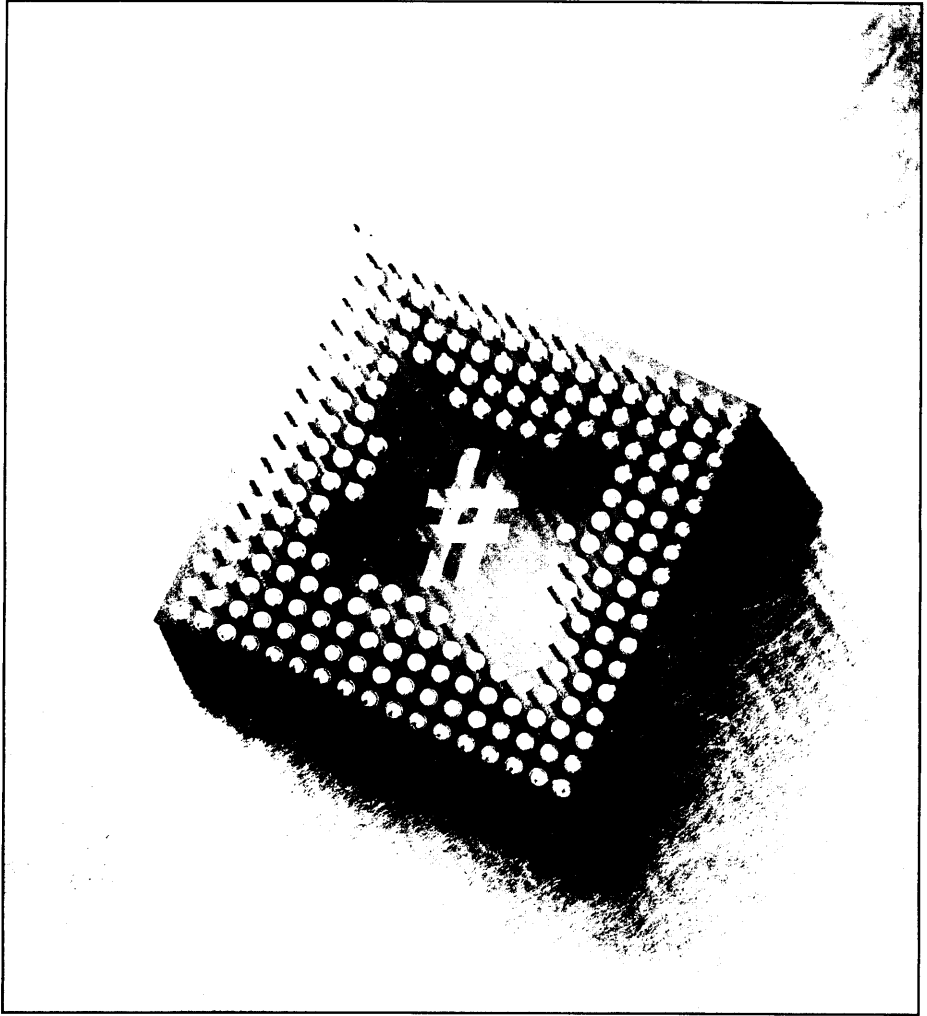
```

Resumen

.NET Framework usa las excepciones para informar de diferentes errores a las aplicaciones .NET. El lenguaje C# admite perfectamente el tratamiento de las excepciones y permite al usuario definir sus propias excepciones, además de trabajar con las excepciones definidas por .NET Framework. El código de C# puede iniciar y atrapar excepciones. También puede atrapar excepciones iniciadas por .NET Framework.

La ventaja de usar excepciones reside en que no es necesario comprobar cada llamada de método en busca de un error. Se puede encerrar un grupo de llamadas de método en un bloque `try` y se puede escribir código como si cada llamada de

método del bloque tuviera éxito. Esto hace que el código del bloque `try` sea mucho más limpio porque no necesita ninguna comprobación de errores entre líneas. Cualquier excepción iniciada desde el código del bloque `try` es gestionada en un bloque `catch`.



17 Cómo trabajar con atributos

Los capítulos anteriores estaban dedicados a las palabras clave que definen el comportamiento de una clase y sus miembros. Por ejemplo, las palabras clave `public`, `private`, `protected` e `internal` definen la accesibilidad de la declaración para otras clases del código. Estos modificadores están implementados por palabras clave predefinidas cuyo significado está integrado en el lenguaje C# y no puede ser cambiado.

C# también permite mejorar las declaraciones de clase y de miembro de clase mediante información que es interpretada por otras clases de C# en tiempo real. Esta información se especifica usando un constructor llamado *atributo*. Los atributos permiten incluir directivas en las clases y en sus miembros. El comportamiento del atributo se define por el código que escribimos o por el código que proporciona .NET Framework. Los atributos permiten ampliar el lenguaje C# mediante la escritura de clases de atributo que mejoran el comportamiento de otras clases cuando se ejecuta el código, aunque escribamos la clase de implementación de atributo antes de que los otros usuarios apliquen el atributo a sus propias clases.

Al compilar aplicaciones, la información del atributo que se añade es enviada a los metadatos del ensamblado, lo que permite que otras aplicaciones o herramientas vean que se está usando el atributo. Mediante el desensamblador IL (ILDASM) o las clases del espacio de nombres `System.Reflection` se pue-

de comprobar fácilmente qué atributos se han añadido a las secciones de código y se puede determinar si son útiles. En C#, se pueden usar dos tipos de atributo: los que están integrados en el lenguaje y los atributos personalizados, que podemos crear. En la primera parte de este capítulo, aprenderemos a usar atributos y examinaremos algunos de los atributos integrados que nos ofrece C#. En la segunda parte, aprenderemos a escribir atributos personalizados y el modo en que las aplicaciones pueden sacar partido de estos atributos.

Atributos

C# permite que los atributos se antepongan a los siguientes constructores de C#:

- Clases.
- Miembros de clase, incluyendo constantes, campos, métodos, propiedades, eventos, indizadores, sobrecargas de operador, constructores y destructores.
- Estructuras.
- Interfaces.
- Miembros de interfaces, incluyendo métodos, propiedades, eventos e indizadores.
- Enumeraciones y miembros de enumeraciones.
- Delegados.

Para especificar un atributo en el código se escribe su nombre entre corchetes. La especificación de atributo debe aparecer antes de la declaración en la que se debe aplicar el atributo. Los atributos más simples pueden tener este aspecto:

```
[MyAttribute]
```

Un buen ejemplo de un atributo simple es el modelo de subprocesos que se usa para crear una aplicación de consola en C#. Observe en el siguiente fragmento de código el atributo `[STAThread]` aplicado a la función `Main()`. Este atributo indica al compilador que la función `Main()` debe introducir un apartamento de un único subproceso (STA) COM antes de que se ejecute algún código basado en COM.

```
// <summary>  
// El punto de entrada principal de la aplicacion.  
// </summary>  
[STAThread]  
static void Main(string[] args)
```

```

}
//
// TODO: Añada el código para iniciar la aplicación aquí
//
}

```

También se puede anteponer a un atributo un modificador que defina el elemento de C# al que se aplica el atributo. El modificador de atributo se escribe antes del nombre del atributo y va seguido de dos puntos. Esto recibe el nombre de *enlazar* un atributo. La tabla 17.1 enumera los tipos de declaraciones y los elementos a los que hace referencia para los atributos específicos. Los elementos a los que las clases de atributo hacen referencia están predefinidos: por ejemplo, si una clase .NET contiene una clase de atributo que sólo hace referencia a enumeraciones, el atributo sólo puede ser usado para enumeraciones y no puede aplicarse a otros constructores del código C#, como clases y estructuras. Más adelante aprenderá a especificar los destinos de los atributos para sus clases de atributo personalizadas.

Tabla 17.1. Listado de destinos de atributos

Declaración	Destino
Assembly	Assembly
Module	Module
Class	Type
Struct	Type
Interface	Type
Enum	Type
Delegate	Type (por defecto) o Return Value
Method	Method (por defecto) o Return Value
Parameter	Param
Field	Field
Property – Indexer	Property
Property – Get Accessor	Method (por defecto) o Return Value
Property – Set Accessor	Method (por defecto), Param or Return Value
Event – Field	Event (por defecto), Field o Method
Event – Property	Event (por defecto), Property
Event – Add	Method (por defecto) o Param
Event – Remove	Method (por defecto) o Param

Por ejemplo, para enlazar explícitamente un atributo a un método, se debe escribir algo parecido a esto:

```
[method:MyAttribute]
int MyMethod()
{
}
```

Los modificadores de atributo son útiles en las situaciones en las que su enlace puede ser ambiguo, como muestra el siguiente ejemplo:

```
[MyAttribute]
int MyMethod()
{
}
```

En realidad, este ejemplo no es muy aclaratorio. ¿Se aplica el atributo `MyAttribute` al método o a su tipo devuelto? Si se especifica explícitamente el enlace, como se muestra en el anterior ejemplo, se indica al compilador de C# que el atributo se aplica a todo el método. En el ejemplo en el que el atributo `[STAThread]` se aplica a la función `Main()` cuando se crea una aplicación de consola, se puede hacer la siguiente modificación para hacer más evidente el enlace:

```
/// <summary>
/// El punto de entrada principal de la aplicación.
/// </summary>
[method: STAThread]
static void Main(string[] args)
{
    //
    // TODO: Añada el código para iniciar la aplicación aquí
    //
}
```

Algunos atributos están contruidos para aceptar parámetros. Los parámetros de atributo siguen al nombre del atributo y están entre paréntesis. Los paréntesis están a su vez entre corchetes. Un atributo con un parámetro puede tener este aspecto:

```
[MyAttribute(Parameter)]
```

Ahora que tiene un conocimiento básico de la sintaxis de los atributos, podemos examinar las clases de atributo integradas que ofrece .NET. Observe que las clases de atributos funcionan en todos los lenguajes, de modo que, aunque escriba atributos para los tipos de C#, la información de atributo puede ser usada por Visual Basic .NET, JScript .NET y todos los lenguajes orientados al entorno común de ejecución (CLR). El objetivo del uso de atributos es aumentar la funcionalidad del lenguaje.

Cómo trabajar con atributos de .NET Framework

.NET Framework proporciona cientos de atributos predefinidos integrados. No son fáciles de encontrar, ya que el SDK no proporciona una lista con cada atributo en orden alfabético. Dependiendo de las clases que estemos usando, los atributos pueden derivar de la clase `System.Attribute` y estos atributos pueden ser objetos específicos. Por ejemplo, cuando se trabaja con la función de .NET Framework que permite que el código .NET interactúe con el código heredado COM (lo que se conoce como interoperabilidad COM), se pueden emplear sobre los modificadores más de 20 clases de atributos, desde el atributo `ComAliasName` hasta el atributo `TypeLibType`. El siguiente código muestra el atributo `DllImportAttribute` y le da una idea de cómo llamar a métodos externos en las DLL de Win32 desde C#.

```
namespace System.Runtime.InteropServices
{
    [AttributeUsage(AttributeTargets.Method)]
    public class DllImportAttribute: System.Attribute
    {
        public DllImportAttribute(string dllName) {...}
        public CallingConvention CallingConvention;
        public CharSet CharSet;
        public string EntryPoint;
        public bool ExactSpelling;
        public bool PreserveSig;
        public bool SetLastError;
        public string Value { get {...} }
    }
}
```

Sin atributos, no sería posible informar al compilador de C# del modo en el que pretendemos usar un método específico en una DLL externa y si el lenguaje C# incluyese esta funcionalidad en el lenguaje base, no sería lo suficientemente genérica como para poder ejecutarse en otras plataformas. Con la posibilidad de llamar a componentes Win32 a través de todas las plataformas, se obtiene control sobre qué propiedades usar, en su caso, cuando se llama a métodos externos.

Como .NET Framework cuenta con tal cantidad de clases de atributos, es imposible describir cada una de ellas en un solo capítulo. Además, como las clases de atributo son específicas de las clases en las que se definen, sólo son útiles en el contexto de esas clases. A medida que codifique aplicaciones y se vaya familiarizando con los espacios de nombres de .NET Framework para los que está codificando, las clases de atributo asociadas a los espacios de nombre se harán más transparentes. Algunas clases de atributo reservadas pueden funcionar por sí mismas y afectar directamente al lenguaje C#. Las clases `System.Obsolete-`

`Attribute`, `System.SerializableAttribute` y `System.ConditionalAttribute` son clases de atributo que pueden usarse independientemente y que afectan directamente al resultado del código.

NOTA: En .NET Framework, las clases de atributo tienen alias, por lo que, cuando se usan clases de atributo, es normal ver el nombre de la clase de atributo sin la palabra "Attribute" a continuación. El sufijo está implícito, de modo que la forma corta no produce un error. Por ejemplo, `ObsoleteAttribute` puede usarse como `Obsolete`, ya que los atributos están entre llaves, lo que hace evidente que son atributos y no algún otro tipo de modificador.

Observemos algunas de las muchas clases de atributo disponibles en .NET Framework. De esta forma, se acostumbrará al modo de trabajar de estas clases y al modo de aplicar estos atributos en el código de C#.

System.Diagnostics.ConditionalAttribute

El atributo `Conditional` es el alias de `System.Diagnostics.ConditionalAttribute`, que sólo puede aplicarse a declaraciones de método de clase. Especifica que el método sólo debe ser incluido como una parte de la clase si el compilador de C# define el símbolo que aparece como parámetro del atributo. El listado 17.1 muestra el funcionamiento del atributo `Conditional`.

Listado 17.1. Cómo trabajar con el atributo `Conditional`

```
using System;
using System.Diagnostics;

public class TestClass
{
    public void Method1()
    {
        Console.WriteLine("Hello from Method1!");
    }

    [Conditional("DEBUG")]
    public void Method2()
    {
        Console.WriteLine("Hello from Method2!");
    }

    public void Method3()
    {
        Console.WriteLine("Hello from Method3!");
    }
}
```

```

}

class MainClass
{
    public static void Main()
    {
        TestClass MyTestClass = new TestClass();

        MyTestClass.Method1();
        MyTestClass.Method2();
        MyTestClass.Method3();
    }
}

```

NOTA: Recuerde hacer referencia al espacio de nombre System.Diagnostics en el código de modo que no tenga que usar el espacio de nombre completo al usar la clase de atributo Conditional y el compilador de C# pueda encontrar la implementación de la clase.

El listado 17.1 declara dos clases: `TestClass` y `MainClass`. La clase `TestClass` contiene tres métodos: `Method1()`, `Method2()` y `Method3()`. Las clases `Method1()` y `Method3()` se implementan sin atributos, pero `Method2()` usa el atributo `Conditional` con un parámetro llamado `DEBUG`. Esto significa que el método `Method2()` es una parte de la clase sólo cuando el compilador de C# construye la clase con el símbolo `DEBUG` definido. Si el compilador de C# construye la clase sin haber definido el símbolo `DEBUG`, el método no se incluye como una parte de la clase y se pasa por alto cualquier llamada al método. La clase `MainClass` implementa el método `Main()` de la aplicación, que crea un objeto de tipo `TestClass` y llama a los tres métodos de la clase. El resultado del listado 17.1 cambia dependiendo del modo en que es compilado el código. En primer lugar, intente compilar el listado 17.1 con el símbolo `DEBUG` definido. Puede usar el argumento de línea de comando del compilador de C# `/D` para definir símbolos para el compilador:

```
csc /D:DEBUG Listing17-1.cs
```

Cuando el código del listado 17.1 se compila mientras el símbolo `DEBUG` está definido, el método `Method2()` de la clase `TestClass` se incluye en la construcción y al ejecutar la aplicación se escribe lo siguiente en la consola:

```

Hello from Method1!
Hello from Method2!
Hello from Method3!

```

Ahora intente compilar el listado 17.1 sin el símbolo `DEBUG` definido:

```
csc Listing17-1.cs
```

Cuando se compila el código del listado 17.1 mientras el símbolo `DEBUG` no está definido, el método `Method2()` de la clase `TestClass` no se incluye en la construcción y se pasa por alto la llamada al método `Method2()` realizada en el método `Main()`. Si se crea el código del listado 17.1 sin definir el símbolo `DEBUG` se genera código que escribe lo siguiente en la consola al ser ejecutado:

```
Hello from Method1!  
Hello from Method3!
```

Como puede ver, el atributo `Conditional` es eficaz y útil. Antes de empezar a usar esta clase, preste atención a las siguientes reglas:

- El método marcado con el atributo `Conditional` debe ser un método de una clase.
- El método marcado con el atributo `Conditional` no debe ser un método `override`.
- El método marcado con el atributo `Conditional` debe tener un tipo devuelto `void`.
- Aunque el método marcado con el atributo `Conditional` no debe estar marcado con el modificador `override`, puede estar marcado con el modificador `virtual`. Los reemplazos de estos métodos son implícitamente condicionales y no deben estar marcados explícitamente con un atributo `Conditional`.
- El método marcado con el atributo `Conditional` no debe ser una implementación de un método de interfaz; en caso contrario, se producirá un error en tiempo de compilación.

System.SerializableAttribute class

El atributo `Serializable` es el alias de la clase `System.SerializableAttribute`, que puede ser aplicado a clases. Indica a .NET Framework que los miembros de la clase pueden ser serializados a y desde un medio de almacenamiento, como un disco duro. El uso de este atributo hace que no resulte necesario agregar la función de estado en las clases para que su almacenamiento en el disco y su posterior recuperación. Cuando se serializan tipos, todos los datos de la clase marcada como `Serializable` se guardan en el estado en el que se encuentran cuando el dato es persistente. Si hay tipos dentro de la clase que no quiere que sean persistentes, puede marcarlos con el atributo `NonSerialized`, que es el alias de la clase `System.NonSerializableAttribute`. En el siguiente fragmento de código, los datos de la cadena `password` marcados como `NonSerialized` no son persistentes para el archivo o flujo para el que se escriben los datos de clase:

```

[Serializable()]
public class Users{

    public string username;
    public string emailaddress;
    public string phonenumber;

    // Añada un campo que no vaya a ser persistente

    [NonSerialized()] public string password;

    public FillData() {

        username = "admin";
        password = "password";
        emailaddress = "billg@microsoft.com";
        phonenumber = "555-1212";
    }
}

```

Para mostrar un ejemplo de serialización completo, el listado 17.2 vuelve a la clase `Point2D` con la que ya hemos trabajado. La clase está marcada con el atributo `Serializable`, lo que significa que puede ser guardada y leída desde un flujo de datos.

Listado 17.2. Cómo trabajar con el atributo `Serializable`

```

using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class Point2D
{
    public int X;
    public int Y;
}

class MyMainClass
{
    public static void Main()
    {
        Point2D My2DPoint = new Point2D();

        My2DPoint.X = 100;
        My2DPoint.Y = 200;

        Stream WriteStream = File.Create("Point2D.bin");
        BinaryFormatter BinaryWrite = new BinaryFormatter();
        BinaryWrite.Serialize(WriteStream, My2DPoint);
        WriteStream.Close();

        Point2D ANewPoint = new Point2D();
    }
}

```



```

        Console.WriteLine("New Point Before Deserialization: ({0},
{1})", ANewPoint.X, ANewPoint.Y);
        Stream ReadStream = File.OpenRead("Point2D.bin");
        BinaryFormatter BinaryRead = new BinaryFormatter();
        ANewPoint = (Point2D)BinaryRead.Deserialize(ReadStream);
        ReadStream.Close();
        Console.WriteLine("New Point After Deserialization: ({0},
{1})", ANewPoint.X, ANewPoint.Y);
    }
}

```

El código del listado 17.2 crea un nuevo objeto `Point2D` y le otorga las coordenadas (100, 200). A continuación serializa la clase en un archivo llamado `Point2D.bin`.

El código crea entonces un nuevo punto y deserializa los contenidos del archivo `Point2D.bin` en el nuevo objeto `Point2D`. El proceso de deserialización lee el archivo `Point2D.bin` y asigna los valores que se encontraban en el archivo binario a los valores del objeto. Si se ejecuta el código del listado 17.2 se escribe lo siguiente en la consola:

```

New Point Before Deserialization: (0, 0)
New Point After Deserialization: (100, 200)

```

Cuando se crea el nuevo objeto `Point2D`, sus miembros se inicializan con sus valores por defecto de 0. El proceso de deserialización, que asigna los valores de acuerdo con los datos almacenados en el archivo `Point2D.bin`, cambia los valores. El listado 17.2 emplea dos clases de .NET Framework en su proceso de serialización. La clase `Stream` se encuentra en el espacio de nombre `System.IO` y gestiona el acceso a los flujos de datos, incluyendo los archivos de disco. La clase `BinaryFormatter` se encuentra en el espacio de nombre `System.Runtime.Serialization.Formatters.Binary` y gestiona la serialización de datos a una representación binaria. .NET Framework incluye otros formateadores que pueden usarse para representar datos serializados en otros formatos. Por ejemplo, la clase `SoapFormatter` da a los datos serializados un formato adecuado para una llamada XML SOAP.

NOTA: La clase `BinaryFormatter` es una patente de .NET Framework. Si tiene pensado que su destino sean otros sistemas que pueden no ser compatibles con el formato binario, considere usar la clase `SoapFormatter` para que persistan los datos de un formato XML que es compatible con otros sistemas.

System.ObsoleteAttribute class

El atributo `Obsolete` puede ser aplicado a cualquier tipo de C# excepto a ensamblados, módulos, parámetros y valores devueltos. El atributo `Obsolete`

permite definir fragmentos de código que se van a reemplazar o que ya no son válidos. Las propiedades `Message` e `IsError` de la clase `Obsolete` otorgan el control del modo en el que el compilador controla los tipos marcados con el atributo `Obsolete`. Al asignar a la propiedad `IsError` el valor `True`, el compilador produce un error y el mensaje de error es la propiedad de cadena asignada a la propiedad `Message`. El valor por defecto de la propiedad `IsError` es `False`, lo que hace que se produzca un aviso cuando se compila el código. En el siguiente código, el método `HelloWorld` está marcado como `Obsolete`.

```
using System;
public class RunThis
{
    public static void Main()
    {
        // Esto genera un aviso de tiempo de compilacion.
        Console.WriteLine>HelloWorld());
        Console.ReadLine();
    }

    // Marca HelloWord como Obsolete
    [Obsolete("Next version uses Hello Universe")]
    public static string>HelloWorld()
    {
        return ("HelloWorld");
    }
}
```

La figura 17.1 muestra la lista de tareas de los avisos producidos al compilar el código anterior.

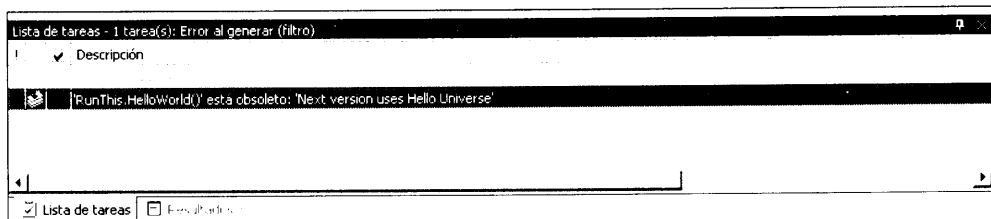


Figura 17.1. Aviso producido al emplear el atributo `Obsolete`

Si quiere asegurarse de que se produce un error y no sólo un mensaje de aviso, puede modificar el código marcado asignando a la propiedad `IsError` el valor `true` y la clase no se compilará. Si modifica el atributo `Obsolete` del código anterior con la siguiente línea, se produce un error:

```
[Obsolete("Next version uses Hello Universe", true)]
```

Como puede ver, el uso del atributo `Obsolete` permite mantener el código existente mientras nos aseguramos de que los programadores no están usando tipos desfasados.

Cómo escribir sus propias clases de atributo

.NET Framework parte con un buen número de clases de atributo que pueden usarse para diferentes propósitos. Sin embargo, podría necesitar un atributo que cumpliera una función no incluida en .NET Framework. Por ejemplo, podría desear disponer de un atributo de revisión de código que etiquetara una clase con la fecha de la última vez que el código de esa clase fue revisado por sus compañeros. En ese caso, necesitará definir sus propios atributos y hacer que funcionen como cualquier atributo incluido en .NET Framework. Por suerte, .NET Framework admite perfectamente la construcción de nuevas clases de atributos. En esta sección aprenderemos cómo el código .NET desarrolla y usa las nuevas clases de atributos.

Puede escribir sus propias clases de atributos y usarlas en su código del mismo modo que usaría un atributo procedente de .NET Framework. Las clases de atributo personales funcionan como clases normales; tienen propiedades y métodos que permiten al usuario del atributo asignar y recuperar datos.

Los atributos se implementan con clases de atributo. Las clases de atributo derivan de una clase del espacio de nombre `System` de .NET llamada `Attribute`. Por norma, las clases de atributo llevan antepuesta la palabra `Attribute`:

```
public class CodeAuthorAttribute : Attribute
{
}
```

Esta clase define un atributo llamado `CodeAuthorAttribute`. Este nombre de atributo puede usarse como un atributo una vez que se ha definido la clase. Si el nombre de atributo termina con el sufijo `Attribute`, el nombre de atributo puede ser usado entre corchetes o sin el sufijo:

```
[CodeAuthorAttribute]
[CodeAuthor]
```

Estos dos atributos hacen referencia a la clase `CodeAuthorAttribute`. Tras definir una clase de atributo, se usa como cualquier otra clase de atributo .NET.

Cómo restringir el uso de atributos

Las clases de atributo pueden, a su vez, usar atributos. El ejemplo más común es un atributo llamado `AttributeUsage`. El atributo `AttributeUsage` contiene un parámetro que especifica dónde puede usarse un atributo. Algunos atributos pueden no tener sentido en todos los constructores de C# válidos. Por ejemplo, el atributo `Obsolete` tratado con anterioridad sólo tiene sentido en métodos. No es lógico marcar una sola variable como obsoleta, de modo que el

atributo `Obsolete` solamente debe aplicarse a métodos y no a otros constructores de C#. La clase de atributo `AttributeUsage` contiene una enumeración pública llamada `AttributeTargets`, cuyos miembros aparecen en la tabla 17.1.

Estos miembros `AttributeTargets` pueden aparecer juntos en una expresión `OR` y ser usados como parámetros del atributo `AttributeUsage` para especificar que la clase de atributo define un atributo que sólo puede usarse en determinados contextos, como muestra el siguiente ejemplo:

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Struct)]
public class CodeAuthorAttribute : Attribute
{
}
```

Este constructor declara una clase llamada `CodeAuthorAttribute` y especifica que el atributo sólo puede ser usado con clases y estructuras.

El compilador de C# le fuerza a usar el atributo para asegurarse de que se emplea de acuerdo con los valores de la enumeración de `AttributeTargets` especificados en el atributo `AttributeUsage`. Si usa un atributo en una expresión que no está permitida en la definición del atributo, el compilador emitirá un error.

Por ejemplo, suponga que escribe un atributo llamado `Name` y sólo usa la enumeración `AttributeTargets.Class` como parámetro del atributo `AttributeUsage`:

```
[AttributeUsage(AttributeTargets.Class)]
public class NameAttribute : Attribute
{
}
```

Si a continuación intenta aplicar el atributo `Name` a algo que no sea una clase, el compilador emitirá un mensaje de error parecido al siguiente:

```
error CS0592: El atributo "Name" no es válido en este tipo de declaración. Sólo es válido en declaraciones "class".
```

Cómo permitir múltiples valores de atributo

También se puede usar el atributo `AttributeUsage` para especificar si una clase permite que varias instancias de un atributo sean usadas en un fragmento de código C# en particular. Esto se especifica mediante un parámetro de atributo llamado `AllowMultiple`. Si el valor de `AllowMultiple` es `True`, se pueden usar varias instancias del atributo en un elemento particular de C#. Si `AllowMultiple` recibe el valor `False`, sólo se puede usar una instancia en cualquier elemento particular de C# (aunque sigue estando permitido aplicar el atributo a más de un constructor de C#):

```
[AttributeUsage(AttributeTargets.class, AllowMultiple = true)]
public class NameAttribute : Attribute
{
    public NameAttribute(string Name)
    {
    }
}
```

El uso de varios atributos permite asignar varios valores a un constructor de C# usando un solo atributo. El siguiente constructor marca el atributo `Name` como un atributo de varios usos y permite a los programadores usar el atributo más de una vez en un solo elemento de C#:

```
[Name("Jeff Ferguson")]
[Name("Jeff Ferguson's Assistant")]
public class MyClass
{
}
```

Los atributos de varios usos también pueden aparecer en un solo conjunto de corchetes, separados por una coma:

```
[Name("Jeff Ferguson"), Name("Jeff Ferguson's Assistant")]
public class MyClass
{
}
```

Si no especifica un valor para el parámetro `AllowMultiple`, no se permite el uso variado.

Cómo asignar parámetros de atributo

Sus clases de atributo pueden aceptar parámetros, que aparecen entre paréntesis después del nombre de atributo. En el ejemplo anterior, el atributo `Name` recibe una cadena que da nombre al creador de código como parámetro. Algunos atributos necesitan parámetros para asociar los datos al atributo, como la cadena de nombre en el atributo `Name` mostrado anteriormente.

Los valores de los parámetros se pasan al constructor de la clase de atributo y la clase de atributo debe implementar un constructor que pueda recibir los parámetros:

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Struct)]
public class CodeAuthorAttribute : Attribute
{
    public CodeAuthorAttribute(string Name)
    {
    }
}
```

Este atributo necesita que se suministre un parámetro de cadena cada vez que es usado:

```
[CodeAuthor("Jeff Ferguson")]
```

Debe suministrar los parámetros especificados en el constructor de la clase cuando se usa el atributo. En caso contrario, obtendrá un error del compilador:

```
error CS1501: Ninguna sobrecarga para el metodo  
'CodeAuthorAttribute' adquiere '0' argumentos
```

Los parámetros proporcionados al constructor de la clase de atributo reciben el nombre de *parámetros posicionales*. Los parámetros posicionales asocian los datos de parámetro con sus nombres de parámetro basándose en la posición de los datos en la lista de parámetros. Por ejemplo, el elemento de datos del segundo parámetro está asociado a la variable del segundo parámetro especificada en la lista de parámetros de la declaración de la función. También se pueden proporcionar parámetros con nombre, que son almacenados por las propiedades implementadas en la clase de atributo. Los parámetros con nombre se especifican con el nombre de la propiedad, un signo igual y el valor de la propiedad. Los parámetros con nombre se asocian a datos de parámetros con el nombre del parámetro basado en el nombre del parámetro que aparece antes del valor. Dado que la asociación entre un nombre de variable y su valor se especifica mediante el nombre del parámetro y no mediante la posición del valor en la lista de parámetros, los parámetros con nombre pueden aparecer en cualquier orden.

Suponga que añade un parámetro con nombre llamado `Date` al atributo `CodeAuthorAttribute`. Esto significa que la clase puede admitir una propiedad llamada `Date` cuyo valor puede asignarse en la definición del atributo:

```
[AttributeUsage(AttributeTargets.Class |  
AttributeTargets.Struct)]  
public class CodeAuthorAttribute : Attribute  
{  
    public CodeAuthorAttribute(string Name)  
    {  
    }  
  
    public string Date  
    {  
        set  
        {  
        }  
    }  
}
```

Tras definir la propiedad, un parámetro con nombre puede establecer su propiedad cuando el atributo aparezca en el código:

```
[CodeAuthor("Jeff Ferguson", Date = "Apr 01 2001")]
```

A diferencia de los parámetros posicionales, los parámetros con nombre son opcionales y pueden omitirse de una especificación de atributo.

Ejemplo explicativo de las clases de atributo

En este apartado creará un nuevo atributo llamado `ClassAuthor` y lo usará en código C#. Esto le dará una idea sobre cómo el código .NET define y usa nuevos atributos. El listado 17.3 agrega una nueva clase al código del listado 17.2. Esta nueva clase recibe el nombre de `ClassAuthorAttribute` y deriva de la clase .NET `Attribute`.

Listado 17.3. Cómo definir nuevas clases de atributo

```
using System;
using System.Diagnostics;
using System.Reflection;

[AttributeUsage(AttributeTargets.Class)]
public class ClassAuthorAttribute : Attribute
{
    private string AuthorName;

    public ClassAuthorAttribute(string AuthorName)
    {
        this.AuthorName = AuthorName;
    }

    public string Author
    {
        get
        {
            return AuthorName;
        }
    }
}

[ClassAuthor("Jeff Ferguson")]
public class TestClass
{
    public void Method1()
    {
        Console.WriteLine("Hello from Method1!");
    }

    [Conditional("DEBUG")]
    public void Method2()
    {
        Console.WriteLine("Hello from Method2!");
    }

    public void Method3()
```

```

    {
        Console.WriteLine("Hello from Method3!");
    }
}

public class MainClass
{
    public static void Main()
    {
        TestClass MyTestClass = new TestClass();

        MyTestClass.Method1();
        MyTestClass.Method2();
        MyTestClass.Method3();

        object [] ClassAttributes;
        MemberInfo TypeInformation;

        TypeInformation = typeof(TestClass);
        ClassAttributes =
        TypeInformation.GetCustomAttributes(typeof(ClassAuthorAttribute),
        false);
        if(ClassAttributes.GetLength(0) != 0)
        {
            ClassAuthorAttribute ClassAttribute;

            ClassAttribute =
            (ClassAuthorAttribute) (ClassAttributes[0]);
            Console.WriteLine("Class Author: {0}",
            ClassAttribute.Author);
        }
    }
}

```

El código del listado 17.3 comienza con una nueva clase de atributo llamada `CodeAuthorAttribute`. La clase sirve como una clase de atributo para un atributo que sólo puede aplicarse a otras clases. La clase recibe un parámetro de cadena, que se almacena en una variable privada y al que se accede públicamente mediante una propiedad de sólo lectura llamada `Author`. La intención del parámetro es marcar una clase como poseedora de un nombre de programador específico adjunto, de modo que los demás programadores sepan con quién deben contactar si tienen alguna duda sobre la implementación de la clase.

La clase `TestClass` usa el atributo `CodeAuthor` y proporciona el parámetro `Jeff Ferguson`.

El rasgo más interesante del listado 17.3 es el método `Main()`, que obtiene un objeto de atributo de la clase y escribe el nombre del autor. Esto lo hace mediante un concepto llamado *reflexión*, que implementa las clases de un espacio de nombres .NET llamado `System.Reflection`. Mediante la *reflexión*, el código puede, en tiempo de ejecución, estudiar la implementación de una clase y descu-

brir cómo está construida. La reflexión permite que el código examine otros fragmentos de código para derivar información, como los métodos y propiedades que admite y la clase base de la que deriva. La reflexión es una función muy potente y es completamente compatible con .NET Framework.

El código del listado 17.3 usa la reflexión para obtener una lista de atributos asociados a una clase particular. El código de atributo comienza recuperando un objeto `Type` para la clase `TestClass`. Para conseguir el objeto `Type` se usa el operador de C# `typeof()`. Este operador toma como argumento el nombre de la clase cuyo tipo de información se va a recuperar. El objeto `Type` devuelto, que está definido en el espacio de nombre de .NET Framework `System`, funciona como una tabla de contenidos, describiendo todo lo que se debe saber sobre la clase requerida.

Después de recuperar el objeto `Type` para la clase, el método `Main()` llama a un método llamado `GetCustomAttributes()` para conseguir una lista de los atributos que permite la clase descrita por el objeto `Type`. Este método devuelve una matriz de objetos y acepta como parámetro el tipo del atributo que debe recuperarse. En el listado 17.3, el método `GetCustomAttributes()` es invocado con información de tipo para la clase `CodeAuthorAttribute` como parámetro. Esto obliga al método `GetCustomAttributes()` a devolver sólo información sobre los atributos de clase que sean del tipo `CodeAuthorAttribute`. Si la clase hubiera usado algún otro atributo, la llamada no podría devolverlos. El código del listado 17.3 finaliza tomando el primer atributo `CodeAuthorAttribute` de la matriz y solicitándole el valor de su propiedad `Author`. El valor de la cadena se escribe en la consola.

Si se ejecuta el código del listado 17.3 se escribe lo siguiente en la consola (si compila el código sin definir el símbolo `DEBUG`):

```
Hello from Method1!  
Hello from Method3!  
Class Author: Jeff Ferguson
```

Resumen

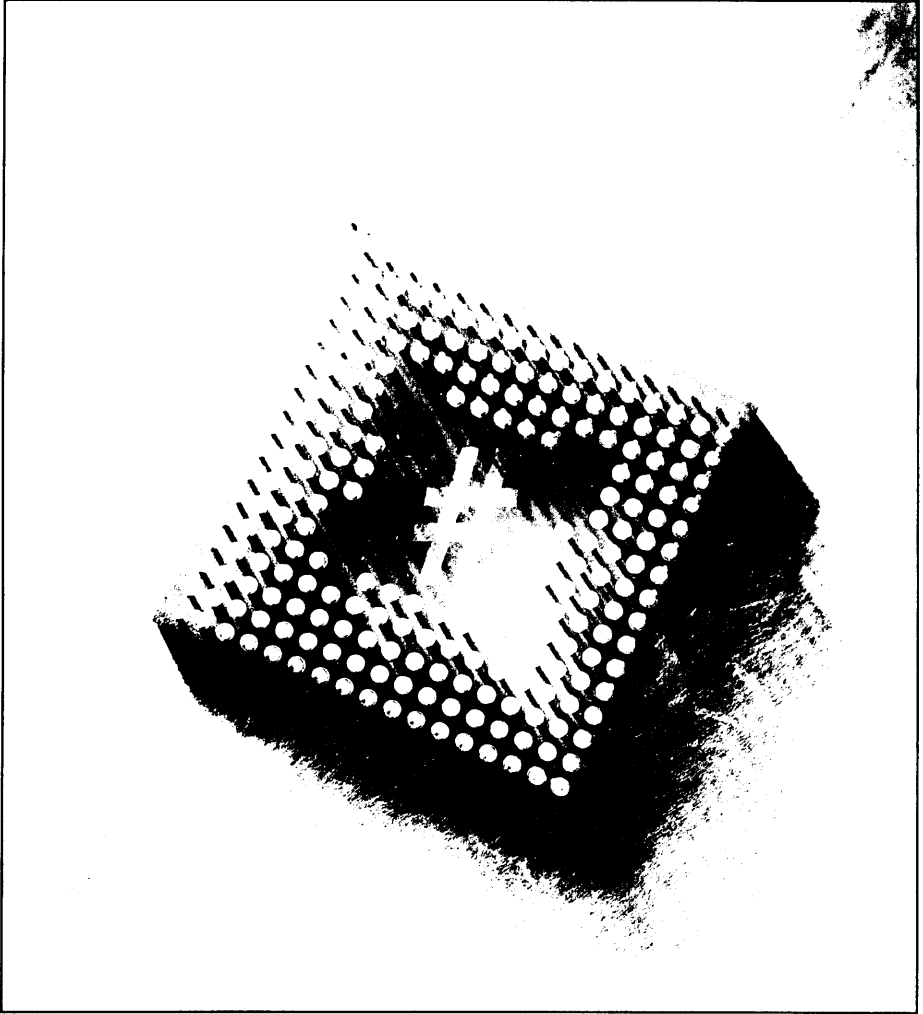
.NET Framework permite usar atributos en los lenguajes que se ejecutan con el CLR. El concepto de atributo abre la puerta a la expansión de la funcionalidad de los lenguajes .NET con clases que pueden agregar comportamientos al código. El lenguaje C# permite el uso de atributos creados por otras personas en su código C# y también la creación de atributos propios, que pueden ser usados por otros programadores de .NET.

El concepto de atributo no es exclusivo de C#; más bien, está disponible para cualquier lenguaje que se ejecute con el CLR. Los atributos le conceden la posibilidad de extender el entorno del lenguaje y aporta nuevas herramientas para que los programadores trabajen con código .NET. El proceso de serialización es un

buen ejemplo de esto. La serialización no está integrada en la especificación del lenguaje C#, pero su funcionalidad está disponible por medio de una clase de atributo escrita por Microsoft. La clase de atributo extiende el lenguaje en tiempo de ejecución para que admita una característica que no fue diseñada en ese lenguaje.

Al igual que los demás constructores de .NET Framework, los atributos son objetos. Se definen por clases que derivan de la clase `System.Attribute` de .NET Framework. Puede usar C# para desarrollar nuevas clases de atributo con sólo derivar una nueva clase de la clase base `System.Attribute`. Los atributos que desarrolle en C# y los atributos ya definidos por .NET Framework pueden ser usados por cualquier lenguaje compatible con el CLR.

Los atributos se usan especificando entre corchetes el nombre de clase del atributo, inmediatamente antes del constructor de C# al que se aplica el atributo. Los atributos pueden aceptar datos en forma de parámetros, que pueden asociar datos de estado al atributo. Estos datos pueden ser recuperados por código de reflexión que puede consultar el código y buscar atributos.



18 Cómo utilizar versiones en sus clases

Casi todo el código escrito para las modernas aplicaciones evoluciona con el tiempo. Los proyectos de software comienzan con un conjunto de requisitos y se diseñan las clases para que cumplan esos requisitos. Este primer código base sirve como código fuente de la versión 1.0 de la aplicación. Sin embargo, casi todas las aplicaciones van más allá de la versión 1.0. Las actualizaciones de la aplicación proceden de un grupo de requisitos mejorado y la versión 1.0 del código base debe ser revisada para implementar los requisitos actualizados.

El lenguaje C# admite constructores que hacen las clases lo suficientemente estables como para evolucionar mientras cambian los requisitos de la aplicación. En este capítulo, aprenderemos a usar las palabras clave `new` y `override` en métodos de clase de C# para asegurarnos de que las clases pueden continuar usándose a medida que cambian los requisitos de la aplicación.

El problema de las versiones

Antes de aprender a usar las palabras clave `new` y `override` para hacer las clases de su código de C# compatibles con el código base que tiene que mantenerse al día con los requisitos de cambio, observe cómo sería la vida sin estas pala-

bras clave. Si recuerda el capítulo 8, las clases que creamos y usamos pueden ser consideradas clases base. Estas clases tienen la funcionalidad básica que necesitan las aplicaciones. Al declarar una instancia de una clase, está derivando de esa clase para usar su funcionalidad.

Las bibliotecas de clases base de .NET Framework están basadas en este modelo: todo lo que hacemos mientras programamos aplicaciones .NET está basado en una clase base. Todo el entorno deriva de la clase base `System.Object`, de modo que incluso cuando deriva una simple variable está derivando una funcionalidad de la clase base `System.Object`.

El listado 18.1 muestra las características de las clases base y derivadas.

Listado 18.1. Una clase base y una clase derivada

```
using System;

public class BaseClass
{
    protected int Value;

    public BaseClass()
    {
        Value = 123;
    }
}

public class DerivedClass : BaseClass
{
    public void PrintValue()
    {
        Console.WriteLine("Value = " + Value);
    }
}

class MainClass
{
    public static void Main()
    {
        DerivedClass DerivedClassObject = new DerivedClass();
        DerivedClassObject.PrintValue();
    }
}
```

El código del listado 18.1 es relativamente sencillo. Contiene una clase base llamada `BaseClass` que incluye una variable entera protegida. Otra clase, llamada `DerivedClass`, deriva de la clase `BaseClass` e implementa un método llamado `PrintValue()`. El método `Main()` crea un objeto de tipo `DerivedClass` y llama a su método `PrintValue()`. Si se ejecuta el código del listado 18.1 se escribe lo siguiente en la consola:

```
Value = 123
```

Ahora suponga que los requisitos cambian y otro programador decide desarrollar la clase `BaseClass` mientras continuamos trabajando en nuevas mejoras para la clase `DerivedClass`. ¿Qué ocurrirá si el otro programador agrega un método a la clase `BaseClass` llamado `PrintValue()` y proporciona un implementación ligeramente diferente? El código sería como el listado 18.2.

Listado 18.2. Adición de `PrintValue()` a la clase `BaseClass`

```
using System;

public class BaseClass
{
    protected int Value;

    public BaseClass()
    {
        Value = 123;
    }

    public virtual void PrintValue()
    {
        Console.WriteLine("Value: " + Value);
    }
}

public class DerivedClass : BaseClass
{
    public void PrintValue()
    {
        Console.WriteLine("Value = " + Value);
    }
}

class MainClass
{
    public static void Main()
    {
        DerivedClass DerivedClassObject = new DerivedClass();

        DerivedClassObject.PrintValue();
    }
}
```

Ahora tenemos un problema. La clase `DerivedClass` deriva de la clase `BaseClass` y ambas implementan un método llamado `PrintValue()`. La clase `BaseClass` ha sido actualizada a una nueva versión, mientras que la clase `DerivedClass` ha permanecido con su implementación original. En el listado 18.2, la relación entre el método `PrintValue()` de la clase base y el método `PrintValue()` de la clase derivada no está clara. El compilador debe saber qué método reemplaza a la versión de la clase base. Y el compilador no sabe qué

implementación debe ejecutar cuando el método `Main()` llama al método `PrintValue()`.

Como resultado, el compilador de C# remarca esta ambigüedad con un aviso:

```
warning CS0114: 'DerivedClass.PrintValue()' oculta el miembro heredado 'BaseClass.PrintValue()'. Para hacer que el método actual reemplace esa implementación, agregue la palabra clave override. De lo contrario, agregue la palabra clave new.
```

Éste es un buen aviso, porque la filosofía del lenguaje C# fomenta la claridad y el compilador de C# siempre avisa sobre los constructores de código que no están claros.

Cómo solucionar el problema de las versiones

C# tiene dos modos de solucionar la ambigüedad del listado 18.2:

- Usar el modificador `new` para especificar los dos métodos que, en realidad, son diferentes.
- Usar el modificador `override` para especificar que el método de la clase derivada debe reemplazar al método de la clase base.

Examinemos estos dos métodos.

Mediante el modificador `new`

Si las dos implementaciones de método del listado 18.2 deben ser tratadas como métodos separados que simplemente tienen el mismo nombre, el método de la clase derivada debe llevar antepuesto el modificador `new`. Al usar el modificador `new`, puede ocultar explícitamente los miembros heredados de la implementación de la clase base. Simplemente debe declarar un miembro en su clase derivada con el mismo nombre y anteponer a la declaración el modificador `new` y se usará la funcionalidad de la clase derivada, como muestra el listado 18.3.

Listado 18.3. Cómo resolver la ambigüedad mediante la palabra reservada `new`

```
using System;

public class BaseClass
{
    protected int Value;

    public BaseClass()
```

```

    {
        Value = 123;
    }

    public void PrintValue()
    {
        Console.WriteLine("Value: " + Value);
    }
}

public class DerivedClass : BaseClass
{
    new public void PrintValue()
    {
        Console.WriteLine("Value = " + Value);
    }
}

class MainClass
{
    public static void Main()
    {
        DerivedClass DerivedClassObject = new DerivedClass();

        DerivedClassObject.PrintValue();
    }
}

```

NOTA: El operador `new` y el modificador `new` son implementaciones diferentes de la palabra clave `new`. El operador `new` se usa para crear objetos, mientras que el modificador `new` se usa para ocultar un miembro heredado de un miembro de clase base.

El código del listado 18.3 usa la palabra clave `new` en la implementación del método `PrintValue()` de la clase `DerivedClass`. Esto indica al compilador de C# que debe tratar este método como distinto del método de la clase base, aunque los dos métodos tengan el mismo nombre. El uso de la palabra clave resuelve la ambigüedad y permite que el compilador de C# compile el código sin emitir advertencias. En este caso, el método `Main()` llama al método de la clase derivada y el listado 18.3 escribe lo siguiente en la consola:

```
Value = 123
```

Todavía puede ejecutar el método de la clase base porque la palabra clave `new` ha asegurado básicamente que los dos métodos `PrintValue()` de cada una de las clases puede ser llamado por separado. Puede llamar al método de la clase base convirtiendo explícitamente el objeto de la clase derivada en un objeto de tipo de la clase base:


```
BaseClass BaseClassObject = (BaseClass)DerivedClassObject;

BaseClassObject.PrintValue();
```

Como puede ver, el uso del modificador `new` solamente permite reemplazar la funcionalidad en una clase base.

Si necesita usar la funcionalidad de la clase original, use el nombre de la clase completo y su ruta de acceso con el miembro de la clase para estar seguro de que está usando la funcionalidad correcta.

Mediante el modificador `override`

La otra opción para resolver la ambigüedad con los nombres de método duplicados es usar el modificador `override` para especificar que la implementación de la clase derivada reemplaza a la implementación de la clase base. El modificador `override` "reemplaza" la funcionalidad del miembro de la clase base al que sustituye.

Para reemplazar a un miembro de una clase, la firma del miembro que reemplaza debe ser la misma que el miembro de la clase base. Por ejemplo, si el miembro que realiza el reemplazo tiene un constructor, los tipos en el constructor deben coincidir con los del miembro de la clase base. En el listado 18.4 puede ver el funcionamiento del modificador `override`.

Listado 18.4. Cómo resolver la ambigüedad mediante el modificador `override`

```
using System;

public class BaseClass
{
    protected int Value;

    public BaseClass()
    {
        Value = 123;
    }

    public virtual void PrintValue()
    {
        Console.WriteLine("Value: " + Value);
    }
}

public class DerivedClass : BaseClass
{
    override public void PrintValue()
    {
        Console.WriteLine("Value = " + Value);
    }
}
```

```

}

class MainClass
{
    public static void Main()
    {
        DerivedClass DerivedClassObject = new DerivedClass();

        DerivedClassObject.PrintValue();
    }
}

```

En el listado 18.4, la palabra clave `override` indica al compilador de C# que la implementación de `PrintValue()` en la clase derivada reemplaza a la implementación del mismo método en la clase base. La implementación de la clase base está oculta básicamente a los invocadores. A diferencia del listado 18.3, el código del listado 18.4 sólo contiene una implementación de `PrintValue()`.

La implementación de la clase base de `PrintValue()` no es accesible al código del método `Main()`, incluso si el código convierte explícitamente el objeto de la clase derivada en un objeto de clase base y llama al método en el objeto de la clase base.

Debido a que la palabra clave `override` se usa en el listado 18.4, todas las llamadas al método realizadas mediante un objeto convertido explícitamente son dirigidas a la implementación reemplazada de la clase derivada.

Observe el código usado en la llamada a la implementación de la clase base de `PrintValue()` cuando se usa el operador `new` para resolver la ambigüedad:

```

BaseClass BaseClassObject = (BaseClass)DerivedClassObject;

BaseClassObject.PrintValue();

```

Este código no es suficiente para hacer que se llame a la implementación de la clase base cuando se use la palabra clave `override`. Esto es debido a que el objeto fue creado como un objeto de la clase `DerivedClass`. Puede llamar al método de la clase base pero, debido a que la implementación de la clase base ha sido reemplazada con el código de la clase derivada, se seguirá invocando a la implementación de la clase derivada. Debe usar la palabra clave de C# `base` para llamar a la implementación de la clase base, como en el siguiente ejemplo:

```

base.PrintValue();

```

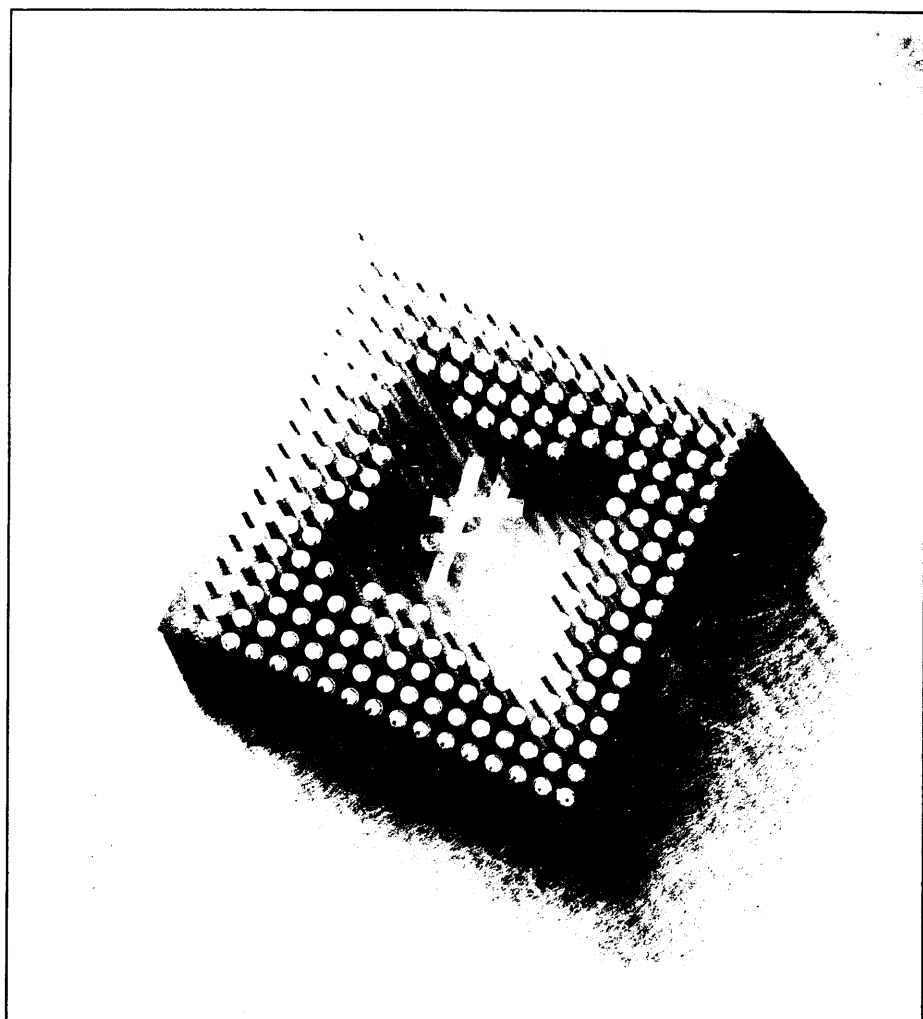
Esta instrucción llama a la implementación de `PrintValue()` que se encuentra en la clase base de la clase actual. Al colocar esta instrucción en la clase `DerivedClass`, por ejemplo, se llama a la implementación de `PrintValue()` que se encuentra en la clase `BaseClass`. Puede considerar el uso de la palabra clave `base` igual al uso de la sintaxis completa y con ruta de acceso `namespace.object.method`, ambas simplemente hacen referencia a la instancia de clase base correcta que se está usando.

Resumen

Los ejemplos de este capítulo situaban juntas todas las clases del listado en un único archivo fuente para hacerlos más sencillos. Sin embargo, la programación en el mundo real puede ser más complicada. Si varios programadores están trabajando en un único proyecto, el proyecto podría estar formado por más de un archivo fuente. El programador que trabaja en la clase base podría colocarla en un archivo fuente de C# y el programador que trabaja en la clase derivada puede colocarla en otro archivo fuente de C#. El asunto podría ser aún más complicado si la clase base se compila en un ensamblado y se implementa la clase derivada en un proyecto que hace referencia al ensamblado.

Lo importante en este caso es que las clases base y las clases derivadas pueden proceder de varias fuentes diferentes y que coordinar la programación de las clases cobra una gran importancia. Es de vital importancia comprender que, con el tiempo, las clases base y las clases derivadas añadirán funcionalidades a medida que el proyecto progrese. Como programador, debería tener esto en cuenta: diseñe sus clases de modo que pueden ser usadas en varias versiones de un proyecto y puedan evolucionar a medida que evolucionen los requisitos del proyecto.

A priori, la otra solución para los problemas que presentan las versiones es incluso más sencillo: no use el mismo nombre para métodos que tengan implementaciones diferentes a menos que esté reemplazando realmente la funcionalidad de la clase base. Aunque, en teoría, éste puede parecer el mejor medio para solventar el problema, en la práctica no siempre es posible. Las palabras clave de C# `new` y `override` le ayudan a evitar este problema de programación y le permiten reutilizar nombres de métodos en caso de que sea necesario para su proyecto. El principal uso de la palabra clave `override` es avisar de la creación de nuevas implementaciones de métodos virtuales en clases base, pero también tiene un papel en las versiones de C#.



19 Cómo trabajar con código no seguro

Al usar la palabra clave `new` para crear una nueva instancia de un tipo de referencia, está pidiendo al CLR que reserve suficiente memoria para la variable. El CLR asigna suficiente espacio en la memoria para la variable y asocia la memoria a la variable. En condiciones normales, el código desconocería la localización actual de esa memoria, por lo que respecta a la dirección de memoria. Después de que la operación `new` tenga éxito, el código puede usar la memoria asignada sin saber ni importarle en qué parte del sistema está realmente situada la memoria.

En C y C++, los programadores tienen acceso directo a la memoria. Cuando un fragmento de código de C o de C++ solicita acceso a un bloque de memoria, se le otorga la dirección específica de la dirección asignada y el código lee y escribe directamente en esa posición de memoria. La ventaja de este enfoque es que el acceso directo a la memoria es extremadamente rápido y contribuye a la eficiencia del código. Sin embargo, hay algunos inconvenientes que superan a las ventajas. El problema de este acceso directo a la memoria es que es fácil usarlo incorrectamente y esto hace que el código falle. Un mal funcionamiento del código de C o C++ puede fácilmente escribir en la memoria de otra variable. Estos problemas de acceso a memoria producen numerosos fallos y errores de software difíciles de localizar. La arquitectura del CLR elimina todos estos problemas realizando toda la gestión de memoria por nosotros. Esto significa que su código de

C# puede trabajar con variables sin que necesitemos conocer los detalles de cómo y dónde se almacenan las variables en memoria. Como el CLR protege al código de C# de estos detalles de la memoria, el código de C# está libre de errores relacionados con el acceso directo a la memoria.

No obstante, alguna vez tendrá que trabajar con una dirección de memoria específica del código C#. El código puede necesitar un poco más de rendimiento o quizás el código de C# quiera trabajar con código heredado que necesite que le proporcione la dirección de un fragmento de memoria específico. El lenguaje C# admite un modo especial, llamado *modo no seguro*, que permite trabajar directamente con memoria desde el interior del código C#.

Este constructor de C# especial recibe el nombre de modo no seguro porque el código ya no dispone de la protección que ofrece la gestión de memoria del CLR. En el modo no seguro, el código de C# puede acceder directamente a la memoria y puede tener los mismos fallos relacionados con la memoria que los códigos de C y C++ si no es extremadamente cuidadoso con su forma de gestionar la memoria.

En este capítulo, estudiaremos el modo no seguro del lenguaje C# y cómo puede ser usado para permitirle acceder directamente a las direcciones de memoria usando constructores de estilo puntero de C y C++.

Conceptos básicos de los punteros

En C# se accede a la memoria usando un tipo de dato especial llamado *puntero*. Un puntero es una variable cuyo valor apunta a una dirección específica de la memoria. En C#, un puntero se declara con un asterisco situado entre el tipo del puntero y su identificador, como se muestra en la siguiente declaración:

```
int * MyIntegerPointer;
```

Esta instrucción declara un puntero entero llamado `MyIntegerPointer`. El tipo del puntero indica el tipo de la variable a la que el puntero puede apuntar. Por ejemplo, un puntero entero sólo puede apuntar a memoria usada por una variable entera. Los punteros deben ser asignados a una dirección de memoria y C# hace que sea fácil escribir una expresión que evalúe la dirección de memoria de una variable. Si se antepone el operador de concatenación, el símbolo de unión, a una expresión unaria, devuelve una dirección de memoria, como muestra en el siguiente ejemplo:

```
int MyInteger = 123;  
int * MyIntegerPointer = &MyInteger;
```

El código anterior hace dos cosas:

- Declara una variable entera llamada `MyInteger` y le asigna el valor 123.

- Declara una variable entera llamada `MyInteger` y la apunta en la dirección de la variable `MyInteger`.

La figura 19.1 muestra cómo se interpreta esta asignación en memoria.

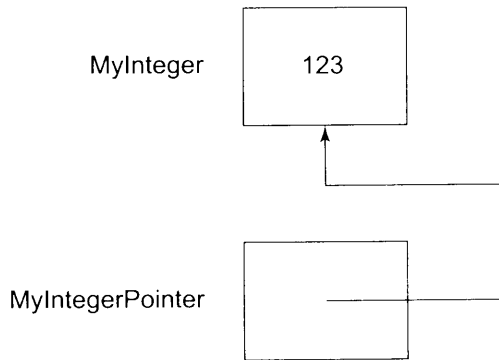


Figura 19.1. Un puntero apuntando a una variable

Los punteros en realidad tienen dos valores:

- El valor de la dirección de memoria del puntero
- El valor de la variable a la que apunta el puntero

C# permite escribir expresiones que evalúen cualquiera de los dos valores. Si se antepone un asterisco al identificador del puntero se obtiene el valor de la variable a la que apunta el puntero, como demuestra el siguiente código:

```
int MyInteger = 123;
int * MyIntegerPointer = &MyInteger;

Console.WriteLine(*MyIntegerPointer);
```

Este código escribe en la consola 123.

Tipos de puntero

Los punteros pueden tener uno de los siguientes tipos:

- `sbyte`
- `byte`
- `short`
- `ushort`
- `int`
- `uint`

- long
- ulong
- char
- float
- double
- decimal
- bool
- un tipo de enumeración
- void, usado para especificar un puntero para un tipo desconocido

No se puede declarar un puntero para un tipo de referencia, como un objeto. La memoria para los objetos está gestionada por el CLR y la memoria puede ser borrada cada vez que el recolector de elementos no utilizados necesite liberar la memoria del objeto. Si el compilador de C# le permite mantener un puntero sobre un objeto, su código corre el riesgo de apuntar a un objeto cuya memoria puede ser liberada en otro punto por el recolector de elementos no utilizados del CLR.

Imagine que el compilador de C# permitiese escribir código como el siguiente:

```
MyClass MyObject = new MyClass();
MyClass * MyObjectPointer;

MyObjectPointer = &MyObject;
```

La memoria usada por `MyObject` es gestionada automáticamente por el CLR y su memoria es liberada cuando todas las referencias al objeto se liberan y se ejecuta el recolector de elementos no utilizados del CLR. El problema es que el código no seguro ahora contiene un puntero al objeto y el resultado es que tiene un puntero que apunta hacia un objeto cuya memoria se ha liberado. El CLR no tiene ningún modo de saber que hay un puntero para el objeto y el resultado es que, después de que el recolector de elementos no utilizados haya liberado la memoria, tiene un puntero apuntando hacia la nada. C# solventa este problema no permitiendo que existan variables a tipos de referencia con memoria que es gestionada por el CLR.

Cómo compilar código no seguro

Por defecto, el compilador de C# sólo compila código seguro de C#. Para obligar al compilador a compilar código de C# no seguro debe usar el argumento del compilador `/unsafe`:

```
csc /unsafe file1.cs
```

El código no seguro permite escribir código que acceda directamente a la memoria, sin hacer caso de los objetos que gestionan la memoria en las aplicaciones. Como a las direcciones de memoria se accede directamente, el código no seguro puede funcionar mejor en algunos tipos de aplicaciones. Esta instrucción compila el archivo fuente `file1.cs` y permite compilar el código no seguro de C#.

NOTA: En C#, el código no seguro permite declarar y usar punteros del mismo modo que en C++.

Cómo especificar punteros en modo no seguro

El compilador de C# no permite por defecto usar punteros en el código C#. Si intenta trabajar con punteros en su código, el compilador de C# emitirá el siguiente mensaje de error:

```
error CS0214: Los punteros solo se pueden utilizar en un
contexto no seguro
```

Los punteros sólo son válidos en C# en código no seguro y hay que definir explícitamente el código no seguro al compilador. Para hacerlo se emplea la palabra clave de C# `unsafe`. La palabra clave `unsafe` debe aplicarse a un bloque de código que use punteros.

Para especificar que un bloque de código se ejecute en el modo no seguro de C# se aplica la palabra clave `unsafe` a la declaración del cuerpo de código, como se muestra en el listado 19.1.

Listado 19.1. Métodos no seguros

```
using System;

public class MyClass
{
    public unsafe static void Main()
    {
        int MyInteger = 123;
        int * MyIntegerPointer = &MyInteger;

        Console.WriteLine(*MyIntegerPointer);
    }
}
```

El método `Main()` del listado 19.1 usa el modificador `unsafe` en su declaración. Esto indica al compilador de C# que todo el código del método debe ser considerado no seguro. Después de usar esta palabra clave, el código del método puede usar constructores de puntero no seguros.

La palabra clave `unsafe` se aplica sólo al método en el que aparece. Si la clase del listado 19.1 va a contener otro método, ese otro método no podrá usar constructores de puntero no seguros a menos que, también, sea declarado con la palabra clave `unsafe`. Las siguientes reglas se aplican al modificador `unsafe`.

- Clases, estructuras y delegados pueden incluir el modificador `unsafe`, que indica que todo el cuerpo del tipo se considera no seguro.
- Campos, métodos, propiedades, eventos, indizadores, operadores, constructores, destructores y constructores estáticos pueden definirse con el modificador `unsafe`, que indica que la declaración del miembro específico no es segura.
- Un bloque de código puede ser marcado con el modificador `unsafe`, que indica que todo el código debe ser considerado no seguro.

Cómo acceder a los valores de los miembros mediante punteros

El modo no seguro de C# permite usar el operador `->` para acceder a los miembros de las estructuras a las que hace referencia el puntero. El operador, que se escribe como un guión seguido por el símbolo mayor que, le permite acceder directamente a los miembros, como se muestra en el listado 19.2.

Listado 19.2. Cómo acceder a los miembros de una estructura con un puntero

```
using System;

public struct Point2D
{
    public int X;
    public int Y;
}

public class MyClass
{
    public unsafe static void Main()
    {
        Point2D MyPoint;
        Point2D * PointerToMyPoint;

        MyPoint = new Point2D();
        PointerToMyPoint = &MyPoint;
        PointerToMyPoint->X = 100;
        PointerToMyPoint->Y = 200;
        Console.WriteLine("{0}, {1}", PointerToMyPoint->X,
        PointerToMyPoint->Y);
    }
}
```

El listado 19.2 contiene la declaración de una estructura llamada `Point2D`. La estructura tiene dos miembros públicos. El listado también incluye un método no seguro `Main()` que crea una nueva variable del tipo de la estructura y crea un puntero para la nueva estructura. A continuación, el método usa el operador de acceso a miembros del puntero para asignar valores a la estructura, que se escribe en la consola.

Esto es diferente del acceso a miembros del modo seguro, por defecto, de C#, que usa el operador `.`. Si se usa un operador incorrecto en un modo incorrecto, el compilador de C# emite un error. Si usa el operador `.` con un puntero no seguro, el compilador de C# emite el siguiente mensaje de error:

```
error CS0023: El operador '.' no se puede aplicar a operandos del tipo 'Point2D*'
```

Si se usa el operador `->` en un contexto seguro, el compilador de C# también emite un mensaje de error:

```
error CS0193: El operador * o -> se debe aplicar a un puntero
```

Cómo usar punteros para fijar variables a una dirección específica

Cuando el CLR gestiona la memoria de una variable, el código trabaja con una variable y los detalles sobre la memoria de la variable son controlados por el CLR. Durante el proceso de recolección de elementos no utilizados del CLR, el tiempo de ejecución puede cambiar la memoria de lugar varias veces para afianzar la pila de memoria disponible en tiempo de ejecución. Esto significa que durante el curso de una aplicación, la dirección de memoria para una variable puede cambiar. El CLR puede tomar los datos de la variable y moverlos a una dirección diferente.

En condiciones normales, el código de C# no tiene en cuenta esta técnica de recolocación. Como el código trabaja con un identificador de variable, normalmente accederá a la memoria de la variable mediante el identificador de la variable y puede confiar en que el CLR trabaje con el fragmento de memoria correcto mientras trabaja con la variable.

Sin embargo, la situación no es tan sencilla cuando se trabaja con punteros. Los punteros apuntan a una dirección de memoria específica. Si asigna un puntero a una dirección de memoria usada por una variable y el CLR después mueve la dirección de memoria de esa variable, su puntero estará apuntando a una memoria que ya no está siendo usada por la variable.

El modo no seguro de C# permite especificar una variable como excluida de la recolocación de memoria del CLR. Esto permite mantener una variable en una dirección específica de memoria, lo que le permite usar un puntero con la variable

sin preocuparse de que el CLR pueda mover la dirección de memoria de la variable de la dirección a la que apunta su puntero. Para especificar que la dirección de memoria de una variable debe ser fija se usa la palabra clave de C# `fixed`. La palabra clave `fixed` va seguida de una expresión entre paréntesis que contiene una declaración de puntero con una asignación a una variable. La expresión fijada va seguida de un bloque de código y la variable fijada permanece en la misma dirección de memoria a lo largo del bloque de código fijado, como se muestra en el listado 19.3.

Listado 19.3. Cómo fijar en memoria los datos gestionados

```
using System;

public class MyClass
{
    public unsafe static void Main()
    {
        int ArrayIndex;
        int [] IntegerArray;

        IntegerArray = new int [5];
        fixed(int * IntegerPointer = IntegerArray)
        {
            for(ArrayIndex = 0; ArrayIndex < 5; ArrayIndex++)
                IntegerPointer[ArrayIndex] = ArrayIndex;
        }
        for(ArrayIndex = 0; ArrayIndex < 5; ArrayIndex++)
            Console.WriteLine(IntegerArray[ArrayIndex]);
    }
}
```

La palabra clave `fixed` del listado 19.3 declara un puntero entero que apunta a una matriz entera. Va seguida por un bloque de código que escribe valores en la matriz usando un puntero. Dentro de este bloque de código, está garantizado que la dirección de la matriz `IntegerArray` es fija y que el CLR no moverá su posición. Esto permite al código usar un puntero con la matriz sin preocuparse de si el CLR va a mover la posición de la memoria física de la matriz. Después de que el bloque de código fijado termine, ya no puede usarse el puntero y el CLR vuelve a tener en cuenta a la variable `IntegerArray` cuando reubica la memoria.

Sintaxis del elemento de matriz puntero

El listado 19.3 también muestra la sintaxis del elemento de matriz puntero. La siguiente línea de código trata un puntero de modo no seguro como si fuera una matriz de bytes:

```
IntegerPointer[ArrayIndex] = ArrayIndex;
```

Esta línea de código trata al puntero como si fuese una matriz. La sintaxis del elemento de matriz puntero permite al código de C# no seguro ver la memoria a la que apunta el puntero como una matriz de variables en la que se puede escribir y leer de ella.

Cómo comparar punteros

El modo no seguro de C# permite comparar punteros usando los siguientes operadores:

- Igualdad (==)
- Desigualdad (!=)
- Menor que (<)
- Mayor que (>)
- Menor o igual que (<=)
- Mayor o igual que (>=)

Al igual que los tipos de valores, estos operadores devuelven los valores booleanos `True` y `False` cuando se usan con tipos de puntero.

Cálculo con punteros

Se pueden combinar punteros con valores enteros en expresiones matemáticas para cambiar la posición a la que apunta el puntero. El operador `+` suma un valor al puntero y el operador `-` resta un valor del puntero. La instrucción `fixed` del listado 19.3 también puede escribirse como se indica a continuación:

```
fixed(int * IntegerPointer = IntegerArray)
{
    for(ArrayIndex = 0; ArrayIndex < 5; ArrayIndex++)
        *(IntegerPointer + ArrayIndex) = ArrayIndex;
}
```

En este bloque de código, el puntero es desplazado por un valor y la suma se usa para apuntar a una dirección de memoria. La siguiente instrucción realiza aritmética de puntero:

```
*(IntegerPointer + ArrayIndex) = ArrayIndex;
```

Esta instrucción debe interpretarse como: "Toma el valor de `IntegerPointer` e increméntalo en el número de posiciones especificadas por `ArrayIndex`. Coloca el valor de `ArrayIndex` en esa posición".

La aritmética de puntero aumenta la posición de un puntero en un número especificado de bytes, dependiendo del tamaño del tipo al que se está apuntando.

El listado 19.3 declara una matriz entera y un puntero entero. Cuando se usa aritmética de puntero en el puntero entero, el valor usado para modificar el puntero especifica el número de tamaños de variable que deben moverse, no el número de bytes. La siguiente expresión usa aritmética de puntero para desplazar la localización de un puntero en tres bytes:

```
IntegerPointer + 3
```

El valor literal 3 de esta expresión especifica que se debe incrementar el puntero en el espacio que ocupan tres números enteros, no en tres bytes. Dado que el puntero apunta a un número entero, el 3 se interpreta como "espacio necesario para tres números enteros" y no "espacio para tres bytes". Dado que un número entero ocupa cuatro bytes de memoria, la dirección del puntero se aumenta en doce bytes (tres números enteros multiplicado por cuatro bytes por cada número entero), no en tres.

Cómo usar el operador sizeof

Se puede usar el operador `sizeof` en modo no seguro para calcular el número de bytes necesarios para contener un tipo de datos específico. El operador va seguido por un nombre de tipo no administrado entre paréntesis y la expresión da como resultado un número entero que especifica el número de bytes necesario para contener una variable del tipo especificado. La tabla 19.1 enumera los tipos administrados admitidos y los valores que devuelve una operación `sizeof`.

Tabla 19.1. Tipos `sizeof()` admitidos

Expresión	Resultado
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(bool)</code>	1

Cómo asignar espacio de la pila para la memoria

C# proporciona un sencillo mecanismo de asignación de memoria en código no seguro. Puede solicitar memoria en modo no seguro usando la palabra clave de C# `stackalloc`, como se muestra en el listado 19.4.

Listado 19.4. Cómo asignar espacio de la pila para la memoria

```
using System;

public class MyClass
{
    public unsafe static void Main()
    {
        int * CharacterBuffer = stackalloc int [5];
        int Index;

        for(Index = 0; Index < 5; Index++)
            CharacterBuffer[Index] = Index;
        for(Index = 0; Index < 5; Index++)
            Console.WriteLine(CharacterBuffer[Index]);
    }
}
```

Tras la palabra clave `stackalloc` se escribe un tipo de dato. Devuelve un puntero al bloque de memoria al que se le asigna el espacio y se puede usar la memoria exactamente igual que la memoria gestionada por el CLR.

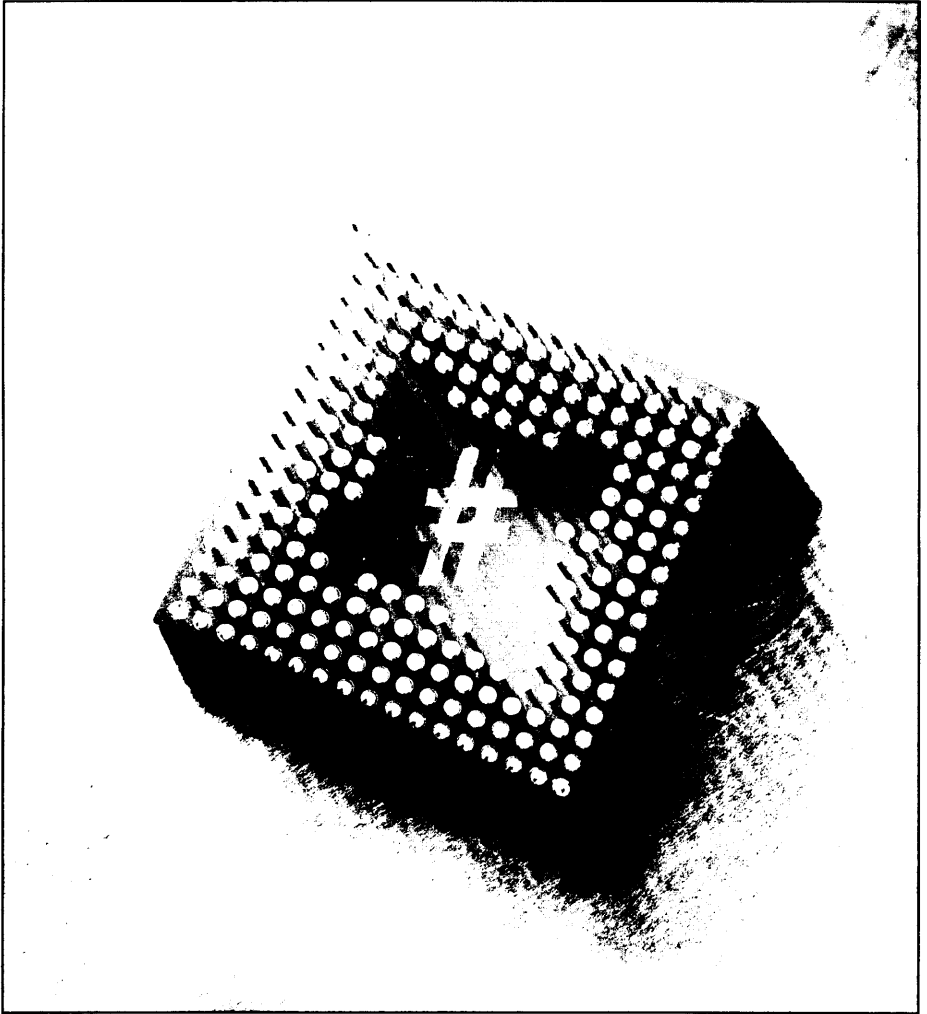
No hay una operación explícita para liberar la memoria asignada por la palabra clave `stackalloc`. La memoria se libera automáticamente cuando finaliza el método que asignó esa memoria.

Resumen

El modo no seguro de C# permite a su código trabajar directamente con la memoria. Su uso puede mejorar el rendimiento porque el código accede directamente a la memoria, sin tener que moverse con cuidado por el CLR. Sin embargo, el modo no seguro puede ser peligroso y puede hacer que el código falle si no trabaja adecuadamente con la memoria.

En general, evite el uso del modo no seguro de C#. Si necesita un poco más de rendimiento para su código o si está trabajando con código heredado de C o C++ que necesita que especifique una posición de memoria, siga con el modo seguro que se ofrece por defecto y deje que el CLR gestione los detalles de la asignación de memoria.

La pequeña reducción en el rendimiento que se produce es compensada con creces por no tener que realizar la pesada tarea de gestionar la memoria de su código y por conseguir la posibilidad de escribir código libre de errores relacionados con la gestión de memoria.



Constructores avanzados de C#

En este capítulo examinaremos algunas facetas interesantes del lenguaje C#. También veremos algunos ejemplos de código y aprenderemos por qué el código funciona como lo hace. La comprensión de problemas de programación como los presentados en este capítulo le ayudarán a enfrentarse a sus propias dudas sobre programación en C#.

En primer lugar, observe la función de conversión implícita de C# y cómo se aplica a objetos de clases derivadas a las que se accede como objetos de la clase base de la clase derivada. Recuerde que puede escribir métodos de operadores implícitos que definan cómo se gestionan las conversiones implícitas de un tipo u otro; pero, como verá, las cosas se vuelven un poco más complicadas cuando se trabaja con tipos de tiempo de ejecución y de tiempo de compilación.

A continuación, nos adentraremos en la inicialización de estructuras. Las estructuras, al igual que las clases, pueden contener campos y propiedades. Sin embargo, la inicialización de estructuras con campos se realiza de forma ligeramente diferente a la inicialización de estructuras con propiedades. En este capítulo, descubrirá por qué y cómo resolver este problema.

En la tercera parte de este capítulo, investigaremos el paso de un objeto de una clase derivada a una llamada de método en el que se espera un objeto de una clase base. Dado que los objetos de las clases derivadas son inherentemente objetos de la clase base, pasar un objeto de clase derivada a un elemento que espera una

clase base puede parecer bastante sencillo. En este apartado estudiaremos por qué esta técnica no es tan simple como podría parecer.

Finalmente, nos adentraremos en el uso avanzado de los indizadores de clase. En la inmensa mayoría de los casos, los indizadores que escriba servirán para hacer que una clase se comporte como una matriz de elementos. Por lo general, las matrices aceptan valores enteros como para especificar el elemento del índice. En este apartado estudiaremos la técnica de usar tipos de datos distintos de enteros para los índices de matriz.

Operadores implícitos y conversiones no válidas

Recuerde que las clases pueden contener un código de conversión de operadores implícitos. Los operadores implícitos se usan para convertir un tipo en otro sin ningún código especial. En el lenguaje C# se crean muchas conversiones implícitas. Por ejemplo, un entero puede ser convertido implícitamente en un entero `long` sin ningún código especial:

```
MyInt = 123;
long MyLongInt = MyInt;
```

C# no define conversiones implícitas para todas las combinaciones existentes de tipos de datos. Sin embargo, puede escribir código de conversión de operadores implícitos que indique al entorno común de ejecución (CLR) cómo debe comportarse cuando un usuario de la clase intente hacer una conversión implícita entre la clase y otro tipo. En este apartado estudiaremos una faceta del operador de conversión implícita que trata con la conversión entre dos clases diferentes.

El listado 20.1 contiene dos clases: `TestClass` y `MainClass`. La clase `MainClass` contiene el método `Main()` de la aplicación. La clase `TestClass` contiene una variable privada de tipo `MainClass`. También define un método de operador implícito que convierte los objetos `TestClass` en objetos `MainClass`. La implementación de operador implícito devuelve una referencia al objeto privado `MainClass` del objeto `TestClass`.

Listado 20.1. Excepciones a las conversiones inválidas con operadores implícitos

```
public class TestClass
{
    private MainClass MyMainClassObject;

    public TestClass()
    {
        MyMainClassObject = new MainClass();
    }
}
```

```

    public static implicit operator MainClass(TestClass Source)
    {
        return Source.MyMainClassObject;
    }
}

public class MainClass
{
    public static void Main()
    {
        object MyObject;
        MainClass MyMainClassObject;
        MyObject = new TestClass();
        MyMainClassObject = (MainClass)MyObject;
    }
}

```

El código del método `Main()` crea un nuevo objeto `TestClass` y convierte explícitamente el objeto en otro objeto de tipo `MainClass`. Como en `TestClass` se define un operador implícito, esta conversión debe tener éxito. Sin embargo, si ejecuta el listado 20.1, recibirá el siguiente mensaje en la consola:

```

Excepción no controlada: System.InvalidCastException: La
conversión especificada no es válida.
   at MainClass.Main()

```

¿Por qué el CLR considera que la conversión es inválida, incluso cuando se define un operador implícito? El problema aquí es que la conversión funciona entre tipos de tiempo de compilación, no entre tipos de tiempo de ejecución.

El método `Main()` crea un nuevo objeto de tipo `TestClass` y asigna el nuevo objeto a la variable de tipo `object`. A continuación, esta variable se convierte en un tipo de clase `MainClass`. Como el objeto fue creado como un objeto de tipo `TestClass`, puede esperar que la conversión explícita convierta un objeto de tipo `TestClass` en un objeto de tipo `MainClass`.

Sin embargo, C# no realiza conversiones explícitas basadas en el tipo usado en el momento de crear el objeto. En su lugar, realiza conversiones basadas en el tipo de variable que contiene el nuevo objeto. En el listado 20.1, el nuevo objeto se asigna a una variable de tipo `object`. Por tanto, el compilador de C# genera código que convierte un objeto de tipo `object` a tipo `MainClass`. Como no se ha definido una conversión de `object` a `MainClass`, no se produce la conversión explícita.

Inicialización de estructuras

Como sabe, las estructuras pueden contener constructores de lenguaje que también se encuentran en las clases, incluyendo métodos, campos y propiedades. Para asignar valores a los métodos y campos de una estructura se usa una simple

instrucción de asignación. Sin embargo, es importante recordar que la diferencia entre una propiedad y un campo es que si se asigna un valor a una propiedad, se ejecuta el código compilado en la estructura. Esto significa que se debe tener especial cuidado cuando se asignan valores a propiedades de estructuras recién creadas. Este capítulo estudia este tema.

Cómo inicializar estructuras

El listado 20.2 contiene dos estructuras. La primera recibe el nombre de `StructWithPublicMembers` y contiene dos miembros públicos enteros llamados `X` e `Y`. La segunda estructura se llama `StructWithProperties` y contiene propiedades públicas llamadas `X` e `Y`, que gestionan dos números enteros privados.

El método `Main()` del listado 20.2 crea dos variables, una que tiene el tipo de estructura `StructWithPublicMembers` y otra que tiene el tipo de estructura `StructWithProperties`. El código asigna valores a los miembros `X` e `Y` de cada estructura.

Listado 20.2. Cómo acceder a las estructuras con propiedades y métodos públicos

```
public struct StructWithPublicMembers
{
    public int X;
    public int Y;
}

public struct StructWithProperties
{
    private int PrivateX;
    private int PrivateY;

    public int X
    {
        get
        {
            return PrivateX;
        }
        set
        {
            PrivateX = value;
        }
    }

    public int Y
    {
        get
        {
            return PrivateY;
        }
    }
}
```

```

        set
        {
            PrivateY = value;
        }
    }
}

public class MainClass
{
    public static void Main()
    {
        StructWithPublicMembers MembersStruct;
        StructWithProperties PropertiesStruct;

        MembersStruct.X = 100;
        MembersStruct.Y = 200;
        PropertiesStruct.X = 100;
        PropertiesStruct.Y = 200;
    }
}

```

El listado 20.2 no se compila. El compilador de C# emite el siguiente error:

```

error CS0165: Uso de la variable local no asignada
'PropertiesStruct'

```

Lo más interesante de este error es que hace referencia a la segunda variable de estructura definida en el método `Main()`. Sin embargo, el compilador de C# compila el código que funciona con la primera variable de estructura. En otras palabras, se puede acceder a los miembros públicos del listado 20.2, pero no a las propiedades públicas del listado 20.2. ¿Cuál es la diferencia?

Cómo resolver los problemas con la inicialización

La respuesta corta es que el código debe usar el operador `new` para crear una nueva instancia de estructura. El listado 20.2 se compila si se crean nuevas referencias de estructura:

```

StructWithProperties PropertiesStruct = new
StructWithProperties();

```

Esta respuesta tiene que ver con el hecho de que las propiedades son implementadas por el compilador de C# como funciones públicas cuando genera el código del lenguaje intermedio de Microsoft (MSIL) para la aplicación. Puede comprobar esto revisando el MSIL generado por el compilador de C#. .NET Framework tiene una herramienta llamada ILDASM, que son las siglas de desensamblador de IL. Se puede usar esta herramienta para examinar el Lenguaje

intermedio (IL) y los metadatos de cualquier resultado binario compatible con el CLR de un compilador .NET.

Modifique el listado 20.2 para que incluya la nueva operación y compílelo en un ejecutable llamado `Listing20-2.exe`. Tras generar el ejecutable, puede observar su contenido con ILDASM. Use la siguiente línea de comando para iniciar el desensamblador IL con el ejecutable del listado 20.2:

```
ildasm Listing20-2.exe
```

La figura 20.1 muestra la ventana ILDASM abierta con el ejecutable `Listing20-2.exe`. El ejecutable contiene el manifiesto, que incluye la información de identificación para el ejecutable y también contiene metadatos que describen el código del ejecutable.

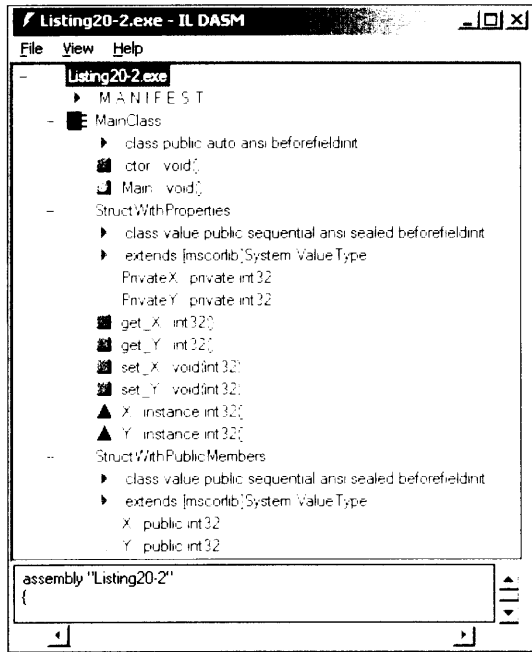


Figura 20.1. ILDASM abierto con el ejecutable del listado 20.2

ILDASM muestra las clases y estructuras del ejecutable en forma de árbol. La parte del árbol `StructWithPublicMembers` muestra dos variables públicas de tipo `int32` llamadas `X` e `Y`. Estas variables reflejan las dos propiedades programadas en la estructura. La parte del árbol `StructWithProperties` muestra las dos variables privadas; también muestra cuatro métodos que no se escribieron en la estructura:

- `int32 get_X()`
- `int32 get_Y()`

- `void set_X(int32)`
- `void set_Y(int32)`

Estos métodos realmente implementan el acceso a la propiedad de la estructura. El método `get_X()` contiene el código del descriptor de acceso `get` de la propiedad `X` y el método `get_Y()` contiene el código del descriptor de acceso `get` de la propiedad `Y`. Del mismo modo, el método `set_X()` contiene el código del descriptor de acceso `set` de la propiedad `X` y el método `set_Y()` contiene el código del descriptor de acceso `set` de la propiedad `Y`. Esto significa que cuando el código accede a una propiedad, en realidad está llamando a un método que implementa la funcionalidad de la propiedad.

El problema con el listado 20.2 es que la variable `PropertiesStruct` no está inicializada con un operador `new` antes de ser usada. Esto significa que la variable no está asociada a una instancia de estructura y los métodos no pueden ser llamados en instancias que no existen.

Las instrucciones de la propiedad del método `Main()` obligan a que se llame a los métodos de propiedades subyacentes, pero la llamada no encuentra ninguna instancia. El compilador de C# detecta este problema y emite el mensaje de error mostrado tras el listado 20.2.

La inicialización del miembro público tiene éxito porque las clases y las estructuras pueden inicializarse directamente, sin usar un constructor, siempre y cuando se haya dado explícitamente un valor a todas las variables de la instancia.

Clases derivadas

Cuando tratamos las clases de C# vimos cómo las clases pueden derivarse de otras clases. Las clases derivadas heredan la funcionalidad de su clase primaria o base. La relación entre una clase derivada y una clase base recibe el nombre de una relación "es un" en términos orientados a objetos. Por ejemplo, todas las clases de C# derivan en última instancia del tipo `System.Object` de .NET, de modo que se puede decir que la clase "es un" `System.Object`.

Como todas las clases derivadas heredan funcionalidad de su clase base, podemos presuponer que los objetos de una clase derivada pueden ser usados en cualquier lugar donde se pueda usar un objeto de la clase base de la clase. Sin embargo, la seguridad de tipo integrada en C# tiene preferencia y el código de este apartado explica este tema.

Cómo pasar clases derivadas

El listado 20.3 contiene una clase llamada `TestClass` que contiene un método llamado `Test()`. El método `Test()` acepta una referencia a un objeto como parámetro.

El listado 20.3 también contiene una clase llamada `MainClass`, que crea un objeto de la clase `TestClass` y una cadena. La cadena se usa como parámetro para la llamada al método `Test()` del objeto `TestClass`.

Listado 20.3. Cómo pasar cadenas donde se esperan objetos

```
public class TestClass
{
    public void Test(ref object ObjectReference)
    {
    }
}

public class MainClass
{
    public static void Main()
    {
        TestClass TestObject = new TestClass();
        string TestString = "Hello from C#!";

        TestObject.Test(ref TestString);
    }
}
```

El listado 20.3 no se compila. El compilador de C# emite los siguientes errores:

```
Listing20-3.cs(16,9): error CS1502: La mejor coincidencia de
metodo sobrecargado para TestClass.Test(ref object)' tiene
algunos argumentos no validos
```

```
Listing20-3.cs(16,29): error CS1503: Argumento '1': no se puede
convertir de 'ref string' a 'ref object'
```

A primera vista, no parece lógico. Como cada tipo de dato de .NET es un objeto, cualquier tipo de dato debería convertirse en última instancia en una variable de tipo `object`. Esto debería incluir a los objetos `string`. ¿Por qué no se puede convertir implícitamente el objeto `string` en una variable de tipo `object`?

Cómo resolver problemas que surgen cuando se pasan clases derivadas

El problema de compilación surge de las estrictas reglas de seguridad de tipos integradas en el lenguaje C#. El método `Test()` del listado 20.3 toma el valor de tipo `object` como parámetro y el compilador emite un error si se le proporciona algo distinto de un objeto. Otro problema con la estrategia del listado 20.3 procede del modificador de parámetro `ref` usado en el método `Test()`. El modifica-

El atributo `ref` permite que la implementación del método cambie el valor de la variable y el cambio es visible para el invocador. Esto le da permiso al método `Test()` para sobrescribir la variable `ObjectReference` con cualquier valor que pueda volver a convertirse explícitamente en un objeto. Observe la siguiente implementación alternativa del método `Test()`:

```
public void Test(ref object ObjectReference)
{
    TestClass TestObject;

    TestObject = new TestClass();
    ObjectReference = (object)TestObject;
}
```

En esta implementación, se crea un objeto de clase `TestClass`. La variable de referencia `ObjectReference` se asigna al nuevo objeto y el invocador ve esta asignación. El problema es que el listado 20.3 pasa una cadena al método `Test()`. Con esta nueva implementación del método `Test()` situado en el listado 20.3, el invocador podría pasar una cadena y recuperaría un objeto `TestClass`. El invocador podría no esperar que la variable cambiara a un tipo diferente del suministrado y podrían surgir numerosos problemas si el invocador siguiera trabajando con el código presuponiendo que la variable sigue siendo una cadena. C# evita este problema exigiendo que sólo se usen tipos de parámetros correctos cuando se invocan métodos.

Cómo usar no enteros como elementos de matriz

El lenguaje C# especifica que sólo se pueden usar enteros como elementos de matriz. Sin embargo, a veces se encontrará en situaciones en las que los enteros no son el modo más conveniente para expresar un índice de matriz en el código. Por ejemplo, imagine un tablero de ajedrez en el que una letra desde A hasta H hace referencia a una de las ocho columnas del tablero y un número de 1 a 8 hace referencia a una de las ocho filas del tablero. Si necesita representar un tablero de ajedrez en el código de C#, podría escoger usar una matriz rectangular de dos dimensiones:

```
int [,] Chessboard;

Chessboard = new int [8,8];
```

Tras inicializar la matriz del tablero de ajedrez, querrá hacer referencia a una casilla usando la tradicional sintaxis para las casillas del tablero de ajedrez. Quiérs quiera hacer referencia a la casilla B7, por ejemplo, de esta manera:

```
Chessboard['B', 7];
```

Sin embargo, no puede hacer esto porque C# no permite el uso de caracteres, como la B del ejemplo, como referencias de elementos de matriz.

El listado 20.4 usa un indizador para solucionar este problema:

Listado 20.4. Cómo representar un tablero de ajedrez con un indizador

```
using System;

public class Chessboard
{
    private int [,] Board;

    public Chessboard()
    {
        Board = new int[8,8];
    }

    public int this [char Column, int RowIndex]
    {
        get
        {
            int ColumnIndex;

            switch(Column)
            {
                case 'A':
                case 'a':
                    ColumnIndex = 0;
                    break;
                case 'B':
                case 'b':
                    ColumnIndex = 1;
                    break;
                case 'C':
                case 'c':
                    ColumnIndex = 2;
                    break;
                case 'D':
                case 'd':
                    ColumnIndex = 3;
                    break;
                case 'E':
                case 'e':
                    ColumnIndex = 4;
                    break;
                case 'F':
                case 'f':
                    ColumnIndex = 5;
                    break;
                case 'G':
                case 'g':
                    ColumnIndex = 6;
            }
        }
    }
}
```

```

        break;
    case 'H':
    case 'h':
        ColumnIndex = 7;
        break;
    default:
        throw new Exception("Illegal column specifier.");
    }
    Console.WriteLine("(returning cell [{0},{1}]",
ColumnIndex, RowIndex);
    return Board[ColumnIndex, RowIndex];
}
}

public class MainClass
{
    public static void Main()
    {
        int CellContents;
        Chessboard MyChessboard = new Chessboard();

        CellContents = MyChessboard ['B', 7];
    }
}

```

El código del listado 20.4 declara una clase llamada `Chessboard` para representar un tablero de ajedrez. La clase incluye una matriz de enteros privada de dos dimensiones llamada `Board`, que puede usarse para representar qué piezas de ajedrez están en qué casillas del tablero. (Para mantener el listado del código, la matriz no se usa realmente.)

La clase también implementa un indizador con un descriptor de acceso `get`. El indizador acepta dos parámetros: un carácter y un entero. El indizador da por hecho que el carácter especificado en el primer parámetro representa un identificador de columna y traduce el carácter como una columna de la matriz privada. El especificador de columna A lo traduce como columna 0 en la matriz privada `Board`. El especificador de columna B lo traduce como columna 1 en la matriz privada `Board` y así sucesivamente. El indizador escribe un mensaje en la consola que especifica los elementos de indizador traducidos y luego devuelve el valor de los elementos de la matriz a los que hacen referencia los parámetros del indizador.

El método `Main()` del listado 20.4 crea un nuevo objeto de tipo `Chessboard` y usa el indizador para hacer referencia a la casilla B7:

```
CellContents = MyChessboard ['B', 7];
```

Cuando el código del listado 20.4 se ejecuta, el código del método `Main()` llama al indizador, que escribe lo siguiente en la consola:

```
(returning cell [1,7])
```

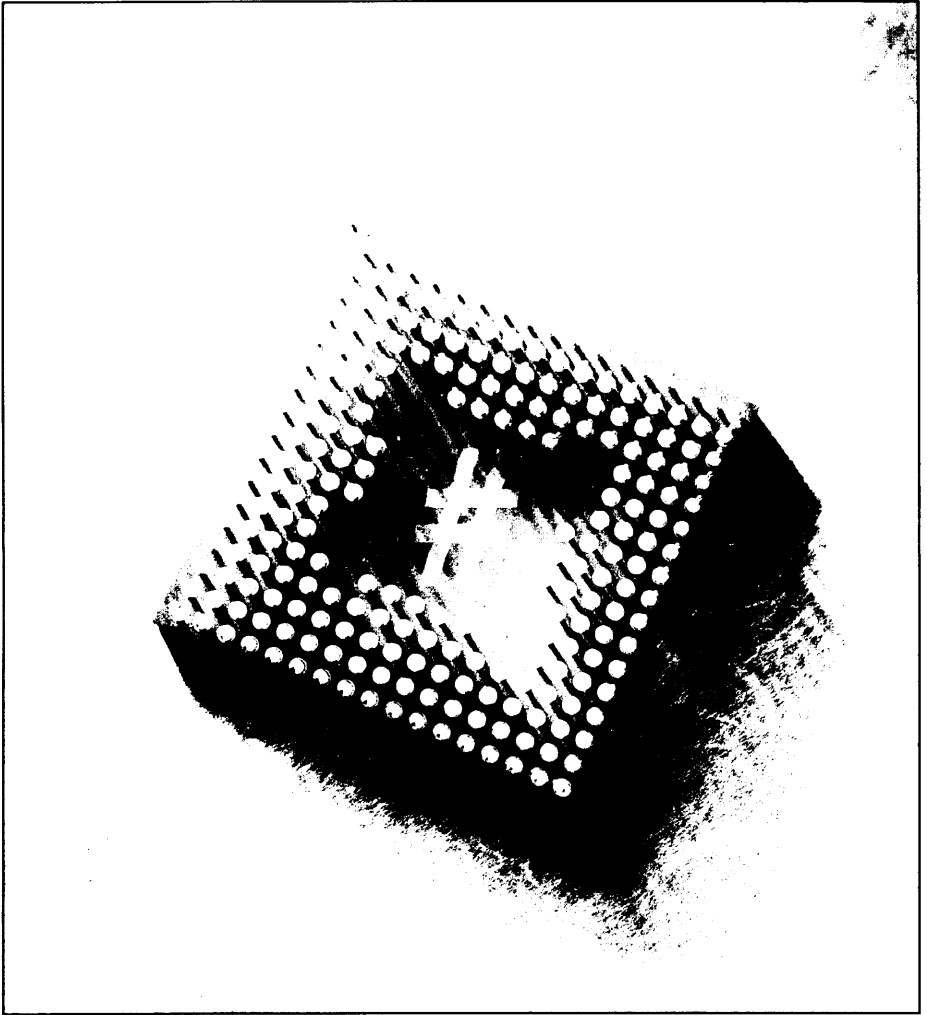
El indizador tradujo el primer parámetro, `B`, en la referencia de columna basada en enteros que el compilador de `C#` necesita para acceder a un elemento de la matriz privada. Este esquema permite diseñar clases que usan una sintaxis natural para la aplicación (en este caso, indizadores de elementos de matriz basados en caracteres) mientras aún cumplen los requisitos de `C#` de usar indizadores de elementos de matriz basados en enteros.

Resumen

El mejor modo de resolver un difícil problema de programación de `C#` es intentar diferentes métodos de codificación con el compilador. No tenga miedo de experimentar. El compilador de `C#` le avisará si hace algo equivocado.

También podría considerar el uso la herramienta `ILDASM` para colarse en sus ensamblados y ver cómo el compilador de `C#` crea código ejecutable a partir de su código fuente. La comprensión del código que genera el compilador de `C#` puede ayudarle a entender por qué el código funciona de la manera que lo hace. En este capítulo, aprendió que el código descriptor de acceso de propiedades se convierte en métodos especiales por el compilador de `C#`. Esto puede ser difícil de descubrir sin observar el código generado por `ILDASM`. La herramienta `ILDASM` funciona con cualquier ensamblado generado por un compilador de `.NET` que genere `MSIL`. Examine con `ILDASM` algunos ensamblados y aplicaciones de `.NET`. Ábralos con la herramienta y vea cómo están contruidos. El análisis de otros ensamblados y su contenido puede darle una mejor comprensión de cómo funciona el código `.NET` y puede ayudarle a ser un mejor programador `.NET`.

Parte IV
Cómo
desarrollar
soluciones .NET
usando C#



21 Cómo construir aplicaciones WindowsForms

La mayor parte del material escrito sobre .NET Framework se centra en la ayuda que reciben los programadores que escriben aplicaciones para Internet. El motor ASP.NET y los modelos de desarrollo del "software como un servicio" son, sin lugar a dudas, unas potentes herramientas para el desarrollo de aplicaciones de Internet. Sin embargo, .NET Framework no trata sólo con Internet.

Microsoft se dio cuenta de que, aunque muchos programadores están escribiendo aplicaciones para Internet, otros muchos se dedican a desarrollar aplicaciones de escritorio al estilo Win32. .NET Framework no ha olvidado a estos programadores. Incluye un conjunto de clases de .NET que facilita el desarrollo de aplicaciones Windows de escritorio que usan un lenguaje compatible con .NET. Este conjunto de clases y el modelo de programación que admite se llama WindowsForms.

En este capítulo estudiaremos la estructura básica de una aplicación WindowsForms. Verá cómo crear una forma básica y cómo añadir controles a los formularios.

Examinaremos las clases de .NET Framework que puede usar una aplicación WindowsForms y también estudiaremos algunos de los atributos de ensamblado que se pueden usar para agregar la información de versión y derecho de autoría a sus aplicaciones.

Arquitectura de WindowsForms

Para programar una aplicación WindowsForms es necesario comprender cómo se usan las clases base de WindowsForms con las aplicaciones .NET. Este capítulo examina la arquitectura de la biblioteca de clases de WindowsForms.

Todas las clases que se usan para construir aplicaciones de WindowsForms se incluyen en un espacio de nombre de .NET Framework llamado `System.Windows.Forms`. Este espacio de nombre contiene todas las clases necesarias para construir elaboradas aplicaciones de escritorio para Windows. Estas clases permiten trabajar con formularios, botones, controles de edición, casillas de verificación, listas y muchos otros elementos de interfaz de usuario. Todas estas clases están a su disposición listas para ser usadas en sus aplicaciones WindowsForms.

Las aplicaciones WindowsForms usan dos clases fundamentales de .NET Framework: La clase `Form`, que gestiona los formularios de la aplicación y los controles del formulario, y la clase `Application`, que gestiona el modo en que la aplicación controla los mensajes Windows enviados y recibidos de los formularios de la aplicación. Estas dos clases se incluyen en el espacio de nombre `System.Windows.Forms` de .NET Framework y componen la estructura básica de una aplicación WindowsForms.

La clase Form

El espacio de nombre `System.Windows.Forms` incluye una clase base llamada `Form`. La clase `Form` representa una forma o ventana de su aplicación. Al crear una aplicación de WindowsForms en C#, se diseña una clase de ventana y se usa la clase `Form` como clase base para la clase de ventana. Esta clase de ventana hereda todo el comportamiento básico de una ventana y agrega la funcionalidad necesaria para la aplicación. Todos los comportamientos básicos de ventana están integrados en la clase base `Form` y esos comportamientos se heredan automáticamente si se deriva una clase de ventana de la clase `Form`.

La clase Application

Las clases `Form` definen el aspecto y comportamiento de las ventanas de una aplicación pero no se ejecutan por sí mismas. WindowsForms debe ejecutarse dentro del contexto de una aplicación. El espacio de nombres `System.Windows.Forms` incluye una clase llamada `Application`, que contiene métodos que ayudan a gestionar las aplicaciones WindowsForms.

La clase `Application` contiene métodos que permiten iniciar, gestionar y detener las aplicaciones WindowsForms. WindowsForms responde a eventos iniciados por el usuario, como mover el ratón o pulsar una tecla del teclado.

Desde los comienzos de Windows, la arquitectura de las aplicaciones de escritorio de Windows ha sido diseñada sobre el concepto de *bucle de mensajes*.

En una aplicación Windows estándar, la parte más importante del código se incluye en un bucle y recibe mensajes del sistema operativo Windows. Cuando el usuario mueve el ratón, pulsa una tecla o realiza alguna otra operación sobre la que puede actuar una ventana, el sistema operativo toma nota de la acción y envía un mensaje a la ventana correspondiente informándole de la acción. Es el código de la ventana el que debe utilizar convenientemente el mensaje.

En la arquitectura WindowsForms, los conceptos básicos siguen siendo los mismos. Las aplicaciones WindowsForms tienen un fragmento de código que espera la llegada de mensajes de interacción del usuario desde el sistema operativo y luego los envían a la ventana apropiada. En la arquitectura WindowsForms, la clase `Application` es el código principal que gestiona este control de mensajes y la clase `Forms` es la clase que controla los mensajes que envía la clase `Application`.

La clase `Application` está sellada. No se pueden derivar clases de ella. Sus métodos son estáticos, lo que significa que pueden ser llamados sin que haya un objeto de la clase `Application` disponible.

Cómo crear la primera aplicación WindowsForms

El listado 21.1 muestra una sencilla aplicación WindowsForms. Representa una sencilla aplicación WindowsForms que abre un sólo formulario y sigue ejecutándose hasta que el usuario cierra el formulario.

Listado 21.1. La aplicación Hello, WindowsForms

```
using System.Windows.Forms;

public class SimpleHelloWorld : Form
{
    public static void Main()
    {
        Application.Run(new SimpleHelloWorld());
    }

    public SimpleHelloWorld()
    {
        Text = "Hello, WindowsForms!";
    }
}
```

El código del listado 21.1 declara una clase llamada `SimpleHelloWorld`. Se deriva de la clase `Form` de .NET Framework, que califica la clase como una

clase `Windows Form`. La clase contiene un método `Main()`, que crea un nuevo objeto de la clase `SimpleHelloWorld` y usa ese objeto como parámetro del método en la clase `Application` llamado `Run()`. El constructor de la clase `SimpleHelloWorld` establece el valor de la propiedad heredada `Text` a `Hello, WindowsForms!`. Esta cadena se muestra como el título del formulario. La figura 21.1 muestra el aspecto del formulario cuando se ejecuta el código del listado 21.1.

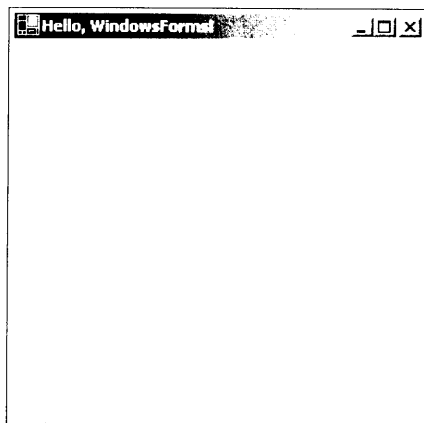


Figura 21.1. El listado 21.1 muestra un sencillo formulario

Cómo compilar una aplicación WindowsForms

Al igual que las aplicaciones de consola diseñadas usando `C#`, las aplicaciones `WindowsForms` deben ser compiladas con el compilador de `C#` antes de poder ejecutar su código. Las aplicaciones `WindowsForms` compiladas con el compilador de `C#` pueden comportarse de forma ligeramente diferente al ser iniciadas, dependiendo del modo en el que se compiló el código. La siguiente línea de comando es la más simple que se puede usar para compilar el código del listado 21.1:

```
csc Listing21-1.cs
```

Como todas las aplicaciones de `C#`, este código genera un ejecutable llamado `Listing21-1.exe`. Si ejecuta este archivo desde una consola, aparecerá la ventana de la aplicación. Sin embargo, ocurre algo interesante cuando se hace doble clic sobre el icono del ejecutable en el explorador de `Windows`. Si se inicia el ejecutable desde el `Explorador de Windows` aparecen dos ventanas: una ventana de consola, que está vacía y la ventana `WindowsForms` de la aplicación. Este comportamiento es diferente de la mayoría de las aplicaciones `Windows`. La mayoría de las aplicaciones `Windows` no presentan una ventana de consola vacía

cuando se inicia la aplicación. ¿Por qué el ejecutable del listado 21.1 muestra una ventana de consola y, lo más importante, cómo podemos eliminarla?

Por defecto, el compilador de C# genera aplicaciones de consola. Estas aplicaciones necesitan una ventana de consola para que los métodos, como `Console.WriteLine()`, dispongan de una consola en la que escribir sus mensajes. El tiempo de ejecución de .NET consulta el ejecutable cuando es iniciado y si el ejecutable está compilado como una aplicación de consola, el cargador del tiempo de ejecución crea una nueva ventana de consola para la aplicación. Si se compila el listado 21.1 mediante el compilador de C# por defecto, se generará una aplicación .NET para consola. Esto explica la ventana de consola vacía que aparece cuando se ejecuta el ejecutable compilado.

WindowsForms trabaja con ventanas, no con consolas y las aplicaciones WindowsForms no necesitan la ventana de consola. El compilador de C# admite un argumento de línea de comando llamado `target` que se usa para especificar el tipo de la aplicación que se quiere compilar. Puede usar este argumento para indicarle al compilador de C# que quiere compilar una aplicación Windows que no necesita una consola. La siguiente línea de comando:

```
csc /target:winexe Listing21-1.cs
```

ordena al compilador de C# que compile el archivo `Listing21-1.cs` y use sus contenidos para crear un ejecutable Windows que no necesita una consola. Si usa esta línea de comando para compilar el código del listado 21.1 e inicia el ejecutable desde el explorador de Windows, aparecerá el formulario de la aplicación, sin que aparezca también una ventana de consola vacía.

NOTA: El compilador de C# acepta `/t` como abreviatura de `/target`. La anterior línea de comandos puede acortarse a `csc /t:winexe Listing21-1.cs`.

Ensamblados: cómo añadir información de versión a las aplicaciones WindowsForms

Por defecto, las aplicaciones `WindowsForms` no contienen información de su versión. Sin embargo, se pueden usar atributos integrados en .NET Framework para añadir información de la versión a las aplicaciones. Estos atributos empiezan a actuar en el nivel de ensamblado y se añaden al código que produce el compilador de C#. Todos estos atributos son opcionales, pero su uso añade información de versión y de derechos de autoría a los binarios .NET. Al añadir esta información se permite a los usuarios finales usar el Explorador de Windows, hacer clic con el botón derecho en la aplicación, seleccionar **Propiedades** en el menú contextual e inspeccionar los valores del atributo incrustado. Los usuarios

finales pueden examinar la información de la versión y de derechos de autoría de las aplicaciones .NET WindowsForms exactamente del mismo modo que pueden examinar dicha información en las aplicaciones Win32 estándar.

Las clases que implementan estos atributos están en un espacio de nombre .NET llamado `System.Reflection`. Cuando se usan estos atributos, hay que hacer referencia a este espacio de nombre con la palabra clave `using` o anteponer el nombre del espacio de nombre a los nombres de los ensamblados.

Parte de la información especificada en estos atributos se escribe en el manifiesto del ensamblado y otros fragmentos de información se almacenan como recurso de información de versión incrustado en el ejecutable del ensamblado. Puede observar el manifiesto del ensamblado mediante la herramienta ILDASM y puede ver el recurso de información de versión del ejecutable haciendo clic con el botón derecho del ratón en el archivo en el Explorador de Windows y seleccionando `Propiedades>Versión`.

TRUCO: Puede agregar estos atributos a cualquier proyecto de código .NET. La información de la versión siempre está disponible en el producto compilado del código.

En este apartado estudiaremos los siguientes atributos:

- `AssemblyTitle`, que asigna un título al ensamblado.
- `AssemblyDescription`, que asigna una descripción al ensamblado.
- `AssemblyConfiguration`, que describe las opciones usadas para construir el ensamblado.
- `AssemblyCompany`, que asigna un nombre de compañía al ensamblado.
- `AssemblyProduct`, que asigna información del producto al ensamblado.
- `AssemblyCopyright`, que asigna información de derechos de autoría al ensamblado.
- `AssemblyTrademark`, que asigna información de marca al ensamblado.
- `AssemblyCulture`, que asigna información local al ensamblado.
- `AssemblyVersion`, que asigna un número de versión al ensamblado.

AssemblyTitle

El atributo `AssemblyTitle` permite asignar un título al ensamblado. El atributo toma un parámetro de cadena en su constructor que especifica el título, como muestra el siguiente ejemplo:

```
[assembly: AssemblyTitle("Listing 21-2")]
```

El título del ensamblado no se escribe en su manifiesto, pero está disponible en el campo `Description` del bloque de información de versión del archivo compilado.

AssemblyDescription

El atributo `AssemblyDescription` permite proporcionar una descripción del ensamblado. El atributo toma un parámetro de cadena en su constructor que especifica la descripción, como muestra el siguiente ejemplo:

```
[assembly: AssemblyDescription("This executable was produced by  
compiling the code in Listing 21-2.")]
```

La descripción del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo `Comment` del bloque de información de versión del archivo compilado.

AssemblyConfiguration

El atributo `AssemblyConfiguration` permite especificar información de configuración de compilación del ensamblado. Por ejemplo, la información de configuración de ensamblado puede especificar si el ensamblado se compiló con configuración para su distribución o depuración. El atributo toma un parámetro de cadena en su constructor que especifica la información de configuración:

```
[assembly: AssemblyConfiguration("retail")]
```

La descripción del ensamblado se escribe en el manifiesto del ensamblado, pero no está disponible en el bloque de información de versión del archivo compilado.

AssemblyCompany

El atributo `AssemblyCompany` permite especificar un nombre de compañía para asociarlo al ensamblado. El atributo toma un parámetro de cadena en su constructor que especifica el nombre de la compañía:

```
[assembly: AssemblyCompany("John Wiley, Inc.")]
```

El nombre de la compañía del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo `Company Name` del bloque de información de versión del archivo compilado.

AssemblyProduct

El atributo `AssemblyProduct` permite especificar un nombre de producto para asociarlo al ensamblado. El atributo toma un parámetro de cadena en su

constructor que especifica el nombre del producto compañía, como muestra el siguiente ejemplo:

```
[assembly: AssemblyProduct("C# Bible")]
```

El nombre del producto del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo `Product Name` del bloque de información de versión del archivo compilado.

AssemblyCopyright

El atributo `AssemblyCopyright` permite especificar información de derechos de autoría para el ensamblado. El atributo toma un parámetro de cadena en su constructor que especifica la información de derechos de autoría, como muestra el siguiente ejemplo:

```
[assembly: AssemblyCopyright("(c) 2002 John Wiley, Inc.")]
```

El nombre del producto del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo `Copyright` del bloque de información de versión del archivo compilado.

AssemblyTrademark

El atributo `AssemblyTrademark` permite especificar información de marca registrada para el ensamblado. El atributo toma un parámetro de cadena en su constructor que especifica la información de marca registrada, como muestra el siguiente ejemplo:

```
[assembly: AssemblyTrademark("Windows is a trademark of  
Microsoft Corporation.")]
```

El nombre del producto del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo `Legal` del bloque de información de versión del archivo compilado.

AssemblyCulture

El atributo `AssemblyCulture` permite especificar información de referencia cultural para el ensamblado.

La información de referencia cultural especifica la información de lenguaje y país que el ensamblado usa para hacer su trabajo. El atributo toma un parámetro de cadena en su constructor que especifica la información de referencia cultural, como muestra el siguiente ejemplo:

```
[assembly: AssemblyCulture("us-en")]
```

Las cadenas de referencia cultural son definidas mediante un estándar de Internet llamado RFC1766. El estándar se titula *Tags for the Identification of Languages* y está disponible en Internet en www.ietf.org/rfc/rfc1766.txt. El nombre del producto del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en el campo *Legal Trademarks* del bloque de información de versión del archivo compilado.

NOTA: La información de referencia cultural sólo se puede añadir a bibliotecas y módulos. No puede añadirse a ejecutables, porque los ejecutables no pueden ser localizados. Si se intenta añadir el atributo `AssemblyCulture` al código base que se compila en el ejecutable final, el compilador de C# emite el siguiente mensaje de error:

```
error CS0647: Error al emitir el atributo
'System.Reflection.AssemblyCultureAttribute'-'Los archivos
ejecutables no se pueden adaptar y no deben tener referencia
cultural'
```

La descripción del ensamblado se escribe en el manifiesto del ensamblado, pero no está disponible en el bloque de información de versión del archivo compilado.

AssemblyVersion

El atributo `AssemblyVersion` permite asignar un número de versión al ensamblado. Los números de versión de .NET constan de cuatro partes:

- Un número de versión principal
- Un número de versión secundaria
- Un número de revisión
- Un número de compilación

Cada una de estas partes está separada por un punto. El atributo toma un parámetro de cadena en su constructor que especifica el número de versión, como muestra el siguiente ejemplo:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Siempre se debe especificar un número de versión. Si se especifican los números de versión principal y secundaria, puede dejar que el compilador de C# genere automáticamente los otros números al compilar el código. Esto puede ser útil si se quiere que cada compilación del código tenga un número de versión único. Si usa el carácter asterisco para un número de compilación, el compilador de C# asigna

uno automáticamente. El número de compilación generado es igual al número de días desde el 1 de enero del 2000, como muestra el siguiente ejemplo:

```
[assembly: AssemblyVersion("1.0.0.*")]
```

Este ejemplo asigna al ensamblado una versión principal igual a 1, una versión secundaria igual a 0, un número de revisión igual a 0 y un número de compilación asignado automáticamente por el compilador de C#.

Si usa el carácter asterisco para un número de revisión, el compilador de C# asigna automáticamente un número de revisión y un número de compilación. El número de revisión generado es igual al número de días desde el 1 de enero del 2000, como muestra el siguiente ejemplo:

```
[assembly: AssemblyVersion("1.0.*")]
```

Este ejemplo asigna al ensamblado una versión principal igual a 1, una versión secundaria igual a 0, un número de revisión igual a 0 y un número de revisión asignado automáticamente por el compilador de C#.

El nombre del producto del ensamblado se escribe en el manifiesto del ensamblado; también está disponible en los campos `AssemblyVersion`, `ProductVersion` y `FileVersion` del bloque de información de versión del archivo compilado. El listado 21.2 añade atributos de versión de ensamblado al código del listado 21.1.

Listado 21.2. Sencilla aplicación de formulario Windows con información de versión

```
using System.Reflection;
using System.Windows.Forms;

[assembly: AssemblyTitle("Listing 21-2")]
[assembly: AssemblyDescription("This executable was produced by
compiling the code in Listing 21-2.")]
[assembly: AssemblyConfiguration("Retail")]
[assembly: AssemblyCompany("John Wiley, Inc.")]
[assembly: AssemblyProduct("C# Bible")]
[assembly: AssemblyCopyright("(c) 2002 John Wiley, Inc.")]
[assembly: AssemblyVersion("1.0.*")]

public class SimpleHelloWorld : Form
{
    public static void Main()
    {
        Application.Run(new SimpleHelloWorld());
    }

    public SimpleHelloWorld()
    {
        Text = "Hello, WindowsForms!";
    }
}
```

El objeto Application con más detalle

El objeto `Application` contiene propiedades, métodos y eventos que pueden usarse en el código `WindowsForms`. En este apartado estudiaremos los miembros de clase más usados.

Eventos Application

La clase `Application` admite cuatro eventos que pueden usarse en las aplicaciones `WindowsForms`:

- El evento `ApplicationExit` se desencadena cuando la aplicación está a punto de cerrarse. A este evento se le pueden asignar los delegados de tipo `EventHandler`. El delegado `EventHandler` está definido en el espacio de nombre `System` de .NET. El delegado toma dos parámetros: un objeto que hace referencia al objeto que envía el evento y un objeto `EventArgs` que especifica los argumentos del evento. No devuelve ningún valor.
- El evento `Idle` se desencadena cuando la aplicación termina de enviar mensajes desde el sistema operativo y está a punto de entrar en estado de inactividad. La aplicación abandona el estado de inactividad cuando considera que hay más mensajes que procesar. A este evento se le pueden asignar delegados de tipo `EventHandler`. El delegado `EventHandler` está definido en el espacio de nombre `.NET System`. El delegado toma dos parámetros: un objeto que hace referencia al objeto que envía el evento y un objeto `EventArgs` que especifica los argumentos del evento. No devuelve ningún valor.
- El evento `ThreadException` se desencadena cuando se inicia una excepción de subproceso no capturada. A este evento se le pueden asignar delegados de tipo `ThreadExceptionHandler`. El delegado `ThreadExceptionHandler` se define en el espacio de nombre `.NET System.Threading`. El delegado toma dos parámetros: un objeto que hace referencia al objeto que envía el evento y un objeto `ThreadingExceptionEventArgs` que especifica los argumentos del evento. No devuelve ningún valor.
- El evento `ThreadExit` se desencadena cuando un subproceso está a punto de cerrarse. Cuando el proceso principal de una aplicación está a punto de cerrarse, antes se desencadena este evento, seguido de un evento `ApplicationExit`. A este evento se le pueden asignar delegados de tipo `EventHandler`. El delegado `EventHandler` está definido en el espacio de nombre `System` de .NET. El delegado toma dos parámetros: un objeto que hace referencia al objeto que envía el evento y un objeto

EventArgs que especifica los argumentos del evento. No devuelve ningún valor.

Cómo trabajar con eventos en el código

El código del listado 21.3 se suma al código de formulario del listado 21.2 mediante el control de eventos desde el objeto Application.

Listado 21.3. Cómo controlar los eventos de la aplicación

```
using System;
using System.Threading;
using System.Reflection;
using System.Windows.Forms;

[assembly: AssemblyTitle("Listing 21-3")]
[assembly: AssemblyDescription("This executable was produced by
compiling the code in Listing 21-3.")]
[assembly: AssemblyCompany("John Wiley, Inc.")]
[assembly: AssemblyProduct("C# Bible")]
[assembly: AssemblyCopyright("(c) 2002 John Wiley, Inc.")]
[assembly: AssemblyVersion("1.0.*")]

public class HelloWorldForm : Form
{
    public HelloWorldForm()
    {
        Text = "Hello, WindowsForms!";
    }
}

public class ApplicationEventHandlerClass
{
    public void OnApplicationExit(object sender, EventArgs e)
    {
        try
        {
            Console.WriteLine("The application is shutting down.");
        }
        catch(NotSupportedException)
        {
        }
    }

    public void OnIdle(object sender, EventArgs e)
    {
        Console.WriteLine("The application is idle.");
    }

    public void OnThreadException(object sender,
ThreadExceptionEventArgs e)
    {

```

```

        Console.WriteLine("an exception thrown from an application
thread was caught!");
    }

    public void OnThreadExit(object sender, EventArgs e)
    {
        Console.WriteLine("The application's main thread is
shutting down.");
    }
}

public class MainClass
{
    public static void Main()
    {
        HelloWorldForm FormObject = new HelloWorldForm();
        ApplicationEventHandlerClass AppEvents = new
ApplicationEventHandlerClass();

        Application.ApplicationExit += new
EventHandler(AppEvents.OnApplicationExit);
        Application.Idle += new EventHandler(AppEvents.OnIdle);
        Application.ThreadException += new
ThreadExceptionHandler(AppEvents.OnThreadException);
        Application.ThreadExit += new
EventHandler(AppEvents.OnThreadExit);
        Application.Run(FormObject);
    }
}

```

La nueva clase del listado 21.3 recibe el nombre de `ApplicationEventHandlerClass` y contiene métodos que controlan los eventos desencadenados desde el objeto `Application`. El método `Main()` crea un objeto de la clase `ApplicationEventHandlerClass` y agrega su código a la lista de controladores de eventos de la clase `Application`. Los controladores de evento de la clase `ApplicationEventHandlerClass` escriben en la consola. Esto es perfectamente válido, incluso en una aplicación `WindowsForms`. Si compila el código del listado 21.3 usando el destino ejecutable de consola (`csc /target:exe Listing21-3.cs`), los mensajes del controlador de eventos se escriben en la ventana de la consola que se usó para iniciar la aplicación. Si compila el código del listado 21.3 usando el destino de ejecutable de `Windows` (`csc /target:winexe Listing21-3.cs`), ninguna consola está asociada al proceso y no se muestran mensajes en la consola.

Propiedades Application

La clase `Application` es compatible con varias propiedades que pueden usarse en las aplicaciones de C#. Los siguientes apartados describen cómo usar cada una de estas propiedades.

AllowQuit

La propiedad booleana `AllowQuit` especifica si el código puede o no terminar la aplicación mediante programación. La propiedad devuelve `True` si el código puede ordenar a la aplicación que termine y `False` en caso contrario. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Por lo general, los usuarios pueden terminar las aplicaciones cerrando el formulario principal de la aplicación. Algunas aplicaciones se incluyen en contenedores, como clientes Web, y no pueden cerrarse mediante programación. Sólo se cierran cuando su contenedor se cierra. Si la propiedad `AllowQuit` devuelve `true`, se puede llamar a un método de `Application` llamado `Exit()` para terminar la aplicación mediante programación. Los métodos del objeto `Application` se estudiarán en el apartado titulado "Métodos `Application`."

CommonAppDataRegistry

La propiedad `CommonAppDataRegistry` devuelve una referencia a un objeto de la clase `RegistryKey`. El objeto `RegistryKey` hace referencia a una clave en el registro que la aplicación puede usar para almacenar datos de registro que deberían estar disponibles para todos los usuarios de la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor.

La clase `RegistryKey` es parte de .NET Framework y está disponible en un espacio de nombre llamado `Microsoft.Win32`. Representa una clave específica del registro y contiene métodos que permiten crear subclaves, leer valores y realizar otras tareas relacionadas con las claves del registro.

CommonAppDataPath

La propiedad de cadena `CommonAppDataPath` hace referencia a una ruta en el sistema de archivos que la aplicación puede usar para almacenar datos basados en archivos que deberían estar disponibles para todos los usuarios de la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor.

La ruta de datos de la aplicación se almacena dentro de la ruta de carpeta de documentos de Windows para todos los usuarios, que suele encontrarse en `C:\Documents and Settings\All Users\Datos de programa`. La ruta real de la aplicación apunta a una carpeta dentro de esta ruta de documentos "*all users*" que tiene la forma `CompanyName\ ProductName\ ProductVersion`. Los nombres de carpeta `CompanyName`, `ProductName` y `ProductVersion` se basan en los valores de las propiedades de la aplicación del mismo nombre. Si no se asignan valores a estas propiedades, la clase `Application` proporciona unos valores por defecto válidos. Por ejemplo, el código del listado 21.1 tiene una ruta de datos comunes de la aplicación `C:\Documents and Settings\All Users\Datos de programa\SimpleHelloWorld\SimpleHelloWorld\V0.0`. Si el código del listado 21.1 va a asignar valores a las propiedades `CompanyName`,

`ProductName` o `ProductVersion` de la clase `Application`. los nombres de carpetas en la ruta de datos comunes de la aplicación puede cambiar para reflejar los valores de esas propiedades.

La ruta de la carpeta de la versión del producto sólo usa los números de versión principal y secundario especificados en la aplicación. independientemente del número de valores que asigne en el atributo de la aplicación [`AssemblyVersion`]. Si la aplicación usa un atributo [`AssemblyVersion`] con un valor, por ejemplo, 1.2.3.4, la parte de número de versión de la ruta de datos comunes de la aplicación será 1.2. La letra `V` siempre se antepone al número de versión principal de la aplicación en la parte de la versión de número de la ruta de datos.

CompanyName

La propiedad de cadena `CompanyName` devuelve el nombre de la compañía asociado a la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Por defecto, este valor se asigna al nombre de la clase que contiene el método `Main()` de la aplicación. Si la aplicación especifica un nombre de compañía con el atributo [`AssemblyCompany`], el valor de ese atributo se usa como el valor de la propiedad `CompanyName` de la aplicación.

CurrentCulture

La propiedad `CurrentCulture` permite trabajar con la información de referencia cultural de la aplicación. Esta propiedad se puede leer y escribir. La propiedad tiene una clase de tipo `CultureInfo`, que es una clase definida por .NET Framework y que se encuentra en el espacio de nombres `System.Globalization`. La clase `CultureInfo` contiene métodos y propiedades que permiten al código trabajar con datos específicos del entorno cultural en el que se ejecuta la aplicación. Los objetos `CultureInfo` ofrecen información como el formato preferido para mostrar la fecha y la hora, la configuración de la hora y el formato numeral.

CurrentInputLanguage

La propiedad `CurrentInputLanguage` permite trabajar con el idioma actual de la aplicación. La propiedad tiene una clase de tipo `InputLanguage`, que es una clase definida por .NET Framework y que se encuentra en el espacio de nombres `System.Windows.Forms`.

La clase `InputLanguage` contiene métodos y propiedades que permiten al código trabajar con el conocimiento que tenga la aplicación de las teclas del teclado y cómo se relacionan con los caracteres que pueden introducirse en la aplicación. Las diferentes versiones específicas de cada lenguaje de Windows establecen equivalencias entre las teclas del teclado y los caracteres específicos de los diferentes lenguajes y la clase `CurrentInputLanguage` especifica el aspecto de esta asignación.

ExecutablePath

La propiedad de cadena `ExecutablePath` devuelve la ruta del ejecutable de la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor.

LocalUserAppDataPath

La propiedad de cadena `LocalUserAppDataPath` hace referencia a una ruta en el sistema de archivos que la aplicación puede usar para almacenar datos basados en archivos que deberían estar disponibles para el usuario conectado en ese momento al equipo.

Esta propiedad es de sólo lectura y no se le puede asignar un valor. Es usada por los usuarios locales con perfiles de sistema operativo del equipo local. Los usuarios con perfiles móviles usados a través de un sistema de redes tienen una propiedad distinta, llamada `UserAppDataPath`, para especificar dónde deben almacenarse los datos de la aplicación.

Al igual que la propiedad `CommonAppDataPath`, la ruta de datos de usuario local señala a una carpeta incluida dentro de la carpeta de documentos de usuario conectada con una estructura de carpetas que tiene la forma `CompanyName\ ProductName\ ProductVersion`. Los nombres de carpeta `CompanyName`, `ProductName` y `ProductVersion` se basan en los valores de la aplicación del mismo nombre. Si no se asigna un valor a estas propiedades, la clase `Application` proporciona unos valores válidos por defecto. Si el código asigna un valor a las propiedades `CompanyName`, `ProductName` o `ProductVersion` de la clase `Application`, los nombres de carpetas en la ruta de datos de la aplicación de usuario local cambia para reflejar los valores de esas propiedades.

Al igual que la ruta de datos comunes de la aplicación, la ruta de carpeta de versión de producto sólo usa los números de versión principal y secundario especificados en la aplicación, independientemente del número de valores especificados en el atributo de la aplicación [`AssemblyVersion`]. Si la aplicación usa un atributo [`AssemblyVersion`] con un valor, por ejemplo, `1.2.3.4`, la parte correspondiente al número de versión de la ruta de datos de usuario local de la aplicación será `1.2`. La letra `V` siempre se antepone al número de versión principal de la aplicación en la parte de la versión de número de la ruta de datos.

MessageLoop

La propiedad booleana `MessageLoop` devuelve `True` si existe un bucle de mensajes para la aplicación y `False` en caso contrario. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Como todas las aplicaciones `WindowsForms` necesitan un bucle de mensajes para que los mensajes de `Windows` puedan ser enviados al formulario correcto, las aplicaciones `WindowsForms` devuelven `True` para esta propiedad.

ProductName

La propiedad de cadena `ProductName` devuelve el nombre del producto asociado a la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Por defecto, a este valor se le asigna el nombre de la clase que contiene el método `Main()` de la aplicación. Si la aplicación especifica un nombre de producto con el atributo `[AssemblyProduct]`, el valor de ese atributo se usa como el valor de la propiedad de la aplicación `ProductName`.

ProductVersion

La propiedad de cadena `ProductVersion` devuelve el número de versión asociado a la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Por defecto, este valor es `0.0.0.0`. Si la aplicación especifica un número de versión con el atributo `[AssemblyVersion]`, el valor de ese atributo se usa como el valor de la propiedad de la aplicación `ProductVersion`.

SafeTopLevelCaptionFormat

La propiedad de cadena `SafeTopLevelCaptionFormat` hace referencia a la cadena de formato que el tiempo de ejecución aplica a los títulos de la ventana de nivel superior cuando las aplicaciones se ejecutan desde un contexto no seguro.

La seguridad es una parte integral de .NET Framework y el entorno común de ejecución (CLR). El CLR respeta la configuración de las diferentes zonas de seguridad en Internet Explorer (Internet, Intranet local, Sitios de confianza y Sitios restringidos) y restringe los servicios de tiempo de ejecución para las aplicaciones que se ejecutan en zonas no fiables. Las aplicaciones `WindowsForms` que se ejecutan desde zonas no fiables, como la zona Internet, tienen una etiqueta de aviso que describe la aplicación como procedente de una localización no fiable. El texto de esta etiqueta de aviso está basado en la plantilla de formato de cadena almacenada en la propiedad `SafeTopLevelCaptionFormat`.

StartupPath

La propiedad de cadena `StartupPath` devuelve la ruta al archivo ejecutable que inició la aplicación. Esta propiedad sólo devuelve la ruta. No incluye el nombre del archivo ejecutable. Esta propiedad es de sólo lectura y no se le puede asignar un valor.

UserAppDataPath

La propiedad de cadena `UserAppDataPath` hace referencia a una ruta en el sistema de archivos que puede usar la aplicación para almacenar datos basados en archivos que deberían estar disponibles para el usuario de red que está conectado en ese momento al equipo local. Esta propiedad es de sólo lectura y no se le puede asignar un valor. Es usada por los usuarios locales con perfiles de sistema operativo en la red. Los usuarios que tengan perfiles de equipo local no utilizados

en la red usan una propiedad distinta, llamada `LocalUserAppDataPath`, para especificar dónde deben almacenarse los datos de la aplicación.

Al igual que la propiedad `CommonAppDataPath`, la ruta de datos de usuario local señala a una carpeta incluida dentro de la carpeta de documentos de usuario conectado con una estructura de carpetas que tiene la forma `CompanyName\ ProductName\ ProductVersion`. Los nombres de carpeta `CompanyName`, `ProductName` y `ProductVersion` se basan en los valores de la aplicación del mismo nombre. Si no se asigna un valor a estas propiedades, la clase `Application` proporciona unos valores válidos por defecto. Si el código asigna un valor a las propiedades `CompanyName`, `ProductName` o `ProductVersion` de la clase `Application`, los nombres de carpetas en la ruta de datos de la aplicación de usuario local cambia para reflejar los valores de esas propiedades.

UserAppDataRegistry

El método `UserAppDataRegistry` devuelve una referencia a un objeto de clase `RegistryKey`. Al igual que la propiedad `CommonAppDataRegistry`, la propiedad devuelve un objeto de tipo `RegistryKey`. El objeto `RegistryKey` devuelto hace referencia a una clave en el registro que la aplicación puede usar para almacenar datos de registro que sólo deberían estar disponibles para el usuario actual de la aplicación. Esta propiedad es de sólo lectura y no se le puede asignar un valor.

Métodos Application

La clase `Application` admite ocho métodos que pueden ser llamados desde las aplicaciones de C#. Estos métodos se describen en las siguientes secciones.

AddMessageFilter

El método `AddMessageFilter()` añade un filtro de mensajes a la aplicación para controlar los mensajes de Windows mientras los mensajes son enviados a sus destinos. El filtro de mensajes que se instala en la aplicación recibe los mensajes de Windows antes de que se envíen al formulario. Un filtro de mensajes instalado por `AddMessageFilter()` puede controlar un mensaje que se le envíe y puede decidir si el mensaje debe enviarse al formulario.

El listado 21.4 muestra cómo se puede usar el filtro de mensajes en una aplicación `WindowsForms`. El controlador de mensajes busca mensajes que anunciar cuando se hace clic con el botón izquierdo en el formulario.

Listado 21.4. Cómo instalar un filtro de mensajes

```
using System;
using System.Windows.Forms;
```

```

public class BlockLeftMouseButtonMessageFilter : IMessageFilter
{
    const int WM_LBUTTONDOWN = 0x201;
    const int WM_LBUTTONUP = 0x202;

    public bool PreFilterMessage(ref Message m)
    {
        if(m.Msg == WM_LBUTTONDOWN)
        {
            Console.WriteLine("The left mouse button is down.");
            return true;
        }
        if(m.Msg == WM_LBUTTONUP)
        {
            Console.WriteLine("The left mouse button is up.");
            return true;
        }
        return false;
    }
}

public class MainForm : Form
{
    public static void Main()
    {
        MainForm MyForm = new MainForm();
        BlockLeftMouseButtonMessageFilter MsgFilter = new
BlockLeftMouseButtonMessageFilter();

        Application.AddMessageFilter(MsgFilter);
        Application.Run(MyForm);
    }

    public MainForm()
    {
        Text = "Message Filter Test";
    }
}

```

El método `AddMessageFilter()` recibe un argumento: una implementación de una interfaz llamada `IMessageFilter`. La interfaz `IMessageFilter` es definida por .NET Framework y se incluye en el espacio de nombres `System.Windows.Forms`. `IMessageFilter` declara un método:

```
public bool PreFilterMessage(ref Message m);
```

El método `PreFilterMessage()` toma como entrada una referencia a una instancia de una estructura llamada `Message`. La estructura `Message` describe un mensaje de Windows y contiene las siguientes propiedades:

- `HWND`, que describe el controlador de la ventana que debe recibir el mensaje.

- `LParam`, que describe un fragmento de número entero que se envía con el mensaje.
- `Msg`, que describe el número entero ID asociado al mensaje. Cada mensaje de Windows tiene su propio ID entero.
- `Result`, que describe el valor que debe devolverse a Windows en respuesta al control del mensaje.
- `WParam`, que describe otro fragmento de número entero que se envía con el mensaje.

El listado 21.4 comienza declarando una clase llamada `BlockLeftMouseButtonMessageFilter`. Esta clase implementa la interfaz `IMessageFilter`. La implementación del método `PreFilterMessage()` de la clase comprueba el ID del mensaje pasado al método. Comprueba si el ID indica que se ha pulsado el botón izquierdo del ratón. En caso afirmativo, se escribe en la consola el mensaje `The left mouse button is down`. A continuación comprueba si el ID indica que se ha soltado el botón izquierdo del ratón. En caso afirmativo, se escribe en la consola el mensaje `The left mouse button is up`.

NOTA: La clase `BlockLeftMouseButtonMessageFilter` declara constantes para dar nombres a los mensajes de Windows que busca el filtro. Los nombres empiezan con `WM` (por Mensaje de Windows) y coinciden con los nombres definidos por Microsoft. Todos los mensajes de Windows disponibles y sus valores numéricos están explicados en la documentación de Platform SDK de Microsoft.

Las implementaciones del método `PreFilterMessage()` deben devolver un valor booleano que describe si el mensaje debe ser enviado al formulario tras pasar por el filtro o no. Si el filtro considera que el mensaje no debe ser enviado, entonces devuelve `True`. Si el filtro considera que el mensaje debe ser enviado, entonces devuelve el valor `False`. El filtro de mensajes del listado 21.4 devuelve `True` para los dos mensajes que controla y `False` para todos los demás mensajes. El método `Main()` en el listado 21.4 crea un nuevo objeto de la clase `BlockLeftMouseButtonMessageFilter` y lo usa en una llamada al método `AddMessageFilter()` del objeto `Application`. Tras instalar el filtro de mensajes, se crea y ejecuta el formulario principal.

Puede ver el filtro de mensajes en acción compilando el listado 21.4. Al ejecutar el código, aparecerá el formulario principal de la aplicación. Cuando aparezca el formulario, mueva el ratón para que el cursor esté dentro del formulario y haga clic con el botón izquierdo del ratón. Se escribirá en la consola un mensaje indicando que ha pulsado un botón del ratón.

DoEvents

El método `DoEvents()` procesa todos los mensajes que se encuentren en la cola de mensajes de la aplicación de Windows. El método no recibe argumentos ni devuelve ningún valor. Este método se invoca cuando se quiere estar seguro de que los mensajes en espera de Windows se envían al formulario mientras se realizan otras tareas.

Suponga, por ejemplo, que crea un formulario que realiza un cálculo largo. Si se mueve otra ventana delante del formulario mientras se realiza el cálculo, Windows envía a la aplicación un mensaje de Windows que indica que el formulario debe ser dibujado de nuevo. Sin embargo, como el código está realizando un enorme cálculo, el mensaje de nuevo dibujo estará en la cola de mensajes de la aplicación; después de todo, el código está ocupado realizando cálculos y no procesando mensajes. Se pueden hacer llamadas al método `DoEvents()` en ciertos puntos del proceso para asegurarnos de que los mensajes de espera de Windows se procesan mientras el código está ocupado realizando otro trabajo.

Exit

El método `Exit()` obliga a la aplicación a terminar. El método informa a la cola de mensajes de la aplicación que debe finalizar y cierra los formularios cuando se procesa el último mensaje de la cola de mensajes de Windows.

Por lo general, el código no necesita invocar al método `Exit()`. Los formularios de Windows incluyen por defecto un cuadro de cierre en la esquina superior derecha del formulario y al hacer clic en ese cuadro se envía un mensaje de cierre a la cola de mensajes de Windows. Sin embargo, puede ser útil llamar a `Exit()` si el formulario incluye un control, como un botón o un elemento de menú que debe finalizar la aplicación al ser seleccionado.

ExitThread

El método `ExitThread()` sale del bucle de mensajes y cierra todos los formularios en el subproceso en curso. El método no recibe argumentos ni devuelve ningún valor. Si la aplicación `WindowsForms` contiene un solo subproceso (como es habitual), entonces la acción de llamar a `ExitThread()` es igual que a llamar a `Exit()`. Sin embargo, si la aplicación usa varios subprocesos, entonces los dos métodos se comportan de forma diferente. El método `ExitThread()` cierra un subproceso pero permite que los otros subprocesos sigan ejecutándose. Sin embargo, el método `Exit()`, cierra todos los subprocesos a la vez. Como todos los procesos de Windows, las aplicaciones `WindowsForms` siguen ejecutándose hasta que finaliza el último subproceso.

OleRequired

El método `OleRequired()` inicializa OLE en el subproceso en curso de la aplicación. Si la aplicación va a trabajar con tecnología COM, como COM, DCOM,

ActiveX o OLE, se debe llamar a este método de la aplicación antes de usar COM.

El método no recibe argumentos, pero devuelve un valor desde una enumeración llamada `ApartmentState` que describe el tipo de apartamento que introdujo el subproceso. La enumeración `ApartmentState` está definida en un espacio de nombre .NET Framework llamado `System.Threading` y puede tener uno de los siguientes valores:

- `STA`, se devuelve cuando el CLR decide inicializar COM para el subproceso entrando en un apartamento de un único subproceso.
- `MTA`, se devuelve cuando el CLR decide inicializar COM para el subproceso entrando en un apartamento de multiproceso.

OnThreadException

El método `OnThreadException()` desencadena un evento `ThreadException`. El evento puede ser capturado por un controlador de eventos `OnThreadException()` instalado en el objeto `Application`.

El listado 21.5 muestra cómo puede usarse una excepción de subproceso en una aplicación `WindowsForms`.

Listado 21.5. Cómo trabajar con excepciones de subprocesos

```
using System;
using System.Threading;
using System.Windows.Forms;

public class BlockLeftMouseButtonMessageFilter : IMessageFilter
{
    const int WM_LBUTTONDOWN = 0x201;

    public bool PreFilterMessage(ref Message m)
    {
        if (m.Msg == WM_LBUTTONDOWN)
        {
            Exception LeftButtonDownException;

            LeftButtonDownException = new Exception("The left mouse
button was pressed.");
            Application.OnThreadException(LeftButtonDownException);
            return true;
        }
        return false;
    }
}

public class ApplicationEventHandlerClass
{
    public void OnThreadException(object sender,
```

```

ThreadExceptionEventArgs e)
{
    Exception LeftButtonDownException;

    LeftButtonDownException = e.Exception;
    Console.WriteLine(LeftButtonDownException.Message);
}
}

public class MainForm : Form
{
    public static void Main()
    {
        ApplicationEventHandlerClass AppEvents = new
ApplicationEventHandlerClass();
        MainForm MyForm = new MainForm();
        BlockLeftMouseButtonMessageFilter MsgFilter = new
BlockLeftMouseButtonMessageFilter();

        Application.AddMessageFilter(MsgFilter);
        Application.ThreadException += new
ThreadExceptionHandler(AppEvents.OnThreadException);
        Application.Run(MyForm);
    }

    public MainForm()
    {
        Text = "Application Exception Test";
    }
}

```

El listado 21.5 es parecido al listado 21.4 ya que incluye un controlador de mensajes que busca mensajes que informan cuando se pulsa el botón izquierdo del ratón en el formulario de la aplicación.

La diferencia es que en el listado 21.5 se inicia una excepción al recibir el mensaje `left mouse button down`. El controlador de mensajes crea un nuevo objeto `Exception` y lo inicia usando el método `OnThreadException()` del objeto `Application`. El código del listado 21.5 también incluye un controlador de eventos de la aplicación, que se implementa en una clase llamada `ApplicationEventHandlerClass`. Esta clase controla el evento `OnThreadException()` y el método principal de la aplicación instala el controlador de eventos usando la propiedad `ThreadException` del objeto `Application`.

El controlador de excepciones del subproceso instalado en la clase `ApplicationEventHandlerClass` extrae la excepción del objeto `ThreadExceptionEventArgs` del controlador y escribe el mensaje de la excepción en la consola. Cuando se ejecuta el código en el listado 21.5, aparece el formulario principal de la aplicación. Cuando aparezca el formulario, mueva el ratón hacia su interior y haga clic con el botón izquierdo del ratón. El controlador

de mensajes iniciará una excepción y el controlador de excepciones de la aplicación escribirá mensajes de excepción en la consola de la aplicación.

RemoveMessageFilter

El método `RemoveMessageFilter()` elimina un filtro de mensajes instalado por el método `AddMessageFilter()`. Elimina el filtro de mensajes del generador de mensajes de la aplicación. El método `RemoveMessageFilter()` recibe un argumento: una implementación de una interfaz llamada `IMessageFilter`. Este argumento debe hacer referencia a una clase que implementa `IMessageFilter` y que ya ha sido usada en una llamada a `AddMessageFilter()`. El listado 21.6 muestra cómo funciona este método.

Listado 21.6. Cómo eliminar un filtro de mensajes instalado

```
using System;
using System.Windows.Forms;

public class BlockLeftMouseButtonMessageFilter : IMessageFilter
{
    const int WM_LBUTTONDOWN = 0x201;

    public bool PreFilterMessage(ref Message m)
    {
        if(m.Msg == WM_LBUTTONDOWN)
        {
            Console.WriteLine("The left mouse button is down.");
            Application.RemoveMessageFilter(this);
            return true;
        }
        return false;
    }
}

public class MainForm : Form
{
    public static void Main()
    {
        MainForm MyForm = new MainForm();
        BlockLeftMouseButtonMessageFilter MsgFilter = new
BlockLeftMouseButtonMessageFilter();

        Application.AddMessageFilter(MsgFilter);
        Application.Run(MyForm);
    }

    public MainForm()
    {
        Text = "Message Filter Removal Test";
    }
}
```

El código del listado 21.6 instala un filtro de mensajes que busca el mensaje `left mouse button down`, igual que hacía el listado 21.4. La diferencia es que la implementación del filtro de mensajes en el listado 21.6 elimina el filtro de mensajes cuando se recibe el mensaje.

Observe que, cuando se ejecuta el código del listado 21.6, sólo se escribe un mensaje en la consola, independientemente del número de veces que pulse el botón izquierdo del ratón con el puntero sobre el formulario. Esto es debido a que el filtro de mensajes es eliminado de la aplicación cuando se recibe el primer mensaje y, como se elimina el filtro de mensajes, no se pueden detectar nuevos mensajes. Todavía se envían mensajes al formulario, pero el código del listado 21.6 elimina el objeto que detecta por primera vez los mensajes después de que se haya eliminado el objeto de la lista de filtros de eventos de la aplicación.

TRUCO: El listado 21.6 usa la palabra clave `this` como parámetro para la llamada al método `RemoveMessageFilter()` del objeto `Application`. Recuerde que la palabra clave `this` se emplea para hacer referencia al objeto cuyo código se está ejecutando. Puede pensar en la instrucción del listado 21.6 que llama a `RemoveMessageFilter()` como si indicara "elimina la referencia a este filtro de mensajes del objeto `Application`".

Run

El método `Run()` inicia el bucle de mensajes de Windows para una aplicación. Todos los listados de este capítulo han usado el método `Run()`, que acepta como parámetro una referencia a un objeto de formulario. Ya debería estar familiarizado con el funcionamiento del método `Run()`.

Cómo añadir controles al formulario

El formulario que las aplicaciones `WindowsForms` crean por defecto no es muy interesante. Tiene una barra de título, un icono por defecto y los botones estándar de Windows Minimizar, Maximizar y Cerrar. Los formularios que encontramos en las aplicaciones reales incluyen controles como botones, cuadros de texto, etiquetas y elementos similares. Esta sección explica cómo añadir controles a los formularios de las aplicaciones `C#`.

En esta sección, examinaremos cómo se implementan los controles desde `.NET Framework`. El entorno de desarrollo dispone de compatibilidad de clases `.NET` para los controles integrados en el sistema operativo Windows, y los ejemplos de esta sección muestran su uso en la creación de aplicaciones `WindowsForms` que usan controles en los formularios de la aplicación.

Jerarquía de las clases de control

.NET Framework incluye varias clases en el espacio de nombres `System.Windows.Forms` para encapsular el comportamiento de un control. Los elementos de interfaz de usuario, como botones, cuadros de texto, casillas de verificación y elementos similares, están representados por una clase de control.

Todas estas clases se derivan de una clase base llamada `Control`. La figura 21.2 muestra la jerarquía de clases de las clases de control. Todos los controles de la interfaz de usuario comparten alguna funcionalidad: todos deben ser capaces de situarse en su contenedor y gestionar sus colores de primer plano y de fondo. Como todos los controles comparten este comportamiento, es lógico encapsularlo en una clase base y derivar la funcionalidad específica al control en las clases derivadas. Los autores de las clases de control de .NET Framework adoptaron este enfoque al construir las clases.

Cómo trabajar con controles en un formulario

El listado 21.7 muestra una aplicación `WindowsForm` que incluye un botón. El botón incluye un mensaje en un cuadro de texto al hacer clic.

Listado 21.7. Cómo trabajar con un botón en un formulario

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class MainForm : Form
{
    public static void Main()
    {
        MainForm MyForm = new MainForm();

        Application.Run(MyForm);
    }

    public MainForm()
    {
        Button MyButton = new Button();

        Text = "Button Test";
        MyButton.Location = new Point(25, 25);
        MyButton.Text = "Click Me";
        MyButton.Click += new EventHandler(MyButtonClicked);
        Controls.Add(MyButton);
    }

    public void MyButtonClicked(object sender, EventArgs
Arguments)
```

```

{
    MessageBox.Show("The button has been clicked.");
}
}

```

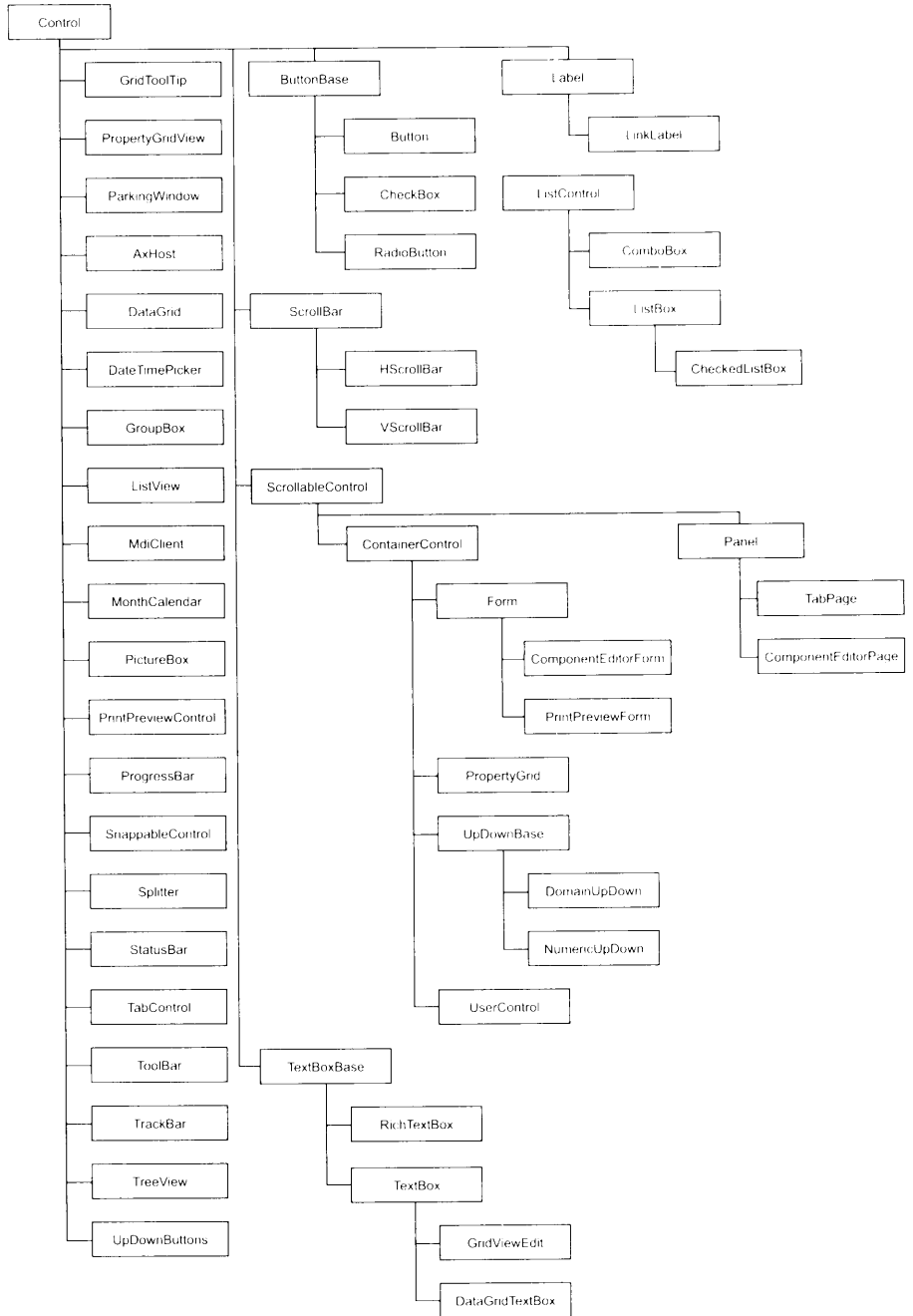


Figura 21.2. Jerarquía de las clases de control

El listado 21.7 muestra varios conceptos importantes que deben tenerse en cuenta cuando se trabaja con controles `WindowsForms`. Estudie el constructor del primer formulario. Crea un nuevo objeto de clase `Button` y establece su posición en el formulario con la propiedad `Location` del botón. Esta propiedad se hereda de la clase `Control` (lo que significa que la propiedad está disponible para cualquier control derivado de la clase `Control`) y establece la posición de la esquina superior izquierda del botón respecto a su contenedor. En el listado 21.7, la posición del botón se establece a 25 píxeles a la derecha del borde izquierdo del formulario y a 25 píxeles por debajo de la parte superior del formulario. La posición se establece con una nueva instancia de una estructura llamada `Point`, que está disponible en el espacio de nombres `System.Drawing` de `.NET Framework`:

```
MyButton.Location = new Point(25, 25);
```

TRUCO: El listado 21.7 usa la propiedad `Location` para establecer la ubicación del control. El uso de esta propiedad para que el programa coloque los controles puede suponer demasiado trabajo en formularios complicados con muchos controles. Visual Studio `.NET` dispone de un diseñador de formularios que permite, de forma visual, arrastrar y colocar controles en formularios. El diseñador crea a continuación los formularios `C#` equivalentes, liberando al programador de la tarea de tener que codificar toda la lógica de posicionamiento por sí mismo.

El siguiente concepto importante del listado 21.7 está relacionado con el control de eventos de un control. Las clases de control admiten muchos eventos que se desencadenan cuando el usuario interactúa con el control. Muchos de estos eventos se incluyen en la clase base `Control`, aunque la clase de control específica controla otros eventos. El evento más evidente de un botón de control sería un evento `Click`. Los botones de los formularios no son de utilidad a menos que puedan responder a una acción de un usuario que haga clic con el botón.

Los controles de `.NET Framework` usan el modelo estándar delegado/evento para compatibilizar sus eventos. Los eventos de control se instalan usando instancias de un delegado llamado `EventHandler`. Este delegado admite dos argumentos: un objeto que especifica el elemento que envió el evento y un objeto de una clase llamada `EventArgs` que encapsula los argumentos del evento. El código del formulario del listado 21.7 incluye un método llamado `MyButtonClicked` que modela el delegado `EventHandler`. Este método se usa como un nuevo controlador de eventos y está conectado al evento `Click` del botón:

```
MyButton.Click += new EventHandler(MyButtonClicked);
```

La clase `Form` controla el evento `Click` del botón mostrando un cuadro de texto. Una clase llamada `MessageBox` admite la visualización de cuadros de

mensajes de Windows. La clase `MessageBox` contiene un método estático llamado `Show()` que muestra un mensaje en un cuadro de mensaje.

El último concepto importante del listado 21.7 es la instrucción que agrega el control a la forma:

```
Controls.Add(MyButton);
```

La propiedad `Control` está definida en la clase base `Control` (recuerde que la clase `Form` se deriva de la clase `Control`). Es un objeto de una clase llamada `ControlsCollection` y gestiona una lista de controles secundarios que son gestionados por el control actual. El código `WindowsForms` debe añadir controles a su colección `Controls` de los formularios contenedores antes de que pueda ser usada realmente.

Cómo trabajar con recursos

En Windows, los recursos se definen como datos que forman parte de una aplicación pero no afectan a la ejecución del código. Estos recursos pueden ser iconos, mapas de bits o cadenas. El sistema `WindowsForms` permite almacenar los recursos en un archivo separado durante el desarrollo del programa e incluirlos en un ensamblado cuando se distribuya la aplicación.

La principal ventaja de guardar los recursos de la aplicación en un archivo separado es que ayuda al desarrollo del programa. Si incrustamos todas las cadenas dentro del código C#, por ejemplo, entonces sólo alguien que conozca C# sabrá dónde buscar para cambiar los valores de cadena. Si la aplicación se escribe usando cadenas en inglés y luego es necesario cambiar la aplicación para que muestre cadenas en alemán, habrá que leer todo el código fuente y cambiar todas las cadenas. Si guardamos las cadenas en un archivo de tabla de cadenas aparte, podemos aplicar un traductor a ese archivo separado para que traduzca las cadenas en inglés a su equivalente en alemán sin cambiar el código C# fuente. En el código de la aplicación, el código dirá "lee una cadena de la tabla de cadenas" en lugar de teclear la cadena en el código de la aplicación.

Cómo trabajar con recursos de cadena

Los recursos de cadena se definen en un archivo de texto separado, que debe tener la extensión `.txt`. El archivo debe contener un conjunto de pares clave/valor, separados por el signo igual. La clave para cada cadena debe ser un nombre único para la cadena que se va a usar en el código C# para hacer referencia a ella. El valor de cadena real se escribe tras el signo igual. Se pueden colocar comentarios en los archivos de tabla de cadenas. Los comentarios comienzan con el símbolo de la libra y llegan hasta el final de la línea. El listado 21.8 muestra un ejemplo de archivo de tabla de cadenas. El archivo contiene una cadena cuyo

nombre de clave es `Message` y cuyo valor es `Hello from the string table!`

Listado 21.8. Ejemplo de un archivo de texto de una tabla de cadena

```
#####  
# String Table  
#####  
Message = Hello from the string table!
```

Los archivos de tabla de cadena deben compilarse para formar un ensamblado de modo que las aplicaciones `C#` puedan leerlos. Esto se hace con una herramienta llamada `ResGen`. La herramienta `ResGen` se incluye en el SDK de `.NET Framework`. Es una aplicación de consola que lee el archivo de texto y produce una representación binaria de la tabla con extensión `resources`. Si la tabla de cadena del listado 21.8 se escribe en un archivo de texto llamado `Listing21-8.txt`, puede compilar la tabla de cadenas usando la siguiente línea de comandos:

```
resgen Listing21-8.txt
```

Esto produce un archivo llamado `Listing21-8.resources`. Tras compilar un archivo `resources` para la aplicación, se puede compilar en el ensamblado usando el argumento `/res` para el compilador de `C#`, como muestra la siguiente línea de comando:

```
csc /res:string.resources /out:test.exe test.cs
```

Esta línea de comando ordena al compilador de `C#` que cree un ejecutable llamado `test.exe` a partir del archivo fuente de `C#` `test.cs`. También le ordena incrustar en el ejecutable `test.exe` los recursos que encuentre en el archivo `string.resources`. Como los recursos están incrustados en el ejecutable, sólo se necesita enviar el ejecutable cuando se distribuye la aplicación. El archivo de recursos binarios no es necesario en tiempo de ejecución.

Una vez que se han incrustado los recursos en la aplicación, se pueden leer desde el código de `C#`. El listado 21.9 es una modificación del listado 21.7, en el que el mensaje del cuadro de mensajes se lee desde un recurso de cadena.

Listado 21.9. Cómo leer desde un recurso de cadena

```
using System;  
using System.Drawing;  
using System.Windows.Forms;  
using System.Resources;  
using System.Reflection;  
  
public class MainForm : Form  
{  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

```

{
    MainForm MyForm = new MainForm();

    Application.Run(MyForm);
}

public MainForm()
{
    Button MyButton = new Button();

    Text = "Button Test";
    MyButton.Location = new Point(25, 25);
    MyButton.Text = "Click Me";
    MyButton.Click += new EventHandler(MyButtonClicked);
    Controls.Add(MyButton);
}

public void MyButtonClicked(object sender, EventArgs
Arguments)
{
    ResourceManager FormResources = new
ResourceManager("StringTable",
Assembly.GetExecutingAssembly());
    string Message;

    Message = FormResources.GetString("Message");
    MessageBox.Show(Message);
}
}

```

El listado 21.9 se compila con un recurso de tabla de cadenas cuyo archivo de texto contiene lo siguiente:

```

#-----
# String Table
#-----
Message = The button has been clicked.

```

Este archivo de texto recibe el nombre de `StringTable.txt` y se compila formando un archivo de recursos binario llamado `StringTable.resources` mediante la siguiente línea de comando:

```
resgen StringTable.txt
```

Este comando produce un archivo llamado `StringTable.resources`. Este recurso se vincula a la aplicación cuando se compila el código C# principal mediante la siguiente línea de comando:

```
csc /res:StringTable.resources Listing21-9.cs
```

Se pueden leer recursos en las aplicaciones de C# usando una clase de .NET Framework llamada `ResourceManager`, que se incluye en un espacio de nom-

bres llamado `System.Resources`. El código del listado 21.9 crea un nuevo objeto `ResourceManager` para gestionar los recursos incrustados en el ejecutable. El constructor recibe dos argumentos:

- El nombre base del archivo de recursos binarios que contiene el recurso que se está abriendo. Se debe especificar este nombre, aunque no se necesita el archivo físico porque el ensamblado agrupa los recursos en bloques y da nombre a los bloques usando el nombre base del archivo de recursos binarios original.
- Una referencia al ensamblado que contiene los recursos que se están abriendo. Este parámetro es una referencia a un objeto de una clase llamada `Assembly`, que se encuentra en el espacio de nombres `System.Reflection`. Como los recursos que se están abriendo están incrustados en el ensamblado que se está ejecutando, si se llama al método estático `GetExecutingAssembly()` se devolverá una referencia al ensamblado actual.

Tras inicializar el objeto `ResourceManager`, las cadenas se pueden abrir desde el administrador mediante un método llamado `GetString()`. Este método recibe un argumento de cadena: el nombre de clave de la cadena que se recupera. El método devuelve el valor de la cadena nombrada por la clave.

Cómo trabajar con recursos binarios

Las tablas de cadenas basadas en texto no son los únicos recursos que se pueden incrustar en los ensamblados. También se pueden incrustar recursos binarios, como gráficos e iconos. Los recursos binarios están codificados, mediante codificación BASE64, en un documento XML con un formato especial. Este documento XML tiene la extensión `.resx` y es compilado para formar un archivo de recurso mediante `resgen`. A partir de aquí, puede usar los métodos de la clase `ResourceManager` para trabajar con los recursos binarios como si fueran recursos de texto.

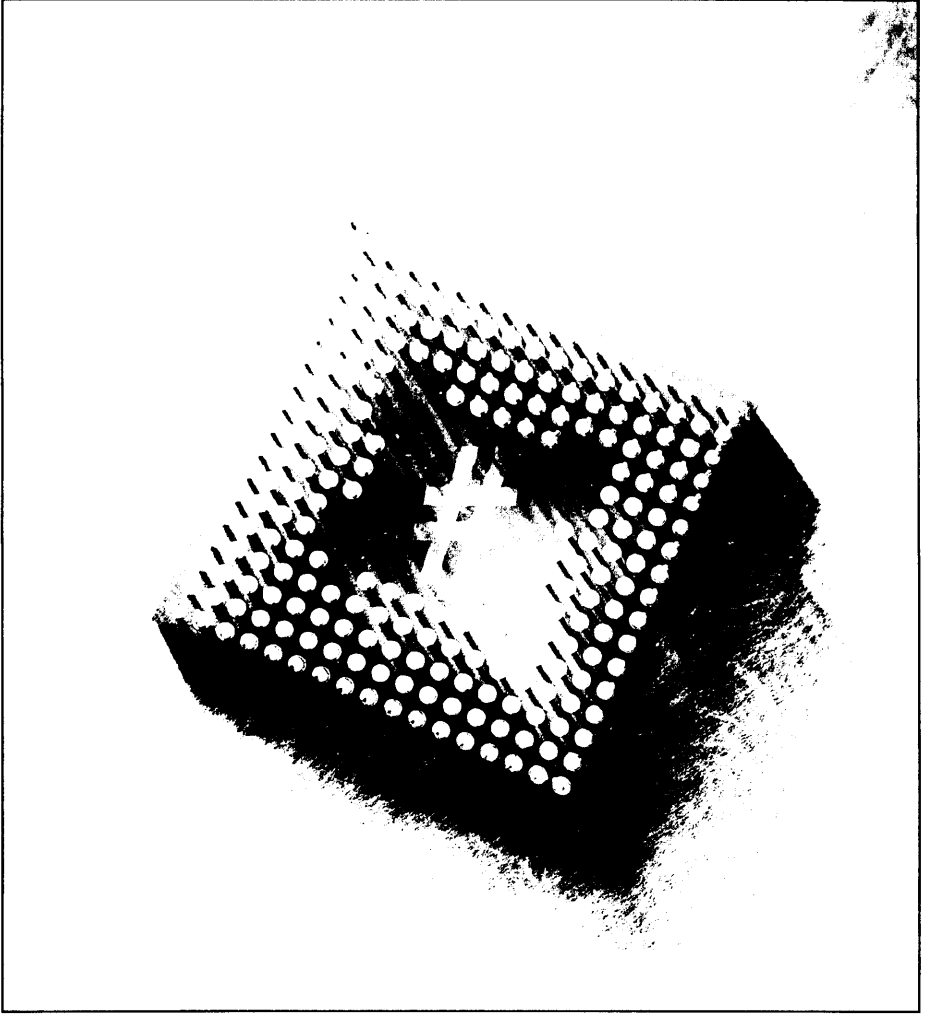
Desafortunadamente, el SDK de .NET Framework no incluye una herramienta para generar documentos XML con codificación BASE64 a partir de entradas de archivo binario. Sin embargo, Visual Studio .NET permite incrustar recursos binarios en los ensamblados.

Resumen

Este capítulo estudia los fundamentos del proceso de desarrollo para la elaboración de aplicaciones WindowsForms en C#. También se estudian algunas clases elementales, como la clase `Application`, que gestiona la aplicación

WindowsForms como un todo y la clase `Form`, que gestiona un formulario de la aplicación. También se hace un repaso a la arquitectura de las clases de control WindowsForms y se examinan los atributos de ensamblado que pueden agregar información de versión y descriptiva al ensamblado.

.NET Framework contiene un variado conjunto de clases para elaborar aplicaciones WindowsForms. El subsistema WindowsForms está compuesto por varias clases; por desgracia, las limitaciones de espacio no permiten una completa descripción de todas ellas en este libro. Puede examinar la documentación de cada clase WindowsForms. Use los conceptos de este capítulo para comenzar a investigar todas las clases del espacio de nombres WindowsForms.



22 **Cómo crear aplicaciones Web con WebForms**

La última década ha sido testigo del crecimiento sin precedentes de Internet como plataforma de negocios. Hoy en día, la mayoría de los modelos de negocios están basados en, o al menos incluyen, el concepto de Internet. Por tanto, el enfoque ha cambiado de las aplicaciones de escritorio a las aplicaciones Web. Este cambio ha subrayado la necesidad de tecnologías que puedan simplificar el desarrollo de aplicaciones Web.

Para crear aplicaciones Web, .NET Framework incluye ASP.NET, que es la nueva versión de ASP 3.0. Se pueden crear aplicaciones en ASP.NET usando Visual Basic .NET o Visual C# como lenguaje de programación del servidor. Visual C# permite a los programadores desarrollar potentes aplicaciones Web. Pero lo más importante es que ayuda a los programadores a luchar contra los ciclos cada vez más rápidos, porque les permite realizar más operaciones con menos líneas de código y con menos errores, lo que reduce considerablemente el coste del proyecto.

Aunque para crear aplicaciones Web ASP.NET sólo se necesita un editor de texto, como el bloc de notas, lo más habitual es usar una plataforma de desarrollo, como Visual Studio .NET, que proporciona un enorme conjunto de herramientas para diseñar páginas Web. En comparación con los primeros lenguajes de programación de páginas Web, en los que había que realizar una gran cantidad de codificación, Visual Studio .NET proporciona de una interfaz WYSIWYG. Esta

interfaz permite arrastrar y colocar controles en WebForms, que luego pueden ser programadas en Visual C#. Al programar en Visual Studio .NET se puede separar el contenido en código y en HTML de un WebForm. Esto hace que resulte muy sencillo separar la programación lógica de la presentación lógica, lo que nos permite concentrarnos en implementar la funcionalidad del proyecto, más que en la presentación de datos.

En este capítulo aprenderemos a crear una aplicación Web ASP.NET mediante Visual C#. Mientras creamos la aplicación, diseñaremos un WebForm que usa controles de servidor, como etiquetas, cuadros de texto, cuadros de listas, hipervínculos y botones. Por último, aprenderemos a controlar los eventos generados por los controladores de servidor.

Fundamentos de las aplicaciones ASP.NET Web

Las aplicaciones Web ASP.NET son aplicaciones que se emplean en servidores Web. Estas aplicaciones forman uno o más WebForms programados en Visual C# o Visual Basic .NET.

En esta sección, estudiaremos las ventajas y desventajas de las aplicaciones Web ASP.NET y cómo crear aplicaciones ASP.NET con Visual C#. También estudiaremos las diferencias entre las aplicaciones ASP.NET y ASP 3.0.

Nuevas características de ASP.NET

ASP.NET incluye algunas nuevas características que no estaban presentes en ASP 3.0. En esta sección se describen brevemente estas características.

Ejecución en el entorno .NET Framework

En comparación con los primeros lenguajes de programación Web, las aplicaciones en Visual C# (y otros lenguajes Visual Studio .NET) se ejecutan en el entorno del marco de trabajo .NET. Así, estas aplicaciones son independientes del navegador cliente y funcionan de la misma manera en todas las plataformas cliente.

Otra ventaja de usar un tiempo de ejecución diferente para ASP.NET es que las aplicaciones ASP 3.0 pueden coexistir con las aplicaciones ASP.NET. Así, se pueden usar sitios Web ASP 3.0 y ASP.NET en el mismo servidor Web.

Presentación de WebForms

Los *WebForms* son la base de una aplicación basada en Web. La aplicación Web los usa para interactuar con el usuario. Un WebForm puede incluir varios controles de servidor, como cuadros de texto, etiquetas, cuadros de listas, botones

de opción, casillas de verificación y botones, los cuales facilitan la interacción del usuario con la aplicación.

Un WebForm consta de dos componentes: la *interfaz de usuario* (IU) y la lógica de la aplicación (aplicación). La interfaz de usuario es el componente visual de un WebForm. Se compone de HTML y controles específicos de la aplicación Web. La interfaz de usuario es el contenedor del texto y los controles que deben aparecer en la página Web. Se especifica en un archivo con la extensión `.aspx`.

La lógica de programación de una aplicación Web de ASP.NET está en un archivo separado que contiene el código encargado de controlar las interacciones del usuario con el formulario. Este archivo recibe el nombre de archivo de "*código oculto*". Cuando se ejecuta un formulario escrito en C#, el archivo de código oculto genera dinámicamente el resultado HTML de la página. El archivo de código oculto de C# tiene una extensión `.aspx.cs`.

La ventaja de separar el código del contenido es que el programador no necesita concentrarse en la lógica que se usa para mostrar el resultado. El diseñador Web puede controlar esta tarea.

Integración con Visual Studio .NET

Visual Studio .NET es la herramienta de desarrollo rápido de aplicaciones para ASP.NET. Visual Studio .NET ofrece una completa integración con ASP.NET y permite arrastrar y colocar controladores de servidor y diseñar WebForms con el aspecto que tendrán cuando un usuario los vea. Algunas de las otras ventajas de crear aplicaciones ASP.NET con Visual Studio .NET se resumen en la siguiente lista:

- Visual Studio .NET es una herramienta de desarrollo rápida de aplicaciones (RAD). En lugar de añadir cada control al WebForm mediante programación, le ayuda a añadir estos controles usando el cuadro de herramientas, ahorrándole trabajo de programación.
- Visual Studio .NET admite controles personalizados y compuestos. Se pueden crear controles personalizados que encapsulen una funcionalidad común cuyo uso puede ser necesario en varias aplicaciones, del mismo modo que se usan los controles Web ASP.NET proporcionados por Visual Studio .NET.

Presentación de los controles de servidor

Además de los controles HTML que existían en la época de ASP 3.0, ASP.NET presenta controles de servidor que son componentes de una aplicación Web que se ejecutan en el servidor y encapsulan la funcionalidad de la aplicación.

Los controles HTML hacen referencia a los elementos HTML que se pueden usar en los WebForms. Por lo general, cuando los controles HTML se envían al servidor a través del navegador, el servidor considera que los controles HTML

son opacos. Es decir, el servidor no los procesa. Sin embargo, al convertir estos controles en controles de servidor HTML pueden quedar a la vista del servidor para que realice el proceso. Mediante el uso de atributos, como `ID` y `RUNAT`, se pueden convertir los controles HTML en controles de servidor HTML. Puede añadir estos controles a un WebForm usando la ficha HTML del cuadro de herramientas. Por otra parte, los controles de servidor son completamente transparentes para la aplicación y permiten al programador controlar eventos del lado del servidor para gestionar la aplicación Web.

Aparte de los cuadros de texto convencionales, esta categoría de controles también incluye los controles de validación. Los controles de validación son controles programables que ayudan a validar las entradas del usuario. Por ejemplo, se pueden usar estos controles para validar el valor de un campo o el patrón de caracteres introducido por el usuario. Para validar la entrada del usuario, es necesario adjuntar estos controles a los controles de la entrada.

Controles de usuario y compuestos

Si quiere duplicar un conjunto de controles en varias páginas, puede crear controles en cada formulario por separado. Ésta no es una opción muy útil. Sin embargo, ASP.NET nos permite realizar esta operación mediante los controles de usuario y compuestos.

Los controles de usuario son WebForms normales que hemos convertido en controles eliminando las etiquetas `<HTML>` y `<FORM>` del control. Así, representan una unidad de código y presentación que puede importarse a otro WebForm.

Otro conjunto de controles disponibles en ASP.NET es el de los controles compuestos. Los controles compuestos son un conjunto de controles que se han compilado para formar una biblioteca. Para usar controles compuestos hay que incluir una referencia a la biblioteca del mismo modo que se incluyen las referencias a otras bibliotecas.

Controles más usados en WebForms

La tarea básica de diseñar una aplicación Web es añadir controles a un WebForm. Algunos de los controles más usados en un WebForm son `Label`, `TextBox`, `CheckBox`, `RadioButton`, `ListBox`, `DropDownList`, `HyperLink`, `Table`, `Button` e `ImageButton`. Las siguientes secciones explican brevemente estos controles.

Control Label

El control `Label` se usa para mostrar texto estático en un WebForm. Los usuarios no pueden editar el texto de un control `Label`. Al añadir un control `Label`, el texto `Label` aparece como su título. Sin embargo, asignando un valor a la propiedad `Text` del control, es posible modificar el título del control. Se puede asignar un valor a las propiedades del control `Label` en tiempo de ejecución en

el archivo de código oculto (.cs file). Por ejemplo, si se quiere cambiar el texto de una etiqueta cuando un usuario pulsa un botón. Para ello, puede utilizar el siguiente código:

```
Label1.Text="Welcome"
```

En el anterior código, Label1 es el ID del control Label cuya identificación quiere cambiar. Si quiere que el control Label desaparezca cuando un usuario pulse un botón, puede usar el siguiente código:

```
Label1.Visible=False
```

Control TextBox

El control TextBox se usa para obtener información, como texto, números y fechas, de los usuarios de un WebForm. Por defecto, un control TextBox es un control de una línea que permite a los usuarios escribir caracteres en una sola línea. Sin embargo, también se puede establecer el control TextBox como un control multilinea. Un cuadro de texto multilinea muestra varias líneas y permite el ajuste de texto. Un control TextBox también puede usarse para aceptar contraseñas. Los controles TextBox utilizados para aceptar contraseñas ocultan los caracteres escritos por los usuarios, mostrándolos como asteriscos (*).

Puede establecer la apariencia de un control TextBox mediante sus propiedades, como BackColor o ForeColor. También puede cambiar la propiedad TextMode de un control TextBox para determinar si un control TextBox actúa como un cuadro de texto para aceptar una contraseña, una sola línea de texto o varias líneas de texto.

Controles CheckBox y CheckBoxList

Las casillas de verificación permiten a los usuarios seleccionar una o más opciones de un conjunto de opciones dado. Se pueden añadir casillas de verificación a un WebForm mediante los controles CheckBox o CheckBoxList. El control CheckBox representa una sola casilla de verificación, mientras que el control CheckBoxList representa una colección de varias casillas de verificación. Para agregar estos controles al formulario, simplemente tiene que arrastrarlos hasta él desde el cuadro de herramientas.

Tras agregar el control CheckBoxList, es necesario añadirle una lista de elementos. Para hacerlo, siga estos pasos:

1. En la ventana Propiedades, haga clic en el botón de puntos suspensivos para acceder a la propiedad Items del control CheckBoxList. Se abrirá el cuadro de diálogo Editor de la colección ListItem.

NOTA: Si la ventana de Propiedades no está abierta, pulse F4. También puede seleccionar Ver>Ventana Propiedades, en la barra de menú.

2. En el cuadro de diálogo **Editor de la colección ListItem**, haga clic en **Agregar** para crear un nuevo elemento. Se creará un nuevo elemento y sus propiedades se mostrarán a la derecha del cuadro de diálogo.
3. Verifique que el elemento está seleccionado en la lista **Miembros** y a continuación establezca las propiedades del elemento. Cada elemento es un objeto distinto y tiene las siguientes propiedades:
 - **Selected**: representa un valor booleano que indica si el elemento está seleccionado.
 - **Text**: representa el texto que se muestra para el elemento de la lista.
 - **Value**: representa el valor asociado al elemento. El valor de un control no se muestra al usuario. Sin embargo, el servidor usa el valor para procesar la información del control. Por ejemplo, puede establecer la propiedad **Text** de un elemento como **Nombre de Ciudad** y la propiedad **Value** del código postal de esa ciudad como identificación única. Cuando el servidor procesa la información representada por el campo **Nombre de Ciudad**, se puede hacer caso omiso del texto proporcionado por el cuadro de texto y cualquier proceso se basará en el correspondiente valor del campo.
4. Especifique el texto que se mostrará al usuario.
5. Repita los pasos 2-4 para agregar los controles necesarios al control `CheckBoxList`.
6. Haga clic en **Aceptar** para cerrar el cuadro de dialogo **Editor de la colección ListItem**.

TRUCO: La decisión de usar el control `CheckBox` o el control `CheckBoxList` depende de las necesidades específicas. El control `CheckBox` proporciona más control sobre la presentación de las casillas de verificación de la página. Por ejemplo, se puede establecer la fuente y el color de las casillas de verificación por separado o incluir texto entre las diferentes casillas de verificación. Por otra parte, el control `CheckBoxList` es una mejor opción si se necesitan agregar series de casillas de verificación.

Controles `RadioButton` y `RadioButtonList`

Los botones de opción proporcionan un conjunto de opciones para el usuario. Puede agregar botones de opción a un `WebForm` usando el control `RadioButton` o el control `RadioButtonList`. El control `RadioButton` representa a un solo botón de opción con el que trabajar. El control `RadioButtonList` es una

colección de botones de opción. Los botones de opción casi nunca se usan individualmente. Más bien se usan dentro de un grupo.

Un grupo de botones de opción proporciona un conjunto de opciones mutuamente excluyentes. Esto significa que sólo se puede seleccionar un botón de opción en un grupo. Un conjunto de botones de opción puede agruparse de estas dos maneras:

- Puede agregar un conjunto de controles `RadioButton` a la página y asignarlos a un grupo manualmente. Puede usar la propiedad `GroupName` para hacerlo.
- Puede agregar un control `RadioButtonList` a la página. Los botones de opción en el control se agrupan automáticamente, de modo que no es necesario agruparlos manualmente.

Tras añadir un control `RadioButtonList` al `WebForm`, hay que agregar los botones de opción. Esto se puede hacer usando la propiedad `Items` del mismo modo que hicimos con el control `CheckBoxList`.

Control `ListBox`

El control `ListBox` representa una colección de elementos de lista. El control permite a los usuarios seleccionar uno o más elementos de la lista. Se pueden añadir elementos a la lista individual mediante la propiedad `Items`. También puede especificar si el usuario puede seleccionar varios elementos de la lista o si sólo puede seleccionar un único elemento, mediante la propiedad `SelectionMode` del control `ListBox`.

Control `DropDownList`

El control `DropDownList` permite a los usuarios seleccionar un elemento de un conjunto de elementos predefinidos (siendo cada elemento un objeto diferente con sus propias propiedades).

Se pueden agregar elementos a un control `DropDownList` mediante su propiedad `Items`. A diferencia del control `ListBox`, sólo se puede seleccionar un elemento cada vez y la lista de elementos permanece oculta hasta que el usuario hace clic en el botón desplegable.

Control `HyperLink`

El control `HyperLink` permite a los usuarios moverse de un `WebForm` a otro dentro de una aplicación. También permite a los usuarios desplazarse hasta una URL que puede estar asociada con el control.

Con el control `HyperLink`, el texto o una imagen pueden funcionar como un hipervínculo. Cuando un usuario hace clic en el control, se abre el `WebForm` de destino o la URL.

El siguiente fragmento de código muestra cómo establecer la propiedad `NavigateUrl`:

```
Hyperlink1.NavigateUrl="http://www.amazon.com";
```

Controles `Table`, `TableRow` y `TableCell`

Las tablas se usan para mostrar información en formato tabular. Se puede agregar una tabla a un `WebForm` mediante el control `Table`. Este control puede mostrar estáticamente información estableciendo las filas y columnas durante su creación. Sin embargo, se puede programar el control `Table` para mostrar información dinámicamente en tiempo de ejecución.

Otros dos controles relacionados con las tablas que se pueden emplear en un `WebForm` son `TableRow` y `TableCell`. El control `TableRow` se usa para declarar una fila y el control `TableCell` para declarar una celda en una tabla.

Para comprender cómo se relacionan entre sí los controles `Table`, `TableRow` y `TableCell`, agregue un control `Table` a su `WebForm` (arrástrelo desde el explorador de soluciones) y realice los siguientes pasos para agregar filas y celdas a la tabla:

1. En la ventana de **Propiedades**, haga clic en el botón de puntos suspensivos para que aparezca la propiedad `Rows` del control `Table`. Se abrirá el cuadro de diálogo **Editor de colección TableRow**.
2. En el cuadro de diálogo **Editor de colección TableRow**, que representa el control `TableRow`, haga clic en **Agregar** para crear una nueva fila. Se creará una nueva fila y se mostrarán sus propiedades a la derecha del cuadro de diálogo.
3. Verifique que la fila ha sido seleccionada en la lista de miembros y a continuación haga clic en el botón de puntos suspensivos para que la propiedad `Cells` añada una celda a la fila. El cuadro de diálogo **Editor de colección TableCell** se abrirá.
4. En el cuadro de diálogo **Editor de colección TableCell**, que representa al control `TableCell`, haga clic en **Agregar** para crear una nueva celda. Se creará una nueva celda y se mostrarán sus propiedades en el lado derecho del cuadro de diálogo.
5. Especifique el texto que debe mostrarse en la celda y haga clic en **Aceptar** para cerrar el cuadro de diálogo **Editor de colección TableCell**.
6. Haga clic en **Aceptar** para cerrar el cuadro de diálogo **Editor de colección TableRow**.

Observe que después de realizar estos pasos, se ha añadido una tabla 1 x 1 al formulario. La tabla 22.1 describe algunas de las propiedades de los controles `Table`, `TableRow` y `TableCell`.

Tabla 22.1. Propiedades de los controles `Table`, `TableRow` y `TableCell`

Propiedad	Disponible con	Descripción
<code>ID</code>	<code>Table</code>	Representa el ID único del control.
<code>Rows</code>	<code>Table</code>	Representa una colección de objetos <code>TableRow</code> . Un control <code>TableRow</code> representa una fila de la tabla.
<code>Cells</code>	<code>TableRow</code>	Representa una colección de objetos <code>TableCell</code> . Un control <code>TableCell</code> representa una celda de la tabla.
<code>VerticalAlign</code>	<code>TableCell</code>	Representa el alineamiento vertical, como las partes superior e inferior de la celda.
<code>HorizontalAlign</code>	<code>TableCell</code>	Representa el alineamiento horizontal, como los alineamientos derecho e izquierdo, de la celda.

Control `ImageButton`

El control `ImageButton` permite a los programadores mostrar imágenes en un `WebForm` y gestionarlas durante el diseño o en tiempo de ejecución. Este control representa un botón gráfico, que mejora la apariencia del `WebForm`. Se puede establecer la propiedad `ImageUrl` para que apunte a una imagen específica.

Controles `Button` y `LinkButton`

El control `Button` de un `WebForm` se usa para enviar la página al servidor. Se pueden agregar tres tipos de botones de control de servidor a un `WebForm`:

- `Button`: Representa un botón estándar.
- `LinkButton`: Representa un botón que hace que la página se envíe al servidor. Además, también puede funcionar como hipervínculo a otra página Web o `WebForm`.
- `ImageButton`: Este control se estudió en el anterior apartado.

Cómo crear y configurar una aplicación Web

Visual C# proporciona la plantilla de aplicación Web ASP.NET para crear aplicaciones Web ASP.NET. Esta plantilla contiene la información necesaria para

crear, procesar y emplear aplicaciones ASP. Antes de crear un proyecto de aplicación Web necesitará asegurarse de que se cumplen en la plataforma de desarrollo los siguientes requisitos básicos para la aplicación Web:

- Deberá tener acceso a un equipo que ejecute Microsoft IIS Server.
- Debe instalar IIS Server en una partición NTFS. Este tipo de partición mejora la seguridad y rendimiento del servidor.

Tras cumplir estos requisitos, puede usar Visual Studio .NET para crear una aplicación Web ASP.NET.

Para ello, siga los siguientes pasos:

1. Agregue un proyecto de aplicación Web ASP.NET a su aplicación.
2. Cree la interfaz de usuario de la aplicación Web.
3. Codifique la lógica de la aplicación.

En la siguiente sección se indican los pasos para crear un nuevo proyecto.

Cómo crear un nuevo proyecto

Use la plantilla de aplicación Web ASP.NET para crear proyectos de aplicaciones Web. Los pasos para crear una nueva aplicación Web usando esta plantilla son los siguientes:

1. Seleccione Archivo > Nuevo > Proyecto para abrir el cuadro de diálogo Nuevo Proyecto, como aparece en la figura 22.1.

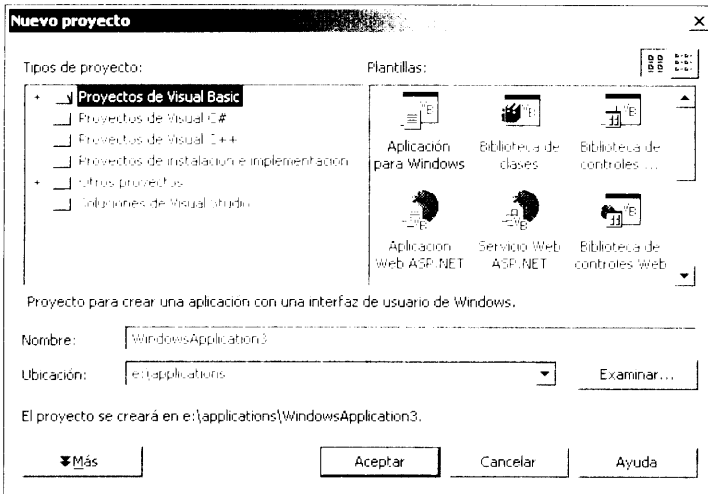


Figura 22.1. Puede seleccionar una o más plantillas de empresa del cuadro de diálogo Nuevo proyecto.

TRUCO: También puede pulsar la combinación **Control-Mayús-N** para abrir el cuadro de diálogo **Nuevo proyecto**.

2. Seleccione **Proyectos de Visual C#** de la lista **Tipos de proyecto**.
3. Seleccione **Aplicación Web ASP.NET** de la lista **Plantillas** en el cuadro de diálogo **Nuevo proyecto**.
4. Escriba el nombre del proyecto en el cuadro **Nombre**.
5. Escriba el nombre de **IIS Server** en el cuadro **Ubicación** o acepte la ubicación por defecto. El cuadro de diálogo **Nuevo proyecto** puede verse completo en la figura 22.2.

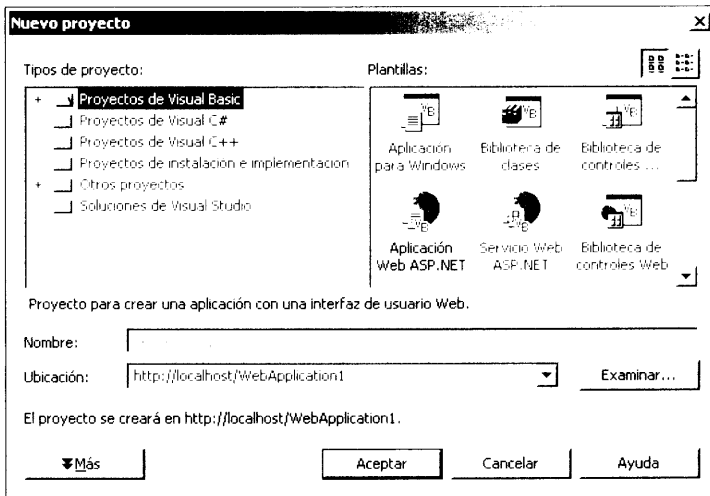


Figura 22.2. Seleccione una plantilla y especifique el nombre de la aplicación en el cuadro de diálogo **Nuevo proyecto**.

TRUCO: Si el servidor **Web** está instalado en su equipo, también puede escribir **http://localhost** in en el cuadro **Ubicación**.

6. Haga clic en **Aceptar**. Aparecerá un cuadro de diálogo mientras **Visual Studio .NET** crea la nueva aplicación **Web ASP.NET**. El cuadro de diálogo se muestra en la figura 22.3.

NOTA: Quizás tenga que esperar unos instantes a que **ASP.NET** cree el proyecto. También debe asegurarse, antes de crear la aplicación, de que el servidor **Web** está funcionando. Para activar el servidor **Web**, seleccione **Inicio>Ejecutar**. En el cuadro de diálogo **Ejecutar**, escriba **net start iisadmin** y pulse **Intro**.



Figura 22.3. Visual Studio .NET crea el nuevo proyecto.

Tras crear un nuevo proyecto de aplicación Web, el asistente para aplicación Web crea automáticamente algunos archivos de proyecto necesarios, como `AssemblyInfo.cs`, `Web.config` y `Global.asax`, junto al archivo principal de la página, `WebForm1.aspx`. La figura 22.4 muestra estos archivos en el explorador de soluciones.



Figura 22.4. La ventana de proyecto muestra todos los archivos creados por el asistente de la aplicación Web.

Un WebForm de Visual Studio .NET tiene dos vistas: *Diseño* y *HTML*:

- **Vista Diseño.** La interfaz de usuario de un WebForm se diseña en vista Diseño. Esta vista ofrece dos diseños para el WebForm: diseño de cuadrícula y diseño de flujo:
 - **Diseño de cuadrícula.** En el diseño de cuadrícula, se pueden colocar los controles en el WebForm según las coordenadas de cada control.
 - **Diseño de flujo.** En el diseño de flujo, se puede diseñar de forma lineal un WebForm, del mismo modo que se diseña un documento de Microsoft Word, de arriba abajo.

Puede alternar entre los diferentes diseños haciendo clic con el botón derecho en el WebForm y seleccionando **Propiedades**. En la página **Propiedades** puede seleccionar el diseño que considere apropiado.

- **Vista HTML.** Esta vista representa la correspondiente sintaxis ASP.NET del WebForm. Para abrir la vista HTML, sólo tiene que hacer clic en la

ficha HTML. Si la ficha HTML no está visible, puede hacer clic con el botón derecho del ratón y seleccionar Ver fuente HTML en el menú abreviado.

Cómo agregar controles al WebForm

Se pueden agregar controles a un WebForm de dos maneras:

- **Mediante el cuadro de herramientas.** Puede agregar controles en la vista Diseño del WebForm (el archivo .aspx) usando el cuadro de herramientas incluido en Visual Studio .NET. Dentro del cuadro de herramientas hay varios tipos de controles clasificados dentro diferentes fichas, como WebForms, HTML y Datos. Por ejemplo, puede usar la ficha HTML para crear controles de servidor HTML y la ficha WebForms para crear los controles de servidor ASP.NET. Todos los controles estudiados hasta ahora pertenecen a la ficha Windows Forms del cuadro de herramientas. Cuando se usa el cuadro de herramientas para agregar controles Web en vista Diseño, se genera automáticamente la sintaxis C# correspondiente.

TRUCO: Al usar controles HTML debe convertirlos a controles de servidor con el fin de que estén disponibles para ser codificados en el servidor. Para ello, haga clic con el botón derecho en el control HTML requerido y seleccione la opción Ejecutar como control del servidor en el menú abreviado. Este método permite crear complicados WebForms cómoda y rápidamente.

- **Usar sintaxis de Visual C# para agregar los controles mediante programación.** También puede agregar controles Web a un WebForm mediante la sintaxis de Visual C#. Sólo se puede usar la sintaxis de C# en la vista HTML de la página (archivo .aspx). La sintaxis real depende del tipo de control que quiera añadir. Por ejemplo, la sintaxis que se usa para agregar un control de cuadro de texto HTML es la siguiente:

```
<input id=Text1 Type=text runat="server">
```

NOTA: Visual Studio .NET permite agregar controles de servidor ASP.NET mediante una etiqueta de lenguaje extensible para el análisis de documentos (XML). La sintaxis usada para agregar un cuadro de texto ASP.NET es:

```
<asp:TextBox id=TextBox1 runat="server"></asp:TextBox>
```

Cada control tiene una propiedad ID que se usa para identificar unívocamente al control.

Para establecer la propiedad de un control en tiempo de ejecución se usa la siguiente sintaxis:

```
Control_ID.Property=Value
```

En la anterior sintaxis:

- `Control_ID` representa la propiedad ID del control.
- `Property` representa la propiedad del control.
- `Value` representa el valor asignado a la propiedad del control.

La figura 22.5 muestra un WebForm que contiene los habituales controles Web como etiquetas, cuadros de texto, hipervínculos, botones de opción, casillas de verificación y botones. Como puede ver, el WebForm es un formulario de registro de usuario. El formulario está diseñado para aceptar entradas de usuario desde varios controles. Tras rellenar el formulario, el usuario puede hacer clic en el botón **Submit** (Enviar) para completar el registro. El botón **Submit** abre otro WebForm que muestra un mensaje junto al nombre que el usuario introdujo en el control `TextBox`. Si el usuario hace clic en el botón **Reset** (Restablecer), la información que ha introducido el usuario se elimina de los controles de ese formulario.

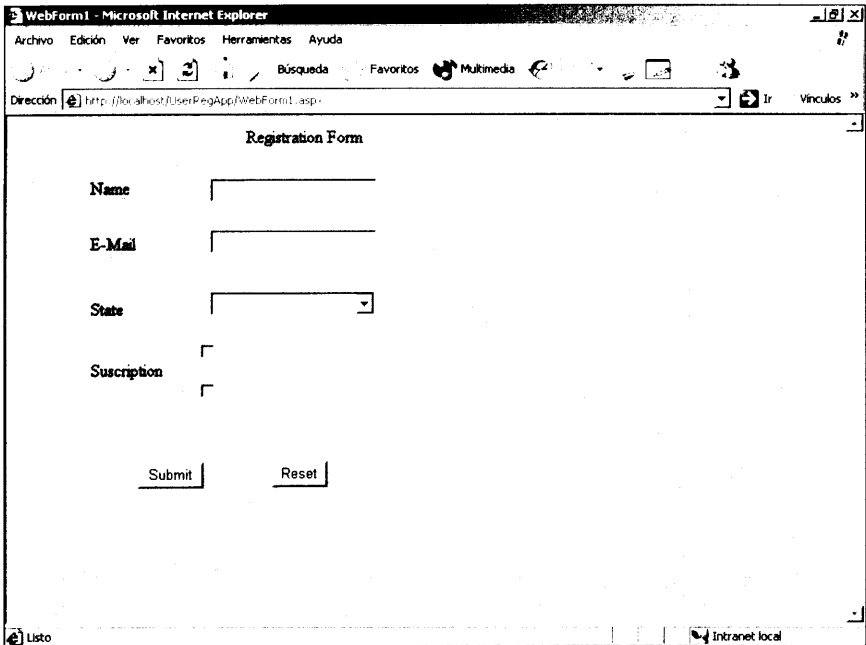


Figura 22.5. El formulario de registro de usuario muestra los controles comunes que se pueden agregar a un WebForm.

Para crear el formulario mostrado en la figura 22.5, realice los siguientes pasos:

1. Seleccione el formulario WebForm1.aspx y pulse F4 para que aparezca la ventana **Propiedades**.
2. En la ventana **Propiedades**, pulse el botón de puntos suspensivos para que aparezca la propiedad `bgColor` y seleccione **Propiedades**. Se abrirá el cuadro de diálogo **Selector de colores**.
3. En el cuadro de diálogo **Selector de colores**, seleccione un matiz rosa y haga clic en **Aceptar**. El color del WebForm cambiará al color que ha especificado.
4. Agregue controles al formulario y cambie sus propiedades, como recoge la tabla 22.2.

Tabla 22.2. Controles que se pueden agregar a un WebForm

Control	Propiedad	Posición
Label	Text=Registration Form Font Bold=True Size=Larger	Para situarlo en la parte superior, inferior o en el centro del formulario
Etiquetas para Nombre, Correo electrónico, Estado y Suscripción	El texto de cada etiqueta debe ser la misma que el título deseado.	Una bajo otra en el lado izquierdo de la pantalla
TextBox	ID=txtName	Junto a la etiqueta Name
TextBox	ID=txtEmail	Junto a la etiqueta E-Mail
DropDownList	ID=lstState Items=Arizona, California, Florida	Junto a la etiqueta State
CheckBoxList	ID=lstOptions Items=Books, Magazines	Junto a la etiqueta Suscription
Button	ID=BtnSubmit	Bajo la etiqueta Suscription
Button	Text=Reset ID=BtnReset Text=Reset	Junto al botón Submit

La interfaz de su WebForm, como aparece en la figura 22.6, está lista.

Agregará la funcionalidad de los botones **Submit** y **Reset** en la siguiente sección. Sin embargo, antes de continuar, agregue a la aplicación Web otro formulario que muestre los detalles sobre el usuario registrado cuando éste haga clic en el botón **Submit**. Para agregar el WebForm, siga estos pasos:

1. Seleccione Proyecto>Agregar WebForm. Se abrirá el cuadro de diálogo Agregar nuevo elemento.

TRUCO: Si no encuentra la opción de menú **Agregar WebForm** bajo el menú **Proyecto**, haga clic en cualquier parte de la ventana **Formulario** y a continuación seleccione la opción de menú.

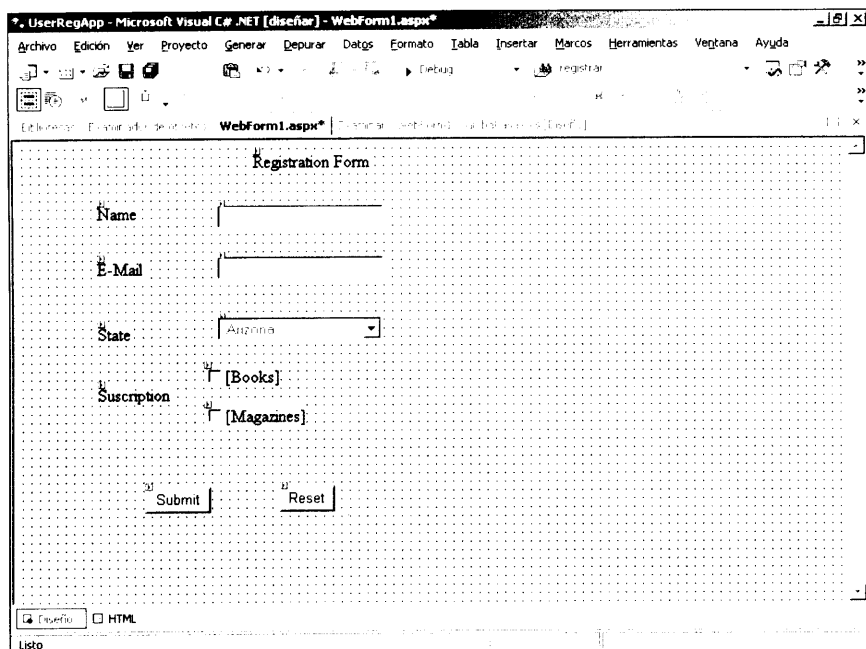


Figura 22.6. El WebForm debería tener este aspecto una vez completa.

2. Seleccione WebForm en la lista de Plantillas, especifique un nombre para el WebForm (o mantenga el nombre por defecto) y haga clic en **Abrir** para crear un nuevo WebForm.

Puede usar el recién agregado WebForm para mostrar un mensaje al usuario. Por lo tanto, necesitará agregar un control `Label` al formulario. Llame a la etiqueta `lblMessage`.

Tras agregar controles al formulario, debe responder a los eventos generados por los controles del formulario para trabajar con la interacción del usuario. Por

ejemplo, si un usuario hace clic en el botón **Submit**, el formulario necesitará ser procesado y los datos de la base de datos tendrán que actualizarse.

La siguiente sección describe el procedimiento para controlar los eventos generados por los controles de un WebForm.

Cómo controlar eventos

Cuando los usuarios interactúan con los diferentes controles Web de una página, se desencadenan eventos. Un evento es una acción que puede tener lugar sobre un objeto o sobre un control, que pueden ser generados por una acción de un usuario o por el sistema. Por ejemplo, cuando pulsa un botón del ratón o una tecla, se genera un evento.

En los formularios de cliente tradicionales o en las aplicaciones Web basadas en clientes, los eventos se desencadenan y gestionan por parte del cliente. En las aplicaciones Web, los eventos se desencadenan en el cliente o en el servidor. Sin embargo, los eventos generados siempre son controlados en el servidor. Los controles de servidor ASP.NET sólo admiten eventos de servidor, mientras que los controles de servidor admiten eventos de servidor y de cliente.

Viajes de ida y vuelta

Los WebForms son procesados en el servidor. Por ejemplo, imagine un formulario de registro de usuario. Cuando un nuevo usuario especifica un valor para el nombre de registro, el servidor debe asegurarse de que el nombre de registro proporcionado por el usuario es único. Puede asegurarse de que el nombre de registro es único interceptando el evento `Click` de un botón y comparando el nombre de usuario con una fuente de datos.

Cada vez que una interacción de usuario requiere ser procesada por el servidor, el WebForm es enviado al servidor y procesado, entonces el resultado es devuelto al cliente por medio de un navegador. Esta secuencia de procesamiento de información en el servidor recibe el nombre de proceso de *ida y vuelta*, como muestra la figura 22.7.

Casi todas las interacciones con los controles del servidor dan como resultado viajes de ida y vuelta. Como un viaje de ida y vuelta implica enviar el WebForm al servidor y luego mostrar el formulario procesado en el navegador, el control del servidor afecta al tiempo de respuesta en el WebForm. Por tanto, el número de eventos disponibles en los controles de servidor de un WebForm deben ser los menos posibles. Por lo general, esto se reduce a los eventos `Click`.

NOTA: Los eventos que tienen lugar con bastante frecuencia en lenguajes de secuencia de comandos, como `OnMouseOver`, no son compatibles con los controles de servidor. Sin embargo, algunos controles de servidor admiten eventos que tienen lugar cuando cambia el valor del control.

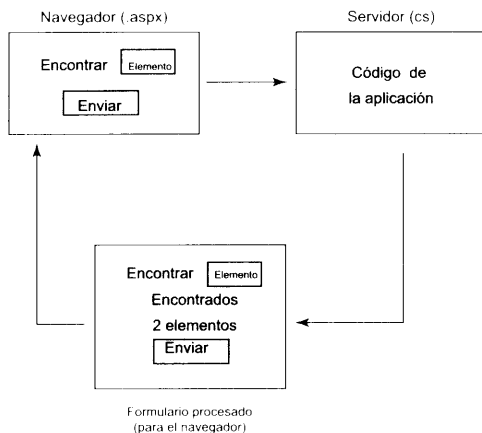


Figura 22.7. El proceso de ida y vuelta

La tabla 22.3 describe los eventos más comunes asociados a diferentes controles de servidor ASP.NET.

Tabla 22.3. Eventos asociados a controles de servidor ASP.NET

Control(es)	Evento	Descripción
TextBox	TextChanged	Tiene lugar cuando el foco sale del control.
RadioButton y CheckBox	CheckedChanged	Tiene lugar cuando se hace clic en el control.
RadioButtonList, CheckBoxList, ListBox y DropDownList	SelectedIndexChanged	Tiene lugar cuando se cambia la selección de la lista.
Button, LinkButton y ImageButton	Click	Tiene lugar cuando se hace clic en el botón. Este evento hace que el formulario sea enviado al servidor.

Por defecto, en un WebForm, sólo el evento `Click` de los controles de servidor `Button`, `LinkButton` y `ImageButton` pueden hacer que se envíe el formulario al servidor para ser procesado. En ese caso, el WebForm es devuelto al servidor. Cuando otros controles generan los eventos de cambio, son atrapados y ocultados. No hacen que el formulario se envíe inmediatamente. Solamente cuando el formulario es devuelto mediante un clic en un botón, todos los eventos ocultos se desencadenan y son procesados. No hay una secuencia particular para procesar estos eventos de cambio en el servidor. Sin embargo, el evento `Click`

se procesa solamente después de que todos los otros eventos de cambio hayan sido procesados.

Controladores de eventos

Cuando se desencadena un evento, éste necesita ser controlado para ser procesado posteriormente. Los procedimientos que se ejecutan cuando ocurre un evento reciben el nombre de *controladores de eventos*. Los controladores de eventos pueden ser creados automática o manualmente.

Cuando los eventos se controlan automáticamente, al hacer doble clic sobre un control en la vista Diseño del WebForm (.aspx file) se crea un controlador de eventos. Por ejemplo, cuando se hace doble clic sobre el botón `btnSubmit`, se genera el siguiente código. A continuación puede escribir el código en el controlador de eventos de la función generada por Visual Studio .NET:

```
Public void btnSubmit_Click(Object sender, System.EventArgs e)
{
}
}
```

En el anterior código, el procedimiento `btnSubmit_Click` es el controlador de eventos del evento `Click` del botón. El procedimiento toma dos argumentos. El primero contiene el emisor de eventos. Un emisor de eventos es un objeto, como un formulario o un control, que puede generar eventos. El segundo argumento contiene información adicional asociada al evento, como las coordenadas de posición *x* e *y*, en las que se ha pulsado el botón del ratón.

Para crear manualmente un controlador de eventos, selecciónelo de la lista emergente de la ventana **Propiedades**.

Ya está preparado para implementar el control de eventos para el WebForm que aparece en la figura 22.7.

Al hacer clic en el botón **Submit**, aparece una nueva página (en este caso `WebForm2.aspx`), que muestra un mensaje de bienvenida junto al nombre del usuario registrado. Para implementar esta funcionalidad debe escribir el siguiente código en el evento `Click` del botón **Submit** del WebForm `WebForm1.aspx`:

```
private void BtnSubmit_Click(object sender, System.EventArgs e)
{
    Response.Redirect("WebForm2.aspx?strName="+ txtName.Text);
}
```

TRUCO: Para codificar el evento del botón **Submit**, haga doble clic en la vista **Diseño**.

En el anterior código, el método `Redirect` de la clase `HttpResponse` redirige al usuario a la página `WebForm2.aspx` y pasa el valor del parámetro `txtName` a la página de destino.

Tras pasar el valor del cuadro de texto `txtName`, debe inicializar `WebForm2` para controlar la cadena pasada desde el formulario de registro. Para hacerlo, `WebForm2.aspx` debe tener el siguiente código en el evento `Load`:

```
private void Page_Load(object sender, System.EventArgs e)
{
    lblMessage.Text="Hi! " + Request.QueryString.Get("strName");
}
```

TRUCO: Para codificar el evento `Load` del formulario `WebForm2.aspx`, haga doble clic en el formulario en vista `Diseño`.

En el anterior código, el título de la etiqueta `lblMessage` en el archivo `WebForm2.aspx` es el valor que se asigna al valor almacenado en la variable `strName`.

Cuando el usuario hace clic en el botón **Submit** de `WebForm1.aspx`, es redirigido a la página `WebForm2.aspx`, como muestra la figura 22.8.



Figura 22.8. Cuando un usuario es redirigido a otra página, el nombre del usuario se pasa en la cadena de consulta.

Cuando el usuario hace clic en el botón **Reset**, debe generarse un evento que elimine todos los controles rellenos por el usuario en `WebForm1.aspx`. Para implementar esta funcionalidad, codifique el evento `Click` del botón **Reset** como se indica a continuación:

```
private void BtnReset_Click(object sender, System.EventArgs e)
{
    txtName.Text="";
    txtEmail.Text="";
    lstState.ClearSelection();
    lstOptions.ClearSelection();
}
}
```

En el código anterior, cuando se hace clic en el botón **Reset** del formulario de registro, el formulario se reinicia y vuelve a su estado original.

Cómo controlar la devolución de datos

Como se menciona antes, sólo se devuelve un WebForm al servidor cuando se hace clic en un control `Button`, `LinkButton` o `ImageButton` control. Una vez que se ha enviado el formulario al servidor, es procesado allí. Puede controlar la devolución de datos correspondiente al clic de un botón de una de estas formas:

- Escribiendo un controlador de eventos para el evento `Click` del botón.
- Escribiendo el controlador de eventos para el evento `Load` del WebForm. El evento `Load` se genera cuando se abre el formulario. Puede usar la propiedad `IsPostBack` del evento `Load` para determinar si la página ha sido procesada por primera vez o si ha sido procesada por un clic de botón. El siguiente código muestra el controlador de eventos para un evento `Load` de un WebForm:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        //Evalua true la primera vez que el navegador llega a
        la página
    }
}
```

Cómo usar el estado de vista

En las aplicaciones Web tradicionales, cada vez que se procesa una página Web en el servidor, la página se crea desde cero. El servidor elimina la información de la página actual después de procesarla y envía la página al cliente (navegador). Como la información de la página no está guardada en el servidor, las páginas Web se llaman sin estado. Sin embargo, el marco de trabajo ASP.NET resuelve esta limitación y puede guardar la información de estado del formulario y sus controles. Para gestionar la información de estado, Visual Studio .NET proporciona las siguientes opciones:

- **Guardar el estado de vista.** Puede guardar el estado de vista de los controles de un objeto. En cada viaje de ida y vuelta, el estado del control del servidor puede cargarse desde el estado guardado de modo que el usuario

pueda ver todas las opciones que el usuario haya seleccionado con anterioridad.

- **StateBag.** La clase `StateBag` es el mecanismo de almacenamiento de los controles de servidor. Esta clase proporciona propiedades para almacenar información en pares clave-valor. Por ejemplo, si quiere almacenar datos especificados por el usuario para una página, puede usar una instancia de la clase `StateBag` para almacenar estos datos.

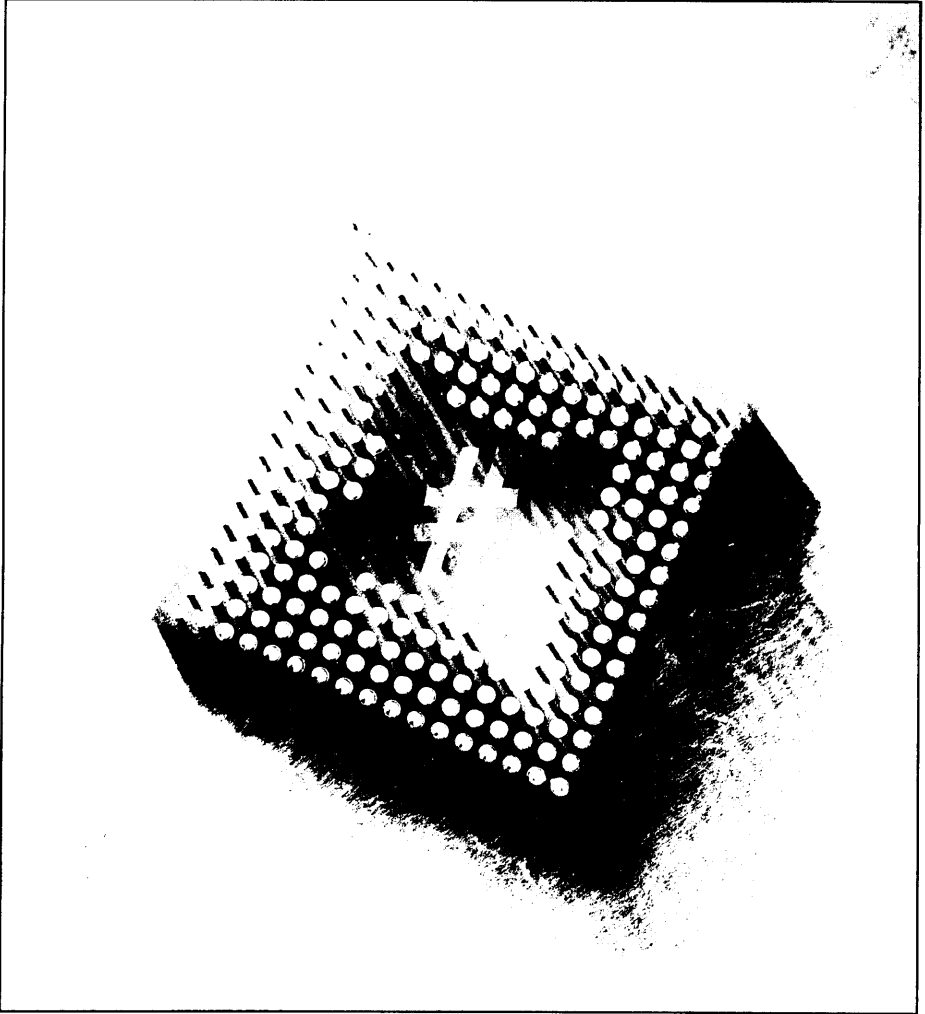
Cada control de servidor incluye una propiedad `EnableViewState`. Cuando establece el valor de esta propiedad como `true`, el estado del control se conserva en el servidor entre los viajes de ida y vuelta. Así, si el usuario ha seleccionado una o más opciones de una lista, las opciones se guardan en el servidor entre los viajes de ida y vuelta.

Resumen

En este capítulo, ha aprendido a crear una sencilla aplicación Web mediante Visual C# en el entorno ASP.NET. Se han estudiado los fundamentos de ASP.NET y cómo se crean aplicaciones Web en Visual C#.

ASP.NET incluye un entorno de tiempo de ejecución separado que gestiona la ejecución de las aplicaciones ASP.NET. También incluye nuevos componentes de servidor, llamados WebForms, que encapsulan la funcionalidad de una página Web. Puede agregar uno o más controles de servidor a un WebForm. Los controles de servidor son los responsables de mostrar los datos a los usuarios y procesar sus interacciones.

Hemos creado un proyecto de aplicación Web y le hemos agregado un WebForm. Al crear la aplicación, usamos la plantilla ASP.NET para aplicaciones Web para crear una solución y agregar un proyecto ASP.NET a la solución. A continuación, diseñamos una página Web usando controles Web corrientes, como los controles que representan etiquetas, cuadros de texto, cuadros de listas, hipervínculos, botones y similares. Por último, aprendimos a controlar los eventos que generan los controles en el WebForm.



23

Programación de bases de datos con ADO.NET

ADO.NET es la tecnología más moderna para el acceso de datos y forma parte de .NET Framework. ADO.NET utiliza y mejora las anteriores tecnologías de acceso a datos. La incursión de Microsoft en el acceso a datos universal comenzó con la conectividad abierta de bases de datos (ODBC). La idea sobre la que se asienta esta tecnología ODBC (crear un modo estándar de acceso a las bases de datos mediante programación) ha sido usada en todas las tecnologías de acceso a datos posteriores procedentes de Redmond, Washington (donde está la sede de Microsoft). En el caso de ODBC, este método estándar está ejemplificado en el API (Interfaz de programación de aplicaciones) de ODBC. Cualquier proveedor de bases de datos que quiera garantizar el cumplimiento del estándar ODBC debe elaborar el software que convierta una llamada ODBC (hecha de acuerdo con el API) en una llamada de una base de datos nativa. Este software recibe el nombre de *controlador ODBC* y es el puente entre una aplicación de cliente genérica y una base de datos específica. Mediante este enfoque, los programadores de aplicaciones evitan tener que aprender a usar el API de base de datos específico del proveedor. Todo lo que un programador necesita saber es cómo escribir aplicaciones cliente usando el API de ODBC. Este hecho mejora la productividad y permite escribir programas que pueden ser usados con diferentes bases de datos.

Sin embargo, el API de ODBC fue diseñado en un principio, sobre todo, con los programadores de C en mente y era difícil de usar en otros lenguajes (como

Visual Basic). Esto condujo finalmente a la creación de Objetos de datos de ActiveX (ADO), una tecnología de acceso a datos diseñada para ser usada con cualquier lenguaje compatible con el modelo de objetos componentes (COM) de Microsoft. ADO presenta un simple modelo de objetos que convierte el acceso a datos en los programas MS Windows sea una tarea sencilla. Además, ADO introduce el concepto de *conjuntos de datos sin conexión* como un modo de transportar datos entre los niveles de una aplicación distribuida. El API de bajo nivel detrás de ADO se llama OLE DB. Este API fue diseñado por programadores de C++ y es lo que los distribuidores de bases de datos suelen usar para escribir proveedores de OLE DB (el término más usado para referirse a los controladores OLE DB, el software que convierte las llamadas ADO en llamadas a bases de datos nativas). Microsoft también ha escrito un proveedor de OLE para ODBC. Este proveedor permite hacer llamadas ADO a cualquier base de datos que cumpla con el estándar ODBC.

Como verá en este capítulo, ADO.NET mantiene un modelo de objetos similar al de ADO y mejora el concepto de conjuntos de registros sin conexión proporcionando un modo de reunir más información en un objeto ADO.NET llamado *conjunto de datos*. De hecho, ADO.NET fue diseñado pensando en los datos sin conexión porque su falta de estado funciona mejor en las aplicaciones de Internet distribuidas. En este capítulo aprenderá a usar ADO.NET para manipular datos. Si ya conoce ADO, muchos de los conceptos le serán familiares e incluso el código le puede resultar conocido.

Clases DataSet y otras clases relacionadas

Esta sección estudia las clases de ADO.NET. Si ya conoce ADO reconocerá muchos de los conceptos que aquí le presentamos. Sin embargo, tenga en cuenta que algunos conceptos de ADO han mejorado mucho en ADO.NET y han aumentado considerablemente sus formas originales. Empecemos con un nuevo concepto: la clase `DataSet` y sus clases relacionadas. `DataSet` desarrolla el concepto de conjunto de registros de ADO. Los conjuntos de registros de ADO son una abstracción de un grupo de registros, como los datos resultantes recuperados al enviar una instrucción `Select SQL`. Un conjunto de registros puede contener más de un conjunto de registros, pero los registros son independientes entre sí y deben ser procesados secuencialmente invocando `NextRecordSet()`. `DataSet` de ADO.NET es una abstracción de toda una base de datos. Un `DataSet` no sólo permite contener más de un conjunto de registros (llamado apropiadamente `DataTable`), sino que además puede definir relaciones entre `DataTables`. La tabla 23.1 describe todas las clases relacionadas con el `DataSet`. Como las clases ADO, las clases ADO.NET usan muy a menudo las colecciones: la clase `DataSet` contiene una colección de `DataTables`; la clase `DataTable` contiene una colección de `DataColumns` y así sucesivamente.

Tabla 23.1. DataSet y clases relacionadas

Clase	Descripción
DataSet	Una caché en memoria de datos, que pueden consistir en varias <code>DataTables</code> relacionadas entre sí. Está diseñada para su uso sin conexión en las aplicaciones distribuidas.
DataTable	Un contenedor de datos, que puede componerse de varias <code>DataColumns</code> . Cada fila de datos se contiene en una <code>DataRow</code> .
DataRow	Una fila de datos concreta en una <code>DataTable</code>
DataColumn	La definición de una columna (nombre, datos, tipo, etc.) en una <code>DataTable</code>
DataRelation	Una relación entre dos <code>DataTables</code> de una <code>DataSet</code> , normalmente usada para representar relaciones de clases externas
Constraint	Una restricción a una o más <code>DataColumns</code> , usada para representar limitaciones como la exclusividad
DataColumnMapping	Asigna los nombres de columna desde la tabla de una base de datos hasta los nombres de columna de <code>DataTable</code> en el <code>DataSet</code> .
DataTableMapping	Asigna los nombres de tabla en una base de datos hasta los nombres de <code>DataTables</code> en <code>DataSet</code>
DataRowView	Una vista personalizada de una <code>DataRow</code> que puede usarse en la clasificación, filtrado y búsqueda.

El `RecordSet` de ADO evolucionó gradualmente como el modo estándar de preparar los datos entre los distintos niveles de una aplicación distribuida. El `DataSet` asume este papel en ADO.NET y proporciona varios métodos para compatibilizar la caché en memoria con su fuente de datos en una base de datos. Estos métodos incluyen `AcceptChanges()`, `GetChanges()`, `HasChanges()`, `HasErrors()` y `RejectChanges()`. Además, permiten recuperar cualquier cambio en forma de un `DataSet` modificado, examinar las modificaciones en busca de errores y decidir si aceptar o rechazar los cambios. Al final de este proceso puede actualizar los datos fuente en la base de datos con una simple llamada al método `Update()`.

Compatibilidad con OLE DB SQL Server

ADO.NET contiene dos conjuntos de clases similares. Una es un conjunto de clases genérico que puede usarse para acceder a todas las bases de datos de los

proveedores de OLE DB. Un segundo conjunto de clases ha sido optimizado para la base de datos insignia de Microsoft, SQL Server. Todos los nombres de las clases genéricas comienzan por `OleDb`. Todos nombres de las clases específicas de SQL Server empiezan con `Sql`. Cada clase genérica tiene su clase específica SQL Server correspondiente. Por ejemplo, la clase que se usa para ejecutar las instrucciones SQL en SQL Server recibe el nombre de `SqlCommand`. La clase genérica recibe el nombre de `OleDbCommand`.

Este capítulo emplea las clases genéricas, incluso para acceder a la base de datos SQL Server. Cuando escriba sus propias aplicaciones que accedan a SQL Server, deberá decidir si quiere emplear las clases específicas SQL Server, más rápidas, o las clases genéricas, que permiten intercambiar distribuidores cambiando la cadena de conexión. La elección se basa en velocidad o portabilidad. Las clases SQL Server llaman directamente al nivel nativo de la base de datos. Las clases genéricas usan `OleDb` y atraviesan un nivel COM antes de llamar al nivel nativo de la base de datos. El coste de este nivel adicional supone un descenso en el rendimiento.

Las clases `DataSet` se usan en conjunción con el proveedor de OLE DB y el proveedor de SQL Server. La tabla 23.2 enumera las clases específicas de cada proveedor. Muchas de ellas le resultarán conocidas a alguien que ya ha trabajado con ADO.

Tabla 23.2. DataSet y clases relacionadas

Clases de proveedores		Descripción
de SQL	de OLE DB	
<code>SqlCommand</code>	<code>OleDbCommand</code>	Un contenedor de clase para una instrucción SQL. La clase puede administrar instrucciones directas SQL como las instrucciones <code>SELECT</code> , <code>UPDATE</code> , <code>DELETE</code> o <code>INSERT</code> y una llamada al procedimiento almacenado.
<code>SqlCommandBuilder</code>	<code>OleDbCommandBuilder</code>	Usada para generar las instrucciones SQL <code>SELECT</code> , <code>UPDATE</code> , <code>DELETE</code> o <code>INSERT</code> .
<code>SqlConnection</code>	<code>OleDbConnection</code>	Una conexión a una base de datos.
<code>SqlDataAdapter</code>	<code>OleDbDataAdapter</code>	Un conjunto de instrucciones <code>SELECT</code> , <code>UPDATE</code> , <code>DELETE</code> , o <code>INSERT</code> y una conexión a una base de datos que puede usarse para completar un <code>DataSet</code> y actualizar la base de datos subyacente.
<code>SqlDataReader</code>	<code>OleDbDataReader</code>	Un conjunto de registros de datos sólo hacia adelante.

Clases de proveedores		Descripción
de SQL	de OLE DB	
SqlError	OleDbError	Un aviso o error devuelto por la base de datos, perteneciente a una colección <code>Errors</code> .
SqlException	OleDbException	El tipo de excepción iniciado cuando ocurren errores en la base de datos.
SqlParameter	OleDbParameter	Un parámetro para un procedimiento almacenado.
SqlTransaction	OleDbTransaction	Una transacción de base de datos.

En la siguiente sección estudiaremos el modo en el que estas clases trabajan con las clases comunes. `DataSet` y sus clases relacionadas, para realizar operaciones con las bases de datos comunes.

Operaciones de bases de datos comunes mediante ADO.NET

Cada uno de los ejemplos de esta sección omite las declaraciones `using` en beneficio de la sencillez. Se supone que las siguientes tres declaraciones de espacio de nombres están presentes en este capítulo:

- `using System;`
- `using System.Data;`
- `using System.Data.OleDb;`

Además, muchas funciones se han sacado del contexto de su clase. Se supone que las funciones están incluidas en el ámbito definido por la siguiente definición de clase:

```
namespace ADOdotNET
{
    class ADOdotNET
    {
        // NOTA: Coloque aquí la función
    }
}
```

Una vez hechos estos comentarios previos, podemos adentrarnos en ADO.NET. Una a una, esta sección examina cada categoría de operaciones que puede realizarse con ADO.NET:

- Operaciones que no devuelven filas.
- Operaciones que sólo devuelven una fila.
- Operaciones que sólo afectan a una fila.
- Operaciones que devuelven varias filas.
- Operaciones que afectan a varias filas.
- Operaciones que devuelven datos jerárquicos.

Operaciones que no devuelven filas

Muchas operaciones de SQL (por ejemplo, las instrucciones `Insert`, `Delete` y `Update`) devuelve sólo éxito o fracaso (o los números de filas afectados por la operación).

NOTA: El programador de SQL Server controla si el número de filas afectadas se devuelve desde un procedimiento almacenado mediante la instrucción `SET NOCOUNT [ON | OFF]`. Los programadores en SQL Server suelen desactivar esta característica mediante `SET NOCOUNT ON` porque mejora ligeramente el rendimiento.

El listado 23.3 muestra lo sencillo que resulta ejecutar una instrucción SQL que no devuelve filas en ADO.NET. En este proceso se emplean dos objetos. En primer lugar se usa un objeto `OleDbConnection` para establecer una conexión con una base de datos. El listado 23.1 muestra un ejemplo de cadena de conexión usada para acceder a la base de datos Northwind de una instrucción de SQL Server instalada localmente.

Listado 23.1. Ejemplo de cadena de conexión para SQL Server

```
private static string OleDbConnectionString
{
    get
    {
        // NOTA: Usar la cuenta sa para producir
        // aplicaciones es, por supuesto, una práctica
        // muy mala. Además, dejar en blanco la
        // contraseña para la cuenta sa es igualmente
        // inadmisible.
        return "Provider=SQLOLEDB.1;"
            +"User ID=sa;Initial Catalog=Northwind;Data
            Source=localhost;";
    }
}
```

El listado 23.2 muestra un ejemplo de cadena de conexión para Access 2000.

Listado 23.2. Ejemplo de cadena de conexión para Microsoft Access

```
private static string oleDbConnectionString
{
    get
    {
        // NOTA: Se presupone que se instaló
        // Microsoft Office Pro en el directorio por defecto.
        return "Provider=Microsoft.Jet.OLEDB.4.0;"
            +"Data Source=C:\\Program Files\\Microsoft
Office\\Office\\Samples\\Northwind.MDB";
    }
}
```

El segundo objeto que se usa para ejecutar una consulta es `OleDbCommand`. Su constructor recibe un argumento de cadena (el texto de la instrucción SQL que se quiere ejecutar) y un objeto `OleDbConnection`. La propiedad `CommandType` permite especificar si el comando que se ejecuta es un procedimiento almacenado, una consulta Access o texto simple. La consulta se realiza mediante el método `ExecuteNonQuery()`. Los errores, como una infracción de clave primaria, se notifican mediante excepciones.

También se pueden usar objetos `Command` para ejecutar comandos SQL que no devuelven filas de datos, como en el ejemplo del listado 23.3:

Listado 23.3. Plantilla para usar un comando y ejecutar una instrucción SQL que no devuelve filas

```
// Declare y asigne los valores apropiados para
oleDbConnectionString y strSQLStatement
// Crea objetos OleDb
OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
OleDbCommand databaseCommand = new
OleDbCommand(strSQLStatement, databaseConnection);
// NOTA: Solo se debe usar una de las dos instrucciones que se
muestran a continuación, NO AMBAS.
// Si estamos tratando con una instrucción SQL (es decir, NO
con un procedimiento almacenado), use:
    databaseCommand.CommandType = CommandType.Text;
// Si estamos tratando con procedimiento almacenado, use:
    databaseCommand.CommandType = CommandType.StoredProcedure;
try
{
    // Establece la conexión de base de datos
    databaseConnection.Open();
    // Ejecuta el comando de SQL
    int numRows = databaseCommand.ExecuteNonQuery();
    // Haz otra cosa, p. ej. informar sobre numRows
}
catch (Exception e)
{
```

```

    //Controla la excepcion, p. ej.:
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}

```

La invocación de procedimientos almacenados con parámetros es un poco más complicada. El listado 23.4 muestra el código SQL para un procedimiento almacenado al que se quiere llamar.

Listado 23.4. Un procedimiento almacenado de SQL Server para insertar un registro

```

USE [Northwind]
GO
CREATE PROCEDURE [pc_insCustomers]
    (@CustomerID_1 [nchar](5),
    @CompanyName_2 [nvarchar](40),
    @ContactName_3 [nvarchar](30),
    @ContactTitle_4 [nvarchar](30),
    @Address_5 [nvarchar](60),
    @City_6 [nvarchar](15),
    @Region_7 [nvarchar](15),
    @PostalCode_8 [nvarchar](10),
    @Country_9 [nvarchar](15),
    @Phone_10 [nvarchar](24),
    @Fax_11 [nvarchar](24))
AS
INSERT INTO [Northwind].[dbo].[Customers]
    ([CustomerID],
    [CompanyName],
    [ContactName],
    [ContactTitle],
    [Address],
    [City],
    [Region],
    [PostalCode],
    [Country],
    [Phone],
    [Fax])
VALUES
    (@CustomerID_1,
    @CompanyName_2,
    @ContactName_3,
    @ContactTitle_4,
    @Address_5,
    @City_6,
    @Region_7,
    @PostalCode_8,

```

```

    @Country_9,
    @Phone_10,
    @Fax_11)

```

La única parte complicada es saber cómo definir y establecer los parámetros. Esto se hace mediante la colección `Parameters`. Como en cualquier colección, los nuevos miembros se crean con el método `Add()`. El parámetro recién creado es devuelto y se puede establecer la dirección (tanto si el parámetro se usa sólo para introducir datos, para salida de datos o para ambas operaciones) y el valor. Los parámetros del método `Add()` son el nombre del parámetro del procedimiento almacenado, su nombre, su tipo de datos y su tamaño. El listado 23.5 muestra el código para todo el proceso.

Listado 23.5. Cómo llamar a un procedimiento almacenado con parámetros en ADO.NET

```

static void TestInsertWithSPStatement(string customerID)
{
    // Establece la cadena de instruccion SQL
    string strSQLInsert = "[pc_insCustomers]";

    // Crea objetos OleDb
    OleDbConnection databaseConnection = new
    OleDbConnection(oleDbConnectionString);
    OleDbCommand insertCommand = new OleDbCommand(strSQLInsert,
    databaseConnection);

    // Estamos tratando con una procedimiento almacenado (es decir,
    NO con una instruccion SQL)
    insertCommand.CommandType = CommandType.StoredProcedure;

    // Agregue cada parametro (1 de 11)
    OleDbParameter param =
    insertCommand.Parameters.Add("@CustomerID_1",
    OleDbType.VarChar, 5);
    param.Direction = ParameterDirection.Input;
    param.Value = customerID;
    // Agregue cada parametro (#2 de 11)
    param = insertCommand.Parameters.Add("@CompanyName_2",
    OleDbType.VarChar, 40);
    param.Direction = ParameterDirection.Input;
    param.Value = "Hungry Coyote Export Store";
    // Agregue cada parametro 3-10
    // Etc.
    // Agregue cada parametro (11 de 11)
    param = insertCommand.Parameters.Add("@Fax_11",
    OleDbType.VarChar, 24);
    param.Direction = ParameterDirection.Input;
    param.Value = "(503) 555-2376";

    try

```

```

{
    // Establezca la conexión de la base de datos
    databaseConnection.Open();

    // Ejecute el comando SQL
    int numPows = insertCommand.ExecuteNonQuery();

    // Informe de los resultados
    Console.WriteLine("Inserted {0} row(s).",
numPows.ToString());
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Operaciones de datos que devuelven entidades de fila única

Algunas operaciones de datos, como recuperar un registro basado en una clave primaria, sólo devuelven una fila. ADO.NET proporciona tres modos de recuperar una sola fila. Un modo sólo se aplica a las entidades de fila única y los otros dos modos son genéricos y pueden ser usados para recuperar varias filas (como verá en la siguiente sección).

El modo más eficiente de recuperar una entidad de fila única suele ser mediante un parámetro de salida. Sin embargo, este método sólo se puede usar cuando se está seguro de que el procedimiento devuelve una sola fila. El listado 23.6 muestra un procedimiento SQL Server almacenado que recupera un único registro mediante parámetros de salida.

Listado 23.6. Un procedimiento SQL Server almacenado para recuperar un único registro

```

USE [Northwind]
GO
CREATE PROCEDURE [pc_getContact_ByCustomerID]
    (@CustomerID_1 [nchar] (5),
    @ContactName_2 [nvarchar] (30) output,
    @ContactTitle_3 [nvarchar] (30) output)
AS
SELECT
    @ContactName_2 = [ContactName],
    @ContactTitle_3 = [ContactTitle]

```

```

FROM [Northwind].[dbo].[Customers]
WHERE
    [CustomerID] = @CustomerID_1

```

La invocación de este procedimiento almacenado es similar al código usado para llamar al procedimiento almacenado que inserta una fila (véase el listado 23.3). Por supuesto, la dirección para los parámetros de salida se establece en `ParameterDirection.Output`. Tras ejecutar el procedimiento almacenado, puede usar la colección `Parameters` para recuperar los valores de los parámetros de salida, como muestra el listado 23.7.

Listado 23.7. Cómo recuperar un único registro mediante parámetros de salida

```

static void TestSPWithOutParam(string customerID)
{
    // Establezca las cadenas de la instrucción SQL
    string strSQLSelect = "[pc_getContact_ByCustomerID]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);

    // Estamos tratando con un procedimiento almacenado (es
decir, NO una instrucción SQL)
    selectCommand.CommandType = CommandType.StoredProcedure;

    // Agregue cada parametro (1 de 3)
    OleDbParameter param =
selectCommand.Parameters.Add("@CustomerID_1",
OleDbType.VarChar, 5);
    param.Direction = ParameterDirection.Input;
    param.Value = customerID;
    // Agregue cada parametro (2 de 3)
    param = selectCommand.Parameters.Add("@ContactName_2",
OleDbType.VarChar, 30);
    param.Direction = ParameterDirection.Output;
    // Agregue cada parametro (3 de 3)
    param = selectCommand.Parameters.Add("@ContactTitle_3",
OleDbType.VarChar, 30);
    param.Direction = ParameterDirection.Output;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        selectCommand.ExecuteNonQuery();

        // Informe de los resultados

```

```

        string contactName =
selectCommand.Parameters["@ContactName_2"].Value.ToString();
        string contactTitle =
selectCommand.Parameters["@ContactTitle_3"].Value.ToString();
        Console.WriteLine("Contact name is {0}, title is {1}.",
contactName, contactTitle);
    }
    catch (Exception e)
    {
        Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
    }
    finally
    {
        databaseConnection.Close();
    }
}
}

```

Observe a hora los métodos genéricos de lectura de datos. El primero usa un objeto `OleDbDataReader`. El objeto `Command` tiene un método `ExecuteReader()`, que devuelve un objeto `OleDbDataReader`. A continuación puede usar el método `Read()` para recorrer el contenido del `DataReader`. `Read()` devuelve `True` cuando se encuentran datos durante la lectura y `False` en caso contrario. El listado 23.8 muestra cómo hacerlo. Observe que este ejemplo usa una instrucción SQL para acceder a un procedimiento almacenado sólo para mostrar un modo alternativo de llamar a un procedimiento almacenado. Esto sólo se hace con fines demostrativos, ya que es más eficiente llamar a un procedimiento almacenado de la forma que se muestra en el listado 23.7.

Listado 23.8. Cómo recuperar un registro único mediante `DataReader`

```

static void TestSelectWithDataReader(string customerID)
{
    // Establezca las cadenas de instruccion SQL, dando por
sentado que customerID no contiene comillas
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='" + customerID + "'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);

    // Estamos tratando con una instruccion SQL (es decir, NO con
un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexion de la base de datos

```

```

databaseConnection.Open();

// Ejecute el comando SQL
OleDbDataReader rowReader = selectCommand.ExecuteReader();

// Informe de los resultados
if (rowReader.Read())
{
    string contactName =
rowReader["ContactName"].ToString();
    string contactTitle =
rowReader["ContactTitle"].ToString();
    Console.WriteLine("Contact name is {0}, title is {1}.",
contactName, contactTitle);
}
else
{
    Console.WriteLine("No rows found!");
}
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

El otro método genérico de recuperar datos es mediante el versátil objeto `DataSet`. Como el objeto `DataSet` fue diseñado para ser usado independientemente de la fuente de datos que lo originó, no hay `OleDbDataSet` ni `SqlDataSet`, sólo un `DataSet`. Un `DataSet` se usa en conjunción con un `DataAdapter`. Un objeto `DataAdapter` se usa específicamente para almacenar datos (es decir, se usa `OleDbDataAdapter`) y contiene cuatro objetos de comando para realizar operaciones:

- `InsertCommand`
- `SelectCommand`
- `UpdateCommand`
- `DeleteCommand`

Tras seleccionar el objeto `SelectCommand` se puede emplear el método `Fill()` para completar un `DataSet`. La siguiente sección muestra cómo usar los otros tres comandos para modificar los datos que contiene un `DataSet`. Un `DataSet` contiene uno o más objetos `DataTable`. Cada `DataTable` contie-

ne uno o más objetos DataRow. Estos objetos DataRow se almacenan en la colección Rows del DataSet. Un objeto DataRow contiene uno o más objetos DataColumn. Estos objetos DataColumn se almacenan en la colección Columns del DataRow. Las colecciones Row y Column son indizadas mediante el índice y el nombre. De hecho, puede imaginar el objeto DataSet como una base de datos en memoria. El listado 23.9 muestra un programa de ejemplo que recupera un único registro usando un objeto DataSet.

Listado 23.9. Cómo recuperar un registro único mediante DataSet

```
static void TestSelectWithDataSet(string customerID)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='" + customerID + "'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con una instrucción SQL (es decir, NO con
un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        dsCmd.SelectCommand = selectCommand;
        int numRows = dsCmd.Fill(resultDataSet, "Customers");

        // Informe de los resultados
        if (numRows > 0)
        {
            string contactName = resultDataSet.Tables["Customers"].
Rows[0]["ContactName"].ToString();
            string contactTitle = resultDataSet.Tables["Customers"].
Rows[0]["ContactTitle"].ToString();

            Console.WriteLine("Contact name is {0}, title is {1}.",
contactName, contactTitle);
        }
        else
        {
            Console.WriteLine("No rows found!");
        }
    }
}
```

```

    }
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}
}

```

Operaciones de datos que afectan a las entidades de fila única

Esta sección estudia las propiedades `InsertCommand`, `UpdateCommand` y `DeleteCommand` del objeto `DataAdapter`. Para cada uno de estos comandos se puede establecer el comando mediante programación o se puede generar automáticamente. El primer método suele dar como resultado un mejor rendimiento porque supone menos encabezados.

Operaciones de introducción de datos que afectan a las entidades de fila única

El código del listado 23.10 usa un idioma automático que es útil para generar automáticamente instrucciones `Insert`. Mediante `SelectCommand` se consigue un `DataSet` vacío. Esta llamada `Fill()` sólo sirve para recuperar la estructura de las filas que quiere manipular. Esto es más flexible que definir la estructura mediante programación.

El secreto para generar automáticamente comandos `DataAdapter` es crear un objeto `CommandBuilder` que usa el `DataAdapter` como un argumento en el constructor, como muestran las siguientes líneas:

```

// ¡¡¡La siguiente línea es clave para generar instrucciones
automaticamente!!! OleDbCommandBuilder custCB = new
OleDbCommandBuilder(dsCmd);

```

Sin estas líneas, la posterior instrucción `Update()` funcionará mal porque `InsertCommand` no puede generarse automáticamente. Las operaciones generadas automáticamente `UpdateCommand` y `DeleteCommand` tienen los mismos requisitos.

El listado 23.10 muestra todo el procedimiento en funcionamiento. Tras recuperar la estructura de la tabla `Customers` mediante un procedimiento almacenado (que aparece en el listado 23.11), se crea una nueva fila mediante una llamada al método `NewRow()`. A continuación se asigna un valor a cada columna de la nueva fila y esa fila recién completada se agrega a la colección `Rows`

mediante una llamada al método `AddRow()`. Finalmente, este cambio se envía a la fuente de datos creada mediante una llamada del `DataAdapter` al método `Update()`.

Listado 23.10. Cómo agregar un registro único mediante un comando `InsertCommand` generado automáticamente

```
static void TestAutoInsertWithDataSet(string customerID)
{
    // Establezca las cadenas de instruccion SQL, solo
    necesitamos los metadatos de modo que
    // ningun registro concuerde con este CustomerID.
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='???'";

    // Crea objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    // ;;;La siguiente linea es clave para generar instrucciones
    automaticamente!!!
    OleDbCommandBuilder custCB = new OleDbCommandBuilder(dsCmd);
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con una instruccion SQL (es decir, NO con
    un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexion de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        dsCmd.SelectCommand = selectCommand;
        // Pecupera la estructura de la tabla Customers
        int numRows = dsCmd.Fill(resultDataSet, "Customers");

        // Cree una nueva fila
        DataRow workRow =
resultDataSet.Tables["Customers"].NewRow();

        // Complete los datos workrow
        workRow["CustomerID"] = customerID;    // 1
        workRow["CompanyName"] = "Hungry Coyote Export Store";
// 2
        workRow["ContactName"] = "Yoshi Latimer";    // 3
        workRow["ContactTitle"] = "Sales Representative";    // 4
        workRow["Address"] = "City Center Plaza 516 Main St.";
// 5
    }
}
```

```

workRow["City"] = "Elgin"; // 6
workRow["Region"] = "OR"; // 7
workRow["PostalCode"] = "97827"; // 8
workRow["Country"] = "USA"; // 9
workRow["Phone"] = "(503) 555-6874"; // 10
workRow["Fax"] = "(503) 555-2376"; // 11

resultDataSet.Tables["Customers"].Rows.Add(workRow);

// Compatibilice los cambios con la fuente de datos
dsCmd.Update(resultDataSet, "Customers");

// Informe de los resultados
Console.WriteLine("Inserted 1 row.");
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Listado 23.11. Procedimiento almacenado para recuperar la estructura de la tabla Customers

```

USE [Northwind]
GO
CREATE PROCEDURE [pc_getCustomer_ByCustomerID]
    (@CustomerID_1 [nchar](5))
AS
SELECT
    [CustomerID],
    [CompanyName],
    [ContactName],
    [ContactTitle],
    [Address],
    [City],
    [Region],
    [PostalCode],
    [Country],
    [Phone],
    [Fax]
FROM [Northwind].[dbo].[Customers]
WHERE
    [CustomerID] = @CustomerID_1

```

Hay que definir mediante programación un comando que realice la introducción de datos. Ya aprendió a hacerlo con un procedimiento almacenado con

parámetros (por cierto, uno de los modos más eficientes de realizar operaciones con datos) en el listado 23.5. Tras definir el comando y sus parámetros, todo lo que debe hacer es establecer el `InsertCommand` de `DataAdapter`, como muestra el listado 23.12. El resto del código (crear una nueva fila, dar valores a las columnas, agregar la nueva fila y actualizar el código fuente) es igual que el código usado para un `InsertCommand` generado automáticamente. Como se usa un enfoque igual a la creación manual de un comando para `UpdateCommand` y `DeleteCommand`, la siguiente sección sólo muestra cómo usar los comandos generados automáticamente.

Listado 23.12. Cómo agregar un registro único mediante `InsertCommand`

```
static void TestDataSetInsertCommand(string customerID)
{
    // Establezca las cadenas de instruccion SQL
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='???'";
    string strSQLInsert = "[pc_insCustomers]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con una instruccion SQL (es decir, NO con
un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    OleDbCommand insertCommand = new OleDbCommand(strSQLInsert,
databaseConnection);
    insertCommand.CommandType = CommandType.StoredProcedure;
    insertCommand.CommandText = "[pc_insCustomers]";
    insertCommand.Connection = databaseConnection;

    // Agregue cada parametro (1 de 11)
    OleDbParameter param =
insertCommand.Parameters.Add("@CustomerID_1",
OleDbType.VarChar, 5);
    param.Direction = ParameterDirection.Input;
    param.SourceColumn = "CustomerID";
    // Agregue cada parametro (2 de 11)
    param = insertCommand.Parameters.Add("@CompanyName_2",
OleDbType.VarChar, 40);
    param.Direction = ParameterDirection.Input;
    param.SourceColumn = "CompanyName";
    // Agregue cada parametro 3-10
    // Etc.
    // Agregue cada parametro (11 de 11)
```

```

param = insertCommand.Parameters.Add("@Fax_11",
OleDbType.VarChar, 24);
    param.Direction = ParameterDirection.Input;
    param.SourceColumn = "Fax";

try
{
    // Establezca la conexión de la base de datos
    databaseConnection.Open();

    // Ejecute el comando SQL
    dsCmd.SelectCommand = selectCommand;
    dsCmd.InsertCommand = insertCommand;

    int numRows = dsCmd.Fill(resultDataSet, "Customers");

    // Cree una nueva fila
    DataRow workRow =
resultDataSet.Tables["Customers"].NewRow();

    // Complete los datos workrow
    workRow["CustomerID"] = customerID;    // 1
    workRow["CompanyName"] = "Hungry Coyote Export Store";
// 2
    workRow["ContactName"] = "Yoshi Latimer";    // 3
    workRow["ContactTitle"] = "Sales Representative";    // 4
    workRow["Address"] = "City Center Plaza 516 Main St.";
// 5
    workRow["City"] = "Elgin";    // 6
    workRow["Region"] = "OR";    // 7
    workRow["PostalCode"] = "97827";    // 8
    workRow["Country"] = "USA";    // 9
    workRow["Phone"] = "(503) 555-6874";    // 10
    workRow["Fax"] = "(503) 555-2376";    // 11

    resultDataSet.Tables["Customers"].Rows.Add(workRow);

    // Compatibilice los cambios con la fuente de datos
    dsCmd.Update(resultDataSet, "Customers");

    // Informe de los resultados
    Console.WriteLine("Inserted 1 row.");
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Operaciones de actualización que afectan a entidades de fila única

Para realizar operaciones que tienen lugar mediante un `DataSet`, obviamente es necesario recuperar antes la fila que se quiere modificar. Por tanto, no es necesario el truco sugerido para la instrucción `Insert`. Tras recuperar un `DataSet`, se puede simplemente actualizar una columna de una fila específica. En ese momento se puede llamar al método `Update()` de `DataAdapter` para propagar los cambios en la fuente de datos. Por ahora, el contenido del listado 23.13 le resultará conocido. Como se señaló en la sección anterior, el no usar instrucciones generadas automáticamente sólo significa que deberá crear manualmente un comando que controle la instrucción `Update`.

Listado 23.13. Cómo actualizar un registro único mediante un `UpdateCommand` generado automáticamente

```
static void TestAutoUpdateWithDataSet(string customerID)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='" + customerID + "'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    // ¡¡¡La siguiente línea es clave para generar instrucciones
automáticamente!!!
    OleDbCommandBuilder custCB = new OleDbCommandBuilder(dsCmd);
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con una instrucción SQL (es decir, NO con
un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        dsCmd.SelectCommand = selectCommand;
        int numRows = dsCmd.Fill(resultDataSet, "Customers");

        // Informe de los resultados
        if (numRows > 0)
        {
            resultDataSet.Tables["Customers"].Rows[0]["ContactTitle"] =
            "Sr. Sales Representative";
        }
    }
}
```

```

        // Compatibilice los cambios con la fuente de datos
        dsCmd.Update(resultDataSet, "Customers");

        Console.WriteLine("1 row updated!");
    }
    else
    {
        Console.WriteLine("No rows found!");
    }
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Operaciones de borrado que afectan a las entidades de fila única

Por supuesto, no todas las operaciones que afectan a las entidades de fila única deben realizarse mediante un `DataSet`. También se puede usar un procedimiento almacenado o una instrucción SQL. El listado 23.14 muestra cómo usar una instrucción SQL para borrar una fila única en una tabla.

Listado 23.14. Cómo borrar un registro único mediante una instrucción SQL.

```

static void TestDeleteStatement(string customerID)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLDelete = "DELETE FROM Customers WHERE CustomerID
= '" + customerID + "'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand deleteCommand = new OleDbCommand(strSQLDelete,
databaseConnection);

    // Estamos tratando con una instrucción SQL (es decir, NO con
un procedimiento almacenado)
    deleteCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();
    }
}

```



```

        // Ejecute el comando SQL
        int numRows = deleteCommand.ExecuteNonQuery();

        // Informe de los resultados
        Console.WriteLine("Deleted {0} row(s).",
numPows.ToString());
    }
    catch (Exception e)
    {
        Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
    }
    finally
    {
        databaseConnection.Close();
    }
}

```

El listado 23.15 finaliza con el estudio de los comandos generados automáticamente, mostrándole cómo borrar una fila única usando este sistema. Tras completar un DataSet, puede eliminar una fila con una llamada Delete(). Como siempre, se necesita una llamada Update() para fijar este cambio en la fuente de datos.

Listado 23.15. Cómo borrar un registro único con un DeleteCommand generado automáticamente.

```

static void TestAutoDeleteWithDataSet(string customerID)
{
    // Establezca las cadenas de instruccion SQL
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='" + customerID + "'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    // ;;;La siguiente linea es clave para generar instrucciones
automaticamente!!!
    OleDbCommandBuilder custCB = new OleDbCommandBuilder(dsCmd);
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con una instruccion SQL (es decir, NO con
un procedimiento almacenado)
    selectCommand.CommandType = CommandType.Text;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();
    }
}

```

```

// Ejecute el comando SQL
dsCmd.SelectCommand = selectCommand;
int numRows = dsCmd.Fill(resultDataSet, "Customers");

// Informe de los resultados
if (numRows > 0)
{
    resultDataSet.Tables["Customers"].Rows[0].Delete();

    // Compatibiliza los cambios con la fuente de datos
    dsCmd.Update(resultDataSet, "Customers");

    Console.WriteLine("1 row deleted!");
}
else
{
    Console.WriteLine("No rows found!");
}
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Operaciones de datos que devuelven conjuntos de filas

Ya vimos dos maneras de recuperar conjuntos de filas cuando explicamos cómo recuperar una fila única. El listado 23.16 usa un procedimiento almacenado para explicar la recuperación de un conjunto de filas. Usa una instrucción TOP 5 para mantener el número de filas devueltas en un número aceptable. La única diferencia reseñable entre el listado 23.17 y el listado 23.8 es el uso del bucle while (en lugar de la instrucción if) para recorrer todos los registros.

Listado 23.16. Procedimiento almacenado SQL Server para seleccionar un conjunto de registros

```

USE [Northwind]
GO
CREATE PROCEDURE [pc_getCustomers]
AS
SELECT TOP 5
    [CustomerID],
    [CompanyName],

```

```

    [ContactName],
    [ContactTitle],
    [Address],
    [City],
    [Region],
    [PostalCode],
    [Country],
    [Phone],
    [Fax]
FROM [Northwind].[dbo].[Customers]

```

Listado 23.17. Cómo recuperar un conjunto de registros con DataReader

```

static void TestSelectManyWithDataReader(string customerID)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "[pc_getCustomers]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);

    // Estamos tratando con procedimientos almacenados (es decir,
NO con una instrucción SQL)
    selectCommand.CommandType = CommandType.StoredProcedure;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecuta el comando SQL
        OleDbDataReader rowReader = selectCommand.ExecuteReader();

        // Informe de los resultados
        while(rowReader.Read())
        {
            string contactName =
rowReader["ContactName"].ToString();
            string contactTitle =
rowReader["ContactTitle"].ToString();
            Console.WriteLine("Contact name is {0}, title is {1}.",
contactName, contactTitle);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
    }
    finally

```

```

    {
        databaseConnection.Close();
    }
}

```

El uso de un objeto `DataSet` para recuperar un conjunto de registros también le resultará familiar (véase el listado 23.9). De nuevo, todo lo que tiene que hacer es añadir una iteración para capturar todos los registros recuperados. Esto puede hacerse con un bucle `for`, como muestra el listado 23.18.

Listado 23.18. Cómo recuperar un conjunto de registros con `DataSet`.

```

static void TestSelectManyWithDataSet(string customerID)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "[pc_getCustomers]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con procedimientos almacenados (es decir,
NO con una instrucción SQL)
    selectCommand.CommandType = CommandType.StoredProcedure;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        dsCmd.SelectCommand = selectCommand;
        int numRows = dsCmd.Fill(resultDataSet, "Customers");

        // Informe de los resultados
        if (numRows > 0)
        {
            // numRows =
resultDataSet.Tables["Customers"].Rows.Count
for (int i=0; i<= numRows - 1; i++)
            {
                string contactName
= resultDataSet.Tables["Customers"].Rows[i]["ContactName"].
ToString();
                string contactTitle
= resultDataSet.Tables["Customers"].Rows[i]["ContactTitle"].
ToString();
            }
        }
    }
}

```

```

        Console.WriteLine("Contact name is {0}, title is
{1}.", contactName, contactTitle);
    }
    else
    {
        Console.WriteLine("No rows found!");
    }
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}
}

```

Operaciones de datos que afectan a conjuntos de filas

Las operaciones de datos que afectan a conjuntos de filas siguen la misma estructura que las operaciones que afectan a las filas únicas. El listado 23.19 agrega dos nuevas filas antes de llamar al comando `Update()`. Si lo comparamos con el listado 23.10 no aparece ninguna diferencia, aparte de la obvia adición del código que crea, da valor y agrega la segunda fila. Debido a las similitudes entre el código para una fila única y el código para filas múltiples, esta sección no repetirá todos los ejemplos mostrados con anterioridad para las instrucciones que afectan a varias filas `Update` y `Delete`.

Listado 23.19. Cómo agregar dos registros mediante un `InsertCommand` generado automáticamente

```

static void TestAutoInsert2WithDataSet(string customerID1,
string customerID2)
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "EXEC [pc_getCustomer_ByCustomerID]
@CustomerID_1='???'";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    // ;;;La siguiente línea es clave para generar instrucciones
automáticamente!!!

```

```

OleDbCommandBuilder custCB = new OleDbCommandBuilder(dsCmd);
DataSet resultDataSet = new DataSet();

// Estamos tratando con una instruccion SQL (es decir, NO con
un procedimiento almacenado)
selectCommand.CommandType = CommandType.Text;

try
{
    // Establezca la conexion de la base de datos
    databaseConnection.Open();

    // Ejecute el comando SQL
    dsCmd.SelectCommand = selectCommand;
    int numRows = dsCmd.Fill(resultDataSet, "Customers");

    // Cree una nueva primera fila
    DataRow workRow =
resultDataSet.Tables["Customers"].NewRow();

    // Complete los datos workrow
    workRow["CustomerID"] = customerID1;    // 1
    workRow["CompanyName"] = "Hungry Coyote Export Store";
// 2
    workRow["ContactName"] = "Yoshi Latimer";    // 3
    workRow["ContactTitle"] = "Sales Representative";    // 4
    workRow["Address"] = "City Center Plaza 516 Main St.";
// 5
    workRow["City"] = "Elgin";    // 6
    workRow["Region"] = "OR";    // 7
    workRow["PostalCode"] = "97827";    // 8
    workRow["Country"] = "USA";    // 9
    workRow["Phone"] = "(503) 555-6874";    // 10
    workRow["Fax"] = "(503) 555-2376";    // 11

    resultDataSet.Tables["Customers"].Rows.Add(workRow);

    // Cree una nueva segunda linea
    workRow = resultDataSet.Tables["Customers"].NewRow();

    // Complete los datos workrow
    workRow["CustomerID"] = customerID2;    // 1
    workRow["CompanyName"] = "Hungry Coyote Export Store";
// 2
    workRow["ContactName"] = "Yoshi Latimer";    // 3
    workRow["ContactTitle"] = "Sales Representative";    // 4
    workRow["Address"] = "City Center Plaza 516 Main St.";
// 5
    workRow["City"] = "Elgin";    // 6
    workRow["Region"] = "OR";    // 7
    workRow["PostalCode"] = "97827";    // 8
    workRow["Country"] = "USA";    // 9
    workRow["Phone"] = "(503) 555-6874";    // 10

```

```

workRow["Fax"] = "(503) 555-2376"; // 11

resultDataSet.Tables["Customers"].Rows.Add(workRow);

// Compatibilice los cambios con la fuente de datos
dsCmd.Update(resultDataSet, "Customers");

// Informe de los resultados
Console.WriteLine("Inserted 2 rows.");
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}

```

Operaciones que no devuelven datos jerárquicos

Un aspecto poco conocido de ADO es que es posible recuperar varios conjuntos de datos de una pasada. ADO.NET también dispone de esta característica. Observe los dos procedimientos almacenados consecutivos del procedimiento almacenado en el listado 23.20. (De nuevo, el resultado se limita a cinco registros para cada uno, para realizar la prueba).

Listado 23.20. Un procedimiento almacenado SQL Server con dos instrucciones Select

```

USE [Northwind]
GO
CREATE PROCEDURE [pc_getOrdersAndDetails]
AS
SELECT TOP 5
    OrderID,
    CustomerID,
    EmployeeID,
    OrderDate,
    RequiredDate,
    ShippedDate,
    ShipVia,
    Freight,
    ShipName,
    ShipAddress,
    ShipCity,

```

```

    ShipRegion,
    ShipPostalCode,
    ShipCountry
FROM Orders

SELECT TOP 5
    OrderID,
    ProductID,
    UnitPrice,
    Quantity,
    Discount
FROM [Order Details]
GO

```

Para recuperar estos datos, emplee un `DataReader` de la misma forma que hizo para recuperar conjuntos de filas. Sin embargo, si compara el código del listado 23.21 con el código del listado 23.17, comprobará que hay un bucle adicional alrededor de la iteración de filas. Este bucle adicional termina cuando `NextResults()` es `False`. Esto es lo que se debe saber para recuperar conjuntos de filas múltiples.

Listado 23.21. Cómo recuperar varios conjuntos de datos con `DataReader`

```

static void TestSelectHierWithDataReader()
{
    // Establezca las cadenas de instrucción SQL
    string strSQLSelect = "[pc_getOrdersAndDetails]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);

    // Estamos tratando con procedimientos almacenados (es decir,
NO con una instrucción SQL)
    selectCommand.CommandType = CommandType.StoredProcedure;

    try
    {
        // Establezca la conexión de la base de datos
        databaseConnection.Open();

        // Ejecute el comando SQL
        OleDbDataReader rowReader = selectCommand.ExecuteReader();

        // Informe de los resultados
        for (;;)
        {
            while (rowReader.Read())
            {
                string row = "";

```



```

        for (int i=0; i<= rowReader.FieldCount - 1; i++)
        {
            row = row + rowReader[i] + ", ";
        }
        Console.WriteLine("Row is {0}",
row.Substring(0,row.Length -2));
    }
    if(!rowReader.NextResult())
        break;
    else
        Console.WriteLine("Next Results:");
}
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}
}

```

En la última sección del código está uno de los puntos fuertes del objeto DataSet: la recuperación de datos relacionados. Como el DataSet fue diseñado para trabajar como una base de datos en memoria, tiene toda la funcionalidad necesaria para tratar con las relaciones primarias y secundarias. Los siguientes dos listados ejemplifican la recuperación de datos. El listado 23.22 muestra cómo se recuperan datos relacionados mediante una instrucción SQL y el listado 23.23 muestra cómo recuperar datos relacionados usando un objeto DataSet. El listado 23.23 demuestra cómo tratar con este tipo de relaciones. De nuevo, se usa un procedimiento almacenado que devuelve datos de dos instrucciones Select. Las instrucciones Select están relacionadas y se quiere conseguir el mismo resultado en ADO.NET como si se recuperasen los datos con una instrucción SQL igual a la que aparece en el listado 23.22.

Listado 23.22. Cómo recuperar datos con instrucciones SQL

```

SELECT
    Orders.OrderID,
    Orders.CustomerID,
    Orders.EmployeeID,
    Orders.OrderDate,
    Orders.RequiredDate,
    Orders.ShippedDate,
    Orders.ShipVia,
    Orders.Freight,
    Orders.ShipName,
    Orders.ShipAddress,

```

```

Orders.ShipCity,
Orders.ShipRegion,
Orders.ShipPostalCode,
Orders.ShipCountry,
[Order Details].ProductID,
[Order Details].UnitPrice,
[Order Details].Quantity,
[Order Details].Discount
FROM Orders
INNER JOIN [Order Details]
ON Orders.OrderID = [Order Details].OrderID

```

En la instrucción `try` del listado 23.23, se empieza asignando las tablas de datos a las tablas de origen usadas en la consulta SQL. El código que le sigue, completando el `DataSet`, es el código habitual incluido más abajo. A continuación viene la parte en la que se define la relación entre las dos tablas. Al establecer las relaciones entre las claves externas en un sistema de gestión de bases de datos relacionales (RDBMS) se necesitan las claves primarias y la base de datos en memoria no es diferente. La propiedad `PrimaryKey` recibe una matriz de objetos `DataColumn`. Tras establecer las claves primarias, se puede definir una relación. El primer parámetro es el nombre de la relación, que usará más tarde para recuperar los registros secundarios. A modo de demostración, el ejemplo recupera sólo la primera fila principal. A continuación recupera todas las filas secundarias asociadas mediante el método `GetChildRows()` usando el nombre de la relación. A continuación se puede usar un bucle que recorra la matriz de objetos `DataRow` para mostrar las filas secundarias.

Listado 23.23. Cómo recuperar datos relacionados con DataSet

```

static void TestSelectHierWithDataSet()
{
    // Establezca las cadenas de instruccion SQL
    string strSQLSelect = "[pc_getOrdersAndDetails]";

    // Cree objetos OleDb
    OleDbConnection databaseConnection = new
OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new OleDbCommand(strSQLSelect,
databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    DataSet resultDataSet = new DataSet();

    // Estamos tratando con procedimientos almacenados (es decir,
NO con una instruccion SQL)
    selectCommand.CommandType = CommandType.StoredProcedure;

    try
    {
        dsCmd.TableMappings.Add("Orders", "Orders");
        dsCmd.TableMappings.Add("Orders1", "Order Details");
    }
}

```

```

// Establezca la conexión de la base de datos
databaseConnection.Open();

// Ejecute el comando SQL
dsCmd.SelectCommand = selectCommand;
// Como no hay tablas en el DataSet antes de invocar el
método Fill,
// el OleDbDataAdapter creará automáticamente las tablas
para el DataSet
// y las completará con los datos devueltos. Si crea las
tablas antes de ejecutar
// el método FillDataSet, el OleDbDataAdapter simplemente
completará las tablas existentes.
int numRows = dsCmd.Fill(resultDataSet, "Orders");

// Reduzca el número de puntos evitando las referencias a
las tablas
DataTable orderTable = resultDataSet.Tables["Orders"];
DataTable detailsTable = resultDataSet.Tables["Order
Details"];

// Establezca la clave primaria de las tablas
orderTable.PrimaryKey =
    new DataColumn[] { orderTable.Columns["OrderID"] };
detailsTable.PrimaryKey = new DataColumn[] {
detailsTable.Columns["OrderID"],
detailsTable.Columns["ProductID"] };
// Establezca la relación de clave externa entre las tablas
resultDataSet.Relations.Add(new DataRelation("Order_Detail",
new DataColumn[] {
orderTable.Columns["OrderID"] },
    new DataColumn[] {
detailsTable.Columns["OrderID"]
    }));
// Informe de los resultados
// Muestre el pedido
DataRow orderRow = orderTable.Rows[0];
Console.WriteLine("Order ID is {0}, date is {1}, Ship To
is {2}.", orderRow["OrderID"],
orderRow["OrderDate"],
orderRow["ShipName"]);
// Recupere las filas secundarias para el pedido usando el
nombre de la relación
DataRow[] detailRows =
orderRow.GetChildRows("Order_Detail");
// Hace algo con la colección de filas secundarias
DataRow detailRow;
for (int i=0; i <= detailRows.Length - 1; i++)
{
    // Hace algo con la fila detail
    detailRow = detailRows[i];
    Console.WriteLine("Product ID is {0}, Quantity is
{1}.",

```

```

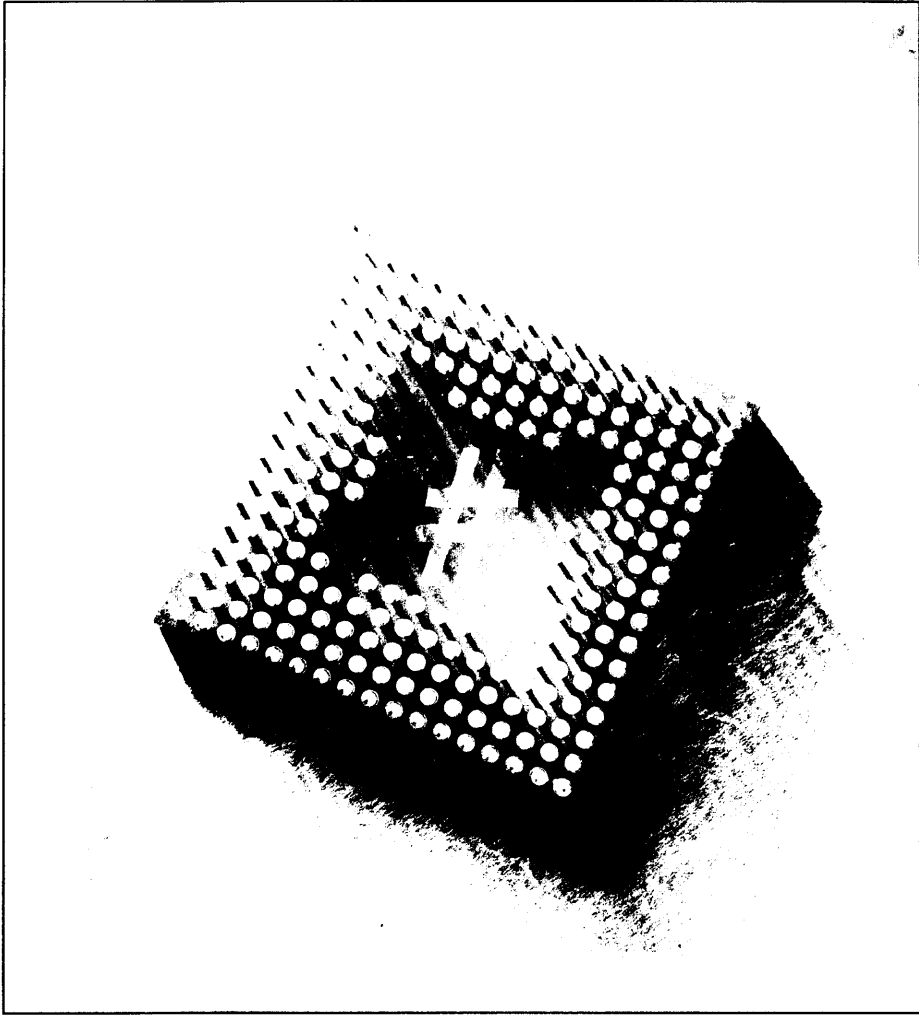
detailRow["ProductID"],
detailRow["Quantity"]);
    }
}
catch (Exception e)
{
    Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
}
finally
{
    databaseConnection.Close();
}
}
}

```

Resumen

Este capítulo describe todos los tipos de operaciones que puede realizar en ADO.NET. También se muestra lo sencillos y versátiles que son los objetos en el espacio de nombres `System.Data`, incluyendo el potente objeto `DataSet`, que funciona como una base de datos en memoria. También describe cómo devolver filas con entidades únicas y cómo borrar y actualizar operaciones.

Tenga en cuenta que la arquitectura de proveedor ADO.NET se refleja en las clases .NET Framework que admiten ADO.NET. Podemos usar las clases `Sql` para aprovechar el proveedor SQL Server de ADO.NET. Si su aplicación sólo va a admitir SQL Server, debe usar las clases `SQL`, porque el proveedor SQL Server para ADO.NET es más eficiente y funciona mejor que el proveedor OLE DB cuando se trata de SQL Server. Si su aplicación necesita incluir compatibilidad con otras bases de datos que no son SQL Server, es aconsejable elegir clases `OleDb`.



24

Cómo trabajar con archivos y con el registro de Windows

Las operaciones de archivo son algo a lo que todo programador debe enfrentarse en un momento u otro. La clase `System.IO` contiene una cantidad ingente de métodos para leer y escribir en y desde archivos. Esta clase simplifica la E/S de archivos y su manipulación y brinda un gran control de acceso a archivos. De forma parecida a la E/S del pasado, el acceso al registro de Windows siempre ha sido una tarea muy pesada. Esta tarea requería muchas llamadas al API que reducían el rendimiento de la aplicación y solían obligar al programador a escribir sus propias clases contenedoras para estas operaciones. Con .NET todo esto cambió. En este capítulo aprenderá a leer y escribir datos en y desde archivos. El capítulo estudia las operaciones de E/S que operan con texto normal y con datos binarios. También estudia algunas operaciones de archivos útiles, como mover, renombrar y eliminar archivos. Por último, aprenderá a supervisar el sistema de archivos para buscar cambios en archivos específicos y seguir recorriendo el registro de Windows.

Cómo acceder a archivos

El acceso a archivos en .NET suele hacerse con objetos de secuencia. Sin embargo, algunas clases confían en los objetos de secuencia para acceder a los

archivos. En este capítulo examinaremos dos de estas clases para mejorar su conocimiento de la E/S de archivos.

Acceso binario

Las clases `BinaryReader` y `BinaryWriter` son compatibles con el acceso binario a archivos. Estas clases permiten el acceso a archivos binarios y las operaciones binarias hacia y desde secuencias. Como se usan secuencias, las clases pueden ser muy flexibles y no tienen que tratar con detalles, como la posición de la secuencia o el acceso a la misma. El siguiente apartado examina la clase de acceso `BinaryWriter`.

BinaryWriter

La clase `BinaryWriter` permite escribir tipos de datos primitivos en secuencias y con el uso de las subclases, se pueden reemplazar los métodos de esta clase y cumplir con los requisitos de la codificación única de caracteres. Como esta clase usa una secuencia subyacente, hay que trabajar con muy pocos métodos y propiedades. La tabla 24.1 contiene las cinco propiedades y métodos básicos que más usará para trabajar con la clase `BinaryWriter`.

Tabla 24.1. Miembros básicos de la clase `BinaryWriter`

Nombre	Tipo	Función
<code>BaseStream</code>	Propiedad	Permite el acceso a la secuencia subyacente <code>BinaryWriter</code>
<code>Close</code>	Método	Cierra la clase <code>BinaryWriter</code> y la secuencia subyacente, eliminando todas las operaciones de escritura pendientes.
<code>Flush</code>	Método	Escribe todos los datos almacenados en el búfer en la secuencia subyacente y luego limpia los búferes.
<code>Seek</code>	Método	Establece la posición actual dentro de la secuencia en uso.
<code>Write</code>	Método	Este método sobrecargado escribe un valor en la secuencia en uso. Actualmente admite 18 variaciones de tipos de datos.

Para escribir datos en un archivo, antes hay que crear una secuencia de archivos. Seguidamente se puede instanciar una nueva clase `BinaryWriter`, pasándola a la secuencia. Tras crear esta clase `BinaryWriter`, sólo hay que llamar a su método `Write()` y pasarle los datos que se deben escribir, como se puede ver en el listado 24.1.

Listado 24.1. Cómo escribir datos en una secuencia de archivos mediante la clase `BinaryWriter`

```
static void Main(string[] args)
{
    FileStream myFStream = new FileStream ("c:\\\\TestFile.dat",
    FileMode.OpenOrCreate, FileAccess.ReadWrite);
    BinaryWriter binWrit = new BinaryWriter(myFStream);
    string testString = "This is a test string.";
    binWrit.Write(testString);
    binWrit.Close();
    myFStream.Close();
}
```

Al acabar con la clase `BinaryWriter`, hay que asegurarse de cerrarla y de cerrar la secuencia. Si no se cierra la clase `BinaryWriter` o la clase `FileStream`, se puede producir una pérdida de datos.

BinaryReader

La clase `BinaryReader`, al igual que la clase `BinaryWriter`, se basa en un objeto `FileStream` para acceder a los archivos.

Para comprender la clase `BinaryReader`, examine la aplicación del listado 24.2, que lee la información del título binario de un archivo de mapa de bits. A partir de la información de este título, se puede determinar el tamaño horizontal y vertical del archivo de imagen, además de su profundidad de color en bits.

Listado 24.2. Aplicación `BitmapSize`

```
using System;
using System.IO;

namespace BitmapSize
{
    class Class1
    {
        static void Main(string[] args)
        {
            long bmpWidth = 0;
            long bmpHeight = 0;
            int bmpPlanes = 0;
            int bmpBitCount = 0;
            string [] cma = Environment.GetCommandLineArgs();

            if (cma.GetUpperBound(0) >= 1)
            {
                FileStream myFStream = new
                FileStream(cma[1], FileMode.Open, FileAccess.Read);
                BinaryReader binRead = new BinaryReader(myFStream);
                binRead.BaseStream.Position=0x12;
                bmpWidth = binRead.ReadInt32();
            }
        }
    }
}
```



```

        bmpHeight= binRead.ReadInt32();
        bmpPlanes= binRead.ReadInt16();
        bmpBitCount = binRead.ReadInt16();

        Console.WriteLine(" [{0}] {1}x{2} {3}-
bit", cma[1], bmpWidth, bmpHeight, bmpBitCount);
        binRead.Close();
        myFStream.Close();
    }
}
}
}

```

Lo primero que se debe hacer en esta aplicación es declarar algunas variables para que contengan la información que se lee desde el archivo de mapa de bits y, a continuación, hay que almacenar todos los argumentos de línea de comandos en una matriz de cadenas.

En el anterior ejemplo, se almacenan los argumentos de línea de comandos en una matriz de cadenas, porque uno de estos argumentos determina qué archivo debe procesarse.

El elemento uno de la matriz de cadenas debe contener el nombre del archivo que debe procesarse. Tras determinar si existe un argumento de línea de comandos, se puede crear un objeto `FileStream` pasando el nombre del archivo y el modo que quiere usar para abrir el archivo, como se muestra a continuación:

```

FileStream myFStream = new FileStream( cma[1], FileMode.Open,
FileAccess.Read);

```

Al igual que la clase `BinaryWriter`, se crea el objeto `BinaryReader` y se pasa al objeto `FileStream` que se quiere usar. En este punto, ya está preparado para leer un archivo usando el modo binario. Tras examinar el diseño del archivo de mapa de bits, sabe que la información que quiere obtener empieza en la posición 18 (posición hexadecimal 12) del archivo. Al usar el objeto de la clase `BinaryReader BaseStream`, se puede acceder directamente al objeto `FileStream`. A partir de aquí, establece la propiedad `Position` del objeto a `0x12`, que busca en el archivo, a la posición desde la que se quiere leer.

Cuando el puntero del archivo está en posición, hay que leer dos valores `long` desde el archivo, seguidos por dos valores enteros. Un valor `long` necesita cuatro bytes, de modo que se usa el método `ReadInt32` dos veces para recuperar los valores.

A continuación, se usa el método `ReadInt16` para recuperar los dos archivos enteros del archivo. Observe en la tabla 24.2 la lista de los métodos más usados en la clase `BinaryReader`.

Una vez que se ha recuperado la información del archivo, sólo queda mostrar los valores almacenados en la consola. Tras compilar esta aplicación, vaya a la ventana de consola y pruébela con una imagen de mapa de bits, como muestra la figura 24.1.

Tabla 24.2. Métodos más usados dentro de la clase BinaryReader

Método	Descripción
Close	Cierra el objeto <code>BinaryReader</code> y la secuencia base
PeekChar	Lee el siguiente byte disponible de la secuencia pero no hace avanzar la posición del byte o del carácter en el archivo
ReadBoolean	Lee un valor booleano (<code>True/False</code>) de la secuencia
ReadByte	Lee un solo byte de la secuencia. También existe un método <code>ReadBytes</code> para especificar el número de bytes que se deben leer.
ReadChar	Lee un solo valor <code>Char</code> de la secuencia. También existe un método <code>ReadChars</code> para especificar el número de <code>Chars</code> que se deben leer.
ReadInt16	Lee un valor entero (2 bytes)
ReadInt32	Lee un valor long (4 bytes)
ReadInt64	Lee un entero de ocho bytes con signo

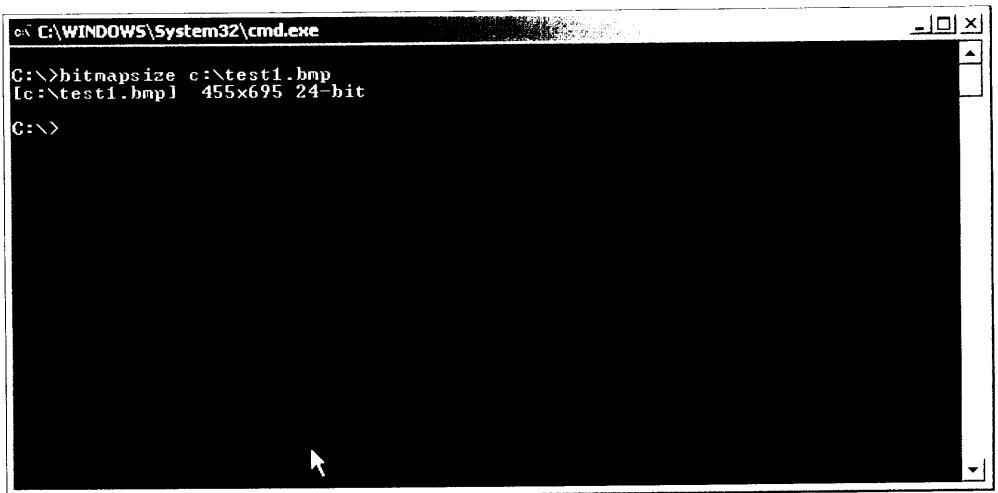


Figura 24.1. Al usar `BinaryReader`, puede observar el tamaño de la imagen de un archivo de mapa de bits

Cómo supervisar los cambios de archivo

La supervisión de los cambios de archivo con `C#` es cada vez más fácil gracias al uso del objeto `FileSystemWatcher`. Este objeto permite observar un

archivo concreto, un grupo de archivos, un directorio o toda una unidad para buscar varios eventos, incluyendo los cambios de archivo, eliminaciones de archivo, creaciones de archivo y cambios de nombre de archivos.

Se empieza usando el objeto `FileSystemWatcher` en uno de estos dos modos. Se puede crear un objeto `FileSystemWatcher` usando código y luego creando métodos para controlar los diferentes eventos, o se puede usar el método más sencillo. El cuadro de herramientas de la interfaz de desarrollo de Visual Studio .NET también contiene un objeto `FileSystemWatcher` que se puede insertar en el proyecto haciendo doble clic sobre ella.

Las aplicaciones, como Microsoft Word y Microsoft Excel, supervisan los archivos que se están usando por si se producen cambios en ellos desde fuera. A continuación aparecerá la opción de volver a abrir el archivo de modo que pueda observar todos los cambios que se han hecho en él. Para explorar las características del objeto `FileSystemWatcher` se construye una aplicación que supervisa todos los archivos de texto en la raíz de la unidad C: y muestra la información importante en la ventana de la aplicación.

Cómo usar la supervisión de archivos

Para empezar esta aplicación de ejemplo, abra una nueva aplicación de Windows C#. Tras abrir la ventana del nuevo proyecto, agregue dos botones y un cuadro de lista. Tras agregar los componentes a `Form1`, cambie los nombres de los botones a `btnStart` y `btnStop`. Cambie las propiedades `Text` de estos botones a **Start** y **Stop**, respectivamente. Ahora que los controles están en su sitio, colóquelos como se muestra en la figura 24.2.

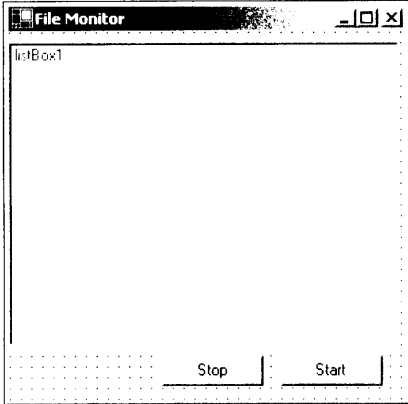


Figura 24.2. Arrastre y coloque controles en la interfaz File monitor como se muestra aquí

Debe agregar otro objeto más al proyecto: `FileSystemWatcher`. Seleccione la ficha **Componentes** del cuadro de herramientas, como muestra la figura 24.3. Haga doble clic en `FileSystemWatcher` para agregar una instancia de este objeto al proyecto.

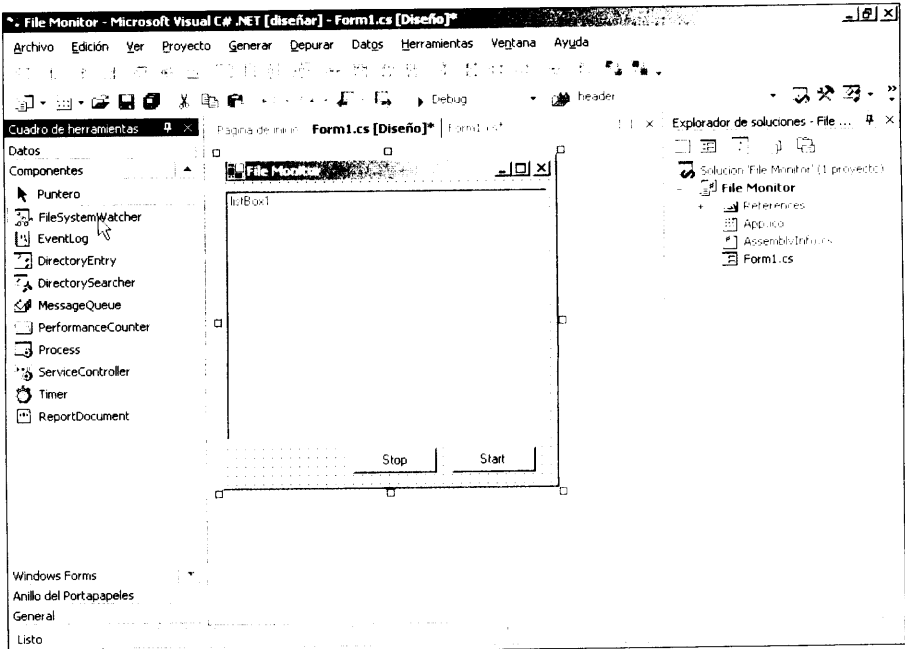


Figura 24.3. La ficha Componentes del cuadro de herramientas contiene el objeto `FileSystemWatcher`

Seleccione el componente `fileSystemWatcher1` situado inmediatamente debajo de `Form1` en el diseñador de formularios. La propiedad `EnableRaisingEvents` activa el objeto y le permite empezar a buscar eventos del sistema de archivos. Como tiene los botones **Start** y **Stop** para conseguirlo, debe asegurarse de que el objeto no se activa cuando se ejecuta la aplicación. Cambie la propiedad `EnableRaisingEvents` a `False`. La propiedad `Filter` permite asignar una máscara de archivo o un archivo que se debe observar. Como queremos ver todos los archivos, escriba `*.txt` en la propiedad `Filter`. Por último, necesita definir la carpeta en la que quiere supervisar archivos. Escriba `C:\` en la propiedad `Path` para asegurarse de que sólo examina los archivos que se encuentran en la raíz de la unidad C:. Se puede supervisar toda la unidad asignando a la propiedad `IncludeSubdirectories` el valor `True` pero es probable que esto reduzca el rendimiento del sistema.

Haga doble clic en el botón **Start** en `Form1` y añada el código del listado 24.3 al controlador de eventos `Click`.

Listado 24.3. Evento clic del botón Start

```
private void btnStart_Click(object sender, System.EventArgs e)
{
    fileSystemWatcher1.EnableRaisingEvents = true;
    btnStop.Enabled = true;
    btnStart.Enabled = false;
}
```

El anterior código asigna a la propiedad `EnableRaisingEvents` el valor `True`, lo que activa el objeto `FileSystemWatcher`. A continuación desactive el botón **Start** para que no se pueda volver a hacer clic en él y active el botón **Stop**.

Ahora debe agregar algo de código al botón **Stop** para desactivar `FileSystemWatcher`. Agregue el código del listado 24.4 al evento `Click` del botón **Stop**.

Listado 24.4. Evento clic del botón Stop

```
private void btnStop_Click(object sender, System.EventArgs e)
{
    fileSystemWatcher1.EnableRaisingEvents = false;
    btnStop.Enabled = false;
    btnStart.Enabled = true;
}
```

`FileSystemWatcher` ya está operativo, pero debe agregar los controladores de evento para capturar todos los eventos de archivo. Regrese al editor de formularios haciendo doble clic en `Form1` en el **Explorador de soluciones** y haciendo clic en el objeto `fileSystemWatcher1`. En la ventana **Propiedades**, haga clic en el icono **Eventos** de la barra de tareas. Al hacer clic en los eventos que aparecen en la ventana, aparecerá el editor de código para que pueda modificar esos controladores de eventos. El listado 24.5 contiene el código que debe introducir en los cuatro eventos.

Listado 24.5. Controladores de eventos `FileSystemWatcher`

```
private void fileSystemWatcher1_Deleted(object sender,
System.IO.FileSystemEventArgs e)
{
    listBox1.Items.Add( "[" + e.Name + "] Deleted" );
}
private void fileSystemWatcher1_Renamed(object sender,
System.IO.RenamedEventArgs e)
{
    listBox1.Items.Add( "[" + e.OldName + "] Renamed to " +
e.Name );
}
private void fileSystemWatcher1_Changed(object sender,
System.IO.FileSystemEventArgs e)
{
    listBox1.Items.Add( "[" + e.Name + "] Changed" );
}
private void fileSystemWatcher1_Created(object sender,
System.IO.FileSystemEventArgs e)
{
    listBox1.Items.Add( "[" + e.Name + "] Created");
}
```

Estos controladores de eventos son bastante simples: solamente muestran un mensaje en el cuadro de lista cuando se produce una operación. Antes de comprobar esta aplicación, es importante tener en cuenta algunas cosas. Como puede ver en cada una de estas funciones, los eventos `Changed`, `Created` y `Deleted` muestran los mismos datos; por tanto, tienen la misma firma. Lo extraño aquí es el evento `Renamed`. Como tres de los cuatro eventos pasan los mismos datos al procedimiento del evento, es posible usar un controlador de eventos que controle estos tres eventos, pero seguirá necesitando que un controlador distinto se haga cargo del evento `Renamed`.

Pulse **F5** para ejecutar el programa y empezar a examinarlo como se indica a continuación:

1. Cuando el programa se abra, haga clic en el botón **Start**. La aplicación está supervisando la raíz de la unidad `C:`, buscando cambios en los archivos de texto.
2. Abra el Explorador de Windows. Haga clic con el botón derecho en la ventana del explorador y seleccione **Nuevo>Documento de texto** para crear un nuevo documento.
3. En el nuevo documento, observará una entrada en el cuadro de lista de su aplicación File Monitor que indica que se ha creado un archivo. Abra este nuevo archivo de texto, agréguele algo de texto y guárdelo. De nuevo, verá entradas en la ventana de registro de la aplicación.
4. Ahora intente renombrar el archivo y borrarlo a continuación. Los resultados deberían ser similares a los que aparecen en la figura 24.4.



Figura 24.4. Su aplicación File monitor muestra la actividad del archivo en tiempo real

Debe ser consciente de que el evento `Changed` puede desencadenarse varias veces, porque `FileSystemWatcher` está examinando varias características del archivo. Si el tamaño del archivo cambia, se desencadena el evento

Changed. Si la fecha de modificación y la marca de hora del archivo cambian, el evento Changed se desencadena de nuevo y así sucesivamente. No se desanime si un evento se desencadena más de una vez. Por lo general no es un problema de la aplicación sino una garantía de que se ha establecido el NotifyFilters correcto.

Cómo codificar FileSystemWatcher

Puede agregar un FileSystemWatcher a su proyecto de C# sin usar el cuadro de herramientas de Visual Studio. Para conseguirlo, agregue una declaración al objeto situado inmediatamente debajo de la declaración de la clase, como muestra el listado 24.6.

Listado 24.6. Cómo agregar una instancia FileSystemWatcher a declaraciones de clase

```
public class Form1 : System.Windows.Forms.Form
{
    private FileSystemWatcher MyFileWatcher = new
    FileSystemWatcher();
```

Como está creando el control en tiempo de ejecución, no puede ir a la ventana **Propiedades** y asignar valores a las diferentes propiedades según sea necesario. En cambio, estas tareas se controlan desde dentro del evento Click del botón **Start**. Tras establecer las propiedades, como en el anterior ejemplo del listado 24.6, debe crear controladores de eventos y dirigirlos a las funciones adecuadas. El listado 24.7 muestra el listado actualizado del evento Click del botón **Start**.

Listado 24.7. Cómo crear declaraciones de controladores de eventos en el evento Click del botón **Start**

```
private void btnStart_Click(object sender, System.EventArgs e)
{
    MyFileWatcher.Path = "c:\\";
    MyFileWatcher.Filter = "*.txt";
    MyFileWatcher.IncludeSubdirectories = false;
    MyFileWatcher.EnableRaisingEvents = true;

    this.MyFileWatcher.Renamed += new
    System.IO.RenamedEventHandler( this.MyFileWatcher_Renamed );
    this.MyFileWatcher.Changed += new
    System.IO.FileSystemEventHandler( this.MyFileWatcher_Changed );
    btnStart.Enabled = false;
    btnStop.Enabled = true;
}
```

A continuación, agregue código al botón **Stop** y cree controladores de eventos para FileSystemWatcher. Los nombres de función para estos controladores de eventos deben coincidir con la declaración que colocó en el evento click de

los botones **Start**. El listado 24.8 contiene el evento `Stop Click` y los controladores de evento para los eventos `Changed` y `Renamed`.

Listado 24.8. Cómo crear controladores de evento para el objeto `FileSystemWatcher`

```
private void btnStop_Click(object sender, System.EventArgs e)
{
    MyFileWatcher.EnableRaisingEvents = false;
    btnStop.Enabled = false;
    btnStart.Enabled = true;
}

private void MyFileWatcher_Changed(object sender,
System.IO.FileSystemEventArgs e)
{
    listBox1.Items.Add ("[" + e.FullPath + "] Changed");
}

private void MyFileWatcher_Renamed(object sender,
System.IO.RenamedEventArgs e)
{
    listBox1.Items.Add ("[" + e.OldName + "] renamed to " +
e.Name);
}
```

Cómo manipular archivos

La manipulación de archivos es un acceso a archivos que sólo manipula el archivo y no los contenidos. La manipulación de archivos puede incluir la copia, borrado y traslado del archivo. Precisamente, .NET proporciona una clase llamada `FileInfo` incluida en el espacio de nombres `System.IO` para estas operaciones. Esta sección describe algunos de los métodos de manipulación y su modo de empleo.

Cómo copiar archivos

La clase `FileInfo` contiene, entre otras cosas, un método para copiar archivos. Este método, `CopyTo`, está sobrecargado y tiene dos variantes. La primera simplemente recibe el nombre del archivo de destino. Si ya existe un archivo con ese nombre, el método termina. La segunda variante recibe un nombre de archivo de destino y un valor booleano que indica si los archivos se pueden o no sobrescribir. Para mostrar el método `CopyTo`, esta sección le muestra cómo construir una aplicación de consola para copiar archivos. Como Windows ya tiene un comando de copia, asigne al nuevo método el nombre `cp`, que es el comando de copia para UNIX. El listado 24.9 muestra la implementación completa del comando `Copy` en C#. Cree una nueva aplicación de consola de C# llamada `cp` e introduzca el siguiente código.

Listado 24.9. Implementación C# del comando File Copy

```
using System;
using System.IO;
namespace cp
{
    class Class1
    {
        static void Main(string[] args)
        {
            string [] cla = Environment.GetCommandLineArgs();
            if (cla.GetUpperBound(0) == 2)
            {
                FileInfo fi = new FileInfo(cla[1]);
                fi.CopyTo(cla[2],true);
                Console.WriteLine("Copied " + fi.Length + " bytes.");
            }
            else
                Console.WriteLine ("Usage: cp <input file> <output
file>");
        }
    }
}
```

Este código usa el método `GetCommandLineArgs` de la clase `Environment` para recuperar los argumentos de línea de comandos que se pasan a la aplicación. Si el número de argumentos no es igual a 2, simplemente muestra un mensaje de uso, como se puede ver en la parte inferior del listado 24.9 y sale. En caso contrario, los argumentos de la línea de comandos se almacenan en la matriz de cadenas `cla` para ser usados más tarde.

Al usar la clase `FileInfo`, debe crear en primer lugar un objeto `FileInfo` y pasarle el nombre del archivo con el que está trabajando. Al usar esta aplicación, el argumento de la primera línea de comandos es el nombre del archivo fuente. Como esto se almacena en el elemento 1 de la matriz de cadenas, simplemente páselo al objeto `FileInfo`.

Para copiar un archivo usando esta clase, simplemente llame a su método `CopyTo` junto con el nombre del archivo de destino (que está en el elemento 2 de la matriz de cadenas) y un valor booleano que indique si se debe sobrescribir un archivo con el mismo nombre. Abra el intérprete de comandos para probar este programa después de compilarlo, como muestra la figura 24.5.

Como puede ver, agregó algo inesperado al programa. Una vez que el método `CopyTo` se haya completado, se muestra un mensaje en la consola que indica que la operación ha concluido y se muestra el número de bytes que se han copiado, gracias a la propiedad `Length` de la clase `FileInfo`.

Cómo eliminar archivos

El proceso de eliminación de archivos con la clase `FileInfo` resulta tan sencillo como llamar al método `Delete()`. Si el archivo que quiere eliminar

tiene un valor en su atributo `Read-Only`, obtendrá una excepción. El ejemplo del listado 24.10 crea una implementación de C# del comando `Delete` del archivo. Después de borrar el archivo, la utilidad muestra el nombre del archivo eliminado y sus atributos.

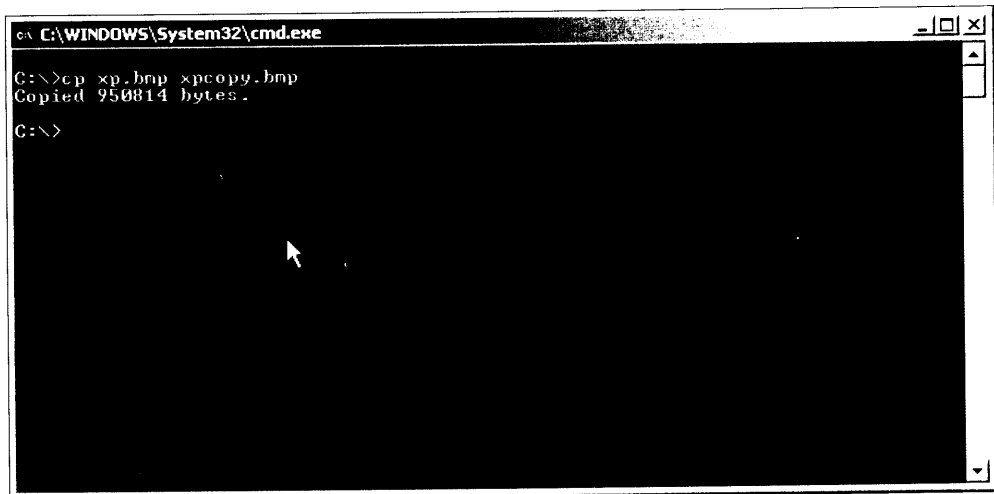


Figura 24.5. CopyTo permite copiar un archivo y mostrar información adicional

Cree un nuevo proyecto de aplicación de consola de C# llamado `rm` e introduzca el código del listado 24.10.

Listado 24.10. Uso de la clase `FileInfo` para eliminar archivos fácilmente

```
using System;
using System.IO;
namespace rm
{
    class Class1
    {
        static void Main(string[] args)
        {
            string [] cla = Environment.GetCommandLineArgs();
            if (cla.GetUpperBound(0) == 1)
            {
                FileInfo fi = new FileInfo(cla[1]);
                fi.Delete();
                Console.WriteLine("File      : " + cla[1]);
                Console.WriteLine("Attributes: " +
                    fi.Attributes.ToString());
                Console.WriteLine("File Deleted...");
            }
            else
                Console.WriteLine ("Usage: rm <filename>");
        }
    }
}
```

Como en los anteriores ejemplos, está almacenando los argumentos de línea de comandos dentro de una matriz de cadenas. Si esa matriz no contiene el número correcto de elementos, solamente se mostrará un mensaje de uso y finalizará.

TRUCO: Con el método `Delete()` de la clase `FileSystemInfo` puede eliminar directorios, además de archivos.

Tras invocar al método `Delete()` de la clase `FileInfo`, puede mostrar al usuario el nombre del archivo y sus atributos, indicando que ha sido eliminado. Mediante la propiedad `Attributes`, puede determinar de un modo seguro, antes de eliminar el archivo, si tiene asignado su atributo `Read-Only`. En ese caso, puede avisar al usuario y/o eliminar el atributo `Read-Only` usando la propiedad `Attributes` junto con el enumerador `FileAttributes`.

Una vez que su programa haya sido compilado, abra un intérprete de comandos y pruébelo. Simplemente escriba `rm` seguido del nombre del archivo que quiere eliminar. Los resultados deberían ser similares a los de la figura 24.6.

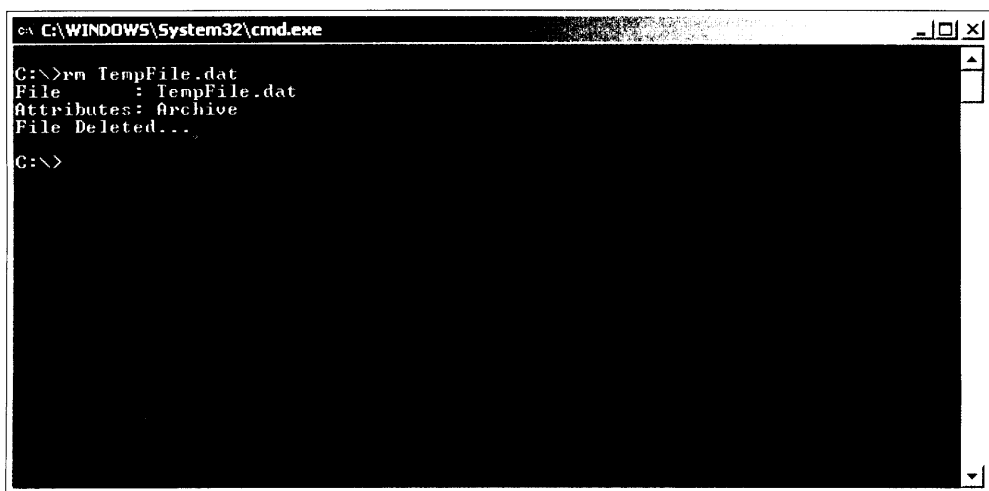


Figura 24.6. El método `Delete()` de la clase `FileInfo` muestra los atributos del archivo eliminado

Cómo trasladar archivos

El método `MoveTo()` de la clase `FileInfo` en realidad encapsula dos métodos diferentes: `CopyTo()` y `Delete()`. Después de copiar un archivo en el nombre de archivo o directorio adecuado, `MoveTo()` simplemente elimina el archivo de forma muy parecida a cómo lo hace el método `Delete()`.

El siguiente ejemplo de aplicación admite dos argumentos de línea de comandos: `Source Filename` y `Destination Filename`. Tras trasladar el archivo, el programa muestra cuándo se creó en realidad el archivo y a dónde se

trasladó. Ninguna de estas indicaciones tiene un uso práctico, excepto para mostrar cómo se pueden obtener algunos atributos, como la hora en que se creó el archivo, mediante la propiedad `CreationTime`.

Cree una nueva aplicación de consola C# y llame al proyecto `mv`, como el comando de UNIX. El listado 24.11 muestra por completo la aplicación.

Listado 24.11. Implementación de File Move

```
using System;
using System.IO;
namespace mv
{
    class Class1
    {
        static void Main(string[] args)
        {
            string [] cla = Environment.GetCommandLineArgs();
            if (cla.GetUpperBound(0) == 2)
            {
                FileInfo fi = new FileInfo(cla[1]);
                fi.MoveTo(cla[2]);
                Console.WriteLine("File Created : " +
                    fi.CreationTime.ToString());
                Console.WriteLine("Moved to      : " + cla[2]);
            }
            else
                Console.WriteLine ("Usage: mv <source file>
<destination file>");
        }
    }
}
```

La figura 24.7 muestra el resultado de la utilidad File Move.

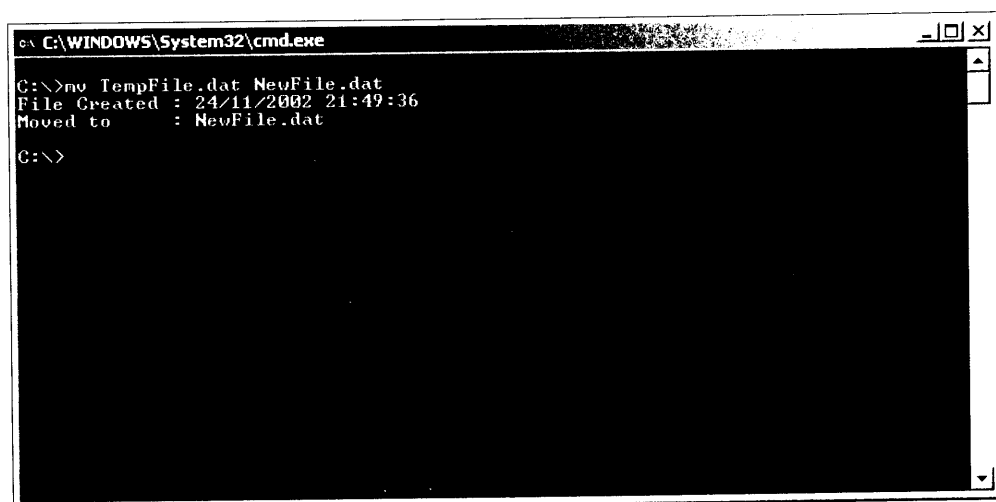


Figura 24.7. Mueva archivos con el método `MoveTo` de la clase `FileInfo`

Observe que en este ejemplo, el nombre del archivo de destino puede ser un nombre de archivo o un nombre de directorio. Si se especifica un nombre de directorio, el archivo se traslada. Si hay un nombre de archivo, el archivo es renombrado y/o trasladado. El método `MoveTo()` básicamente incorpora las funciones de copia y renombrado en un método.

Cómo acceder al registro

El acceso al Registro era una tarea bastante pesada en el API de Windows. C# proporciona algunos objetos de clase que permiten leer y escribir en y desde el registro fácilmente. El uso del registro tiene varias ventajas sobre los antiguos métodos, como archivos INI con formato texto. Como el registro está indizado, la búsqueda de claves se realiza rápidamente. El registro es un "documento" estructurado, lo que permite que la información estructurada, como una base de datos, sólo tenga que nombrar un tipo.

Cómo leer claves del registro

El acceso a la funcionalidad del registro se incluye en el espacio de nombres `Microsoft.Win32`, así que tenemos que incluir este espacio de nombres en todos los proyectos introduciendo la siguiente línea en la parte superior del archivo de código fuente:

```
using Microsoft.Win32;
```

Para leer una clave del registro use el objeto `RegistryKey`. Para empezar a estudiar este objeto, examine el listado 24.12, una aplicación que recupera dos fragmentos de información del registro.

Listado 24.12. Recupera el tipo de CPU y su velocidad del Registro

```
using System;
using Microsoft.Win32;

namespace CPUInfo
{
    class Class1
    {
        static void Main(string[] args)
        {
            RegistryKey RegKey = Registry.LocalMachine;
            RegKey = RegKey.OpenSubKey(
                "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0");
            Object cpuSpeed = RegKey.GetValue("~MHz");
            Object cpuType = RegKey.GetValue("VendorIdentifier");
            Console.WriteLine("You have a {0} running at {1}");
        }
    }
}
```

```

MHz.", cpuType, cpuSpeed);
    }
}
}

```

Al instanciar una instancia de `RegistryKey`, hace que su valor sea igual a un miembro de la clase `Registry`. El anterior ejemplo asigna al objeto `RegistryKey` el valor del campo `Registry.LocalMachine`, que permite el acceso a la clave base `HKEY_LOCAL_MACHINE`. La tabla 24.3 tiene una lista de todos los campos públicos de la clase de registro.

Tras establecer el objeto `RegistryKey`, invoque a su método `OpenSubKey()` y proporcione la clave que quiere abrir. En este caso concreto, debe dirigirse a la clave `HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\Central\Processor\0\` y leer dos valores de esa clave. Tenga en cuenta que debe incluir dos barras invertidas en la cadena para que no sean confundidas con un signo de escape.

Tras abrir la subclave, use las siguientes dos líneas de código para recuperar los valores `"~MHz"` y `"VendorIdentifier"` de esa subclave:

```

Object cpuSpeed = RegKey.GetValue("~MHz");
Object cpuType = RegKey.GetValue("VendorIdentifier");

```

Ahora tiene los valores almacenados en las variables adecuadas, de modo que puede mostrar la información en la ventana. Examine el programa desde una ventana de consola, como muestra la figura 24.8.

Tabla 24.3. Campos públicos de la clase `Registry`

Campo	Descripción
<code>ClassesRoot</code>	<code>ClassesRoot</code> define los tipos de documentos y las propiedades asociadas a esos tipos. Este campo empieza en la clave <code>HKEY_CLASSES_ROOT</code> del registro de Windows.
<code>CurrentConfig</code>	<code>CurrentConfig</code> contiene información relativa al hardware de su equipo. Este campo empieza en la clave <code>HKEY_CURRENT_CONFIG</code> del registro de Windows.
<code>CurrentUser</code>	Aquí se almacenan todas las preferencias del actual usuario. Este campo empieza en la clave <code>HKEY_CURRENT_USER</code> del registro de Windows.
<code>DynData</code>	<code>DynData</code> .
<code>LocalMachine</code>	<code>LocalMachine</code> contiene información para el equipo. Este campo empieza en la clave <code>HKEY_LOCAL_MACHINE</code> del registro de Windows.

Campo	Descripción
PerformanceData	La clave base almacena información relativa al rendimiento de los diferentes componentes de software. Este campo empieza en la clave <code>HKEY_PERFORMANCE_DATA</code> del Registro de Windows.
Users	Esta clave base contiene información para la configuración de usuario por defecto. Este campo empieza en la clave <code>HKEY_USERS</code> del registro de Windows.

```

C:\WINDOWS\System32\cmd.exe
C:\>cpuinfo
You have a AuthenticAMD running at 1396 MHz.
C:\>_

```

Figura 24.8. La clase `RegistryKey` simplifica la lectura de la importante información del registro

Si se está trabajando en equipos con varios procesadores puede obtener una lista de todos los procesadores enumerando la clave `CentralProcessor`. Hay una subclave en `CentralProcessor` para cada CPU del equipo.

Cómo escribir claves de registro

Crear y escribir claves en el Registro también se consigue usando el objeto `RegistryKey`. Varios métodos de la clase `RegistryKey` son útiles para escribir claves. La tabla 24.4 describe la función de algunos de los miembros más importantes.

ADVERTENCIA: Escribir valores puede ser peligroso y puede hacer que su sistema no responda si no se tiene suficiente cuidado. Compruebe dos veces todo el código antes de probar cualquier aplicación que escriba valores en el Registro.

Tabla 24.4. Miembros comunes de RegistryKey

Nombre	Tipo	Descripción
SubKeyCount	Propiedad	Esta propiedad recupera un recuento de las subclaves de la clave actual.
ValueCount	Propiedad	Esta propiedad recupera un recuento del número de valores de la clave actual.
Close	Método	Este método cierra la clave actual. Si se han realizado cambios en la clave, los cambios se guardan en el disco.
CreateSubKey	Método	Este método crea una nueva subclave o abre la subclave si ya existe.
DeleteSubKey	Método	Este método elimina una subclave. Este método está sobrecargado y contiene un parámetro booleano que permite que se inicie una excepción si no se puede encontrar la clave.
DeleteSubKeyTree	Método	Este método elimina una clave y todas las subclaves secundarias de manera recurrente.
DeleteValue	Método	Este método elimina un valor de una clave. Este método está sobrecargado y contiene un parámetro booleano que permite que se inicie una excepción si no se puede encontrar el valor.
GetSubKeyNames	Método	Este método devuelve una matriz de cadenas que contiene todos los nombres de las subclaves.
GetValue	Método	Este método devuelve un valor para una clave específica. Este método está sobrecargado y contiene un parámetro que admite un valor por defecto. Si no se puede encontrar el valor para la clave, se devuelve el valor por defecto que se especificó.
GetValueNames	Método	Este método devuelve una matriz de cadenas que contiene todos los valores para la clave especificada.
OpenSubKey	Método	Este método abre una subclave para un proceso (acceso de lectura y escritura).
SetValue	Método	Este método asigna un valor a una clave. Para asignar el valor por defecto a una clave, asigne el parámetro <code>subKey</code> a una cadena vacía.

El listado 24.13 muestra una sencilla aplicación que escribe dos valores en el Registro y a continuación lee esos valores para mostrarlos.

Listado 24.13. Cómo escribir texto y un valor DWord en el Registro

```
using System;
using Microsoft.Win32;

namespace WriteRegValues
{
    class Class1
    {
        static void Main(string[] args)
        {
            RegistryKey RegKeyWrite = Registry.CurrentUser;
            RegKeyWrite = RegKeyWrite.CreateSubKey
("Software\\CSHARP\\WriteRegistryValue");
            RegKeyWrite.SetValue("Success", "TRUE");
            RegKeyWrite.SetValue("AttemptNumber", 1);
            RegKeyWrite.Close();

            RegistryKey RegKeyRead = Registry.CurrentUser;
            RegKeyRead = RegKeyRead.OpenSubKey
("Software\\CSHARP\\WriteRegistryValue");
            Object regSuccessful = RegKeyRead.GetValue("Success");
            Object regAttemptNumber =
RegKeyRead.GetValue("AttemptNumber");
            RegKeyRead.Close();

            if ((string)regSuccessful == "TRUE")
                Console.WriteLine("Succeeded on attempt #
{0}", regAttemptNumber);
            else
                Console.WriteLine("Failed!");
        }
    }
}
```

Tras crear un objeto `RegistryKey`, puede crear una nueva subclave con el método `CreateSubKey()`. Asegúrese de emplear dos barras invertidas al usar este método para que el compilador no confunda los caracteres con una secuencia de escape. En este ejemplo, se crea una nueva clave bajo `HKEY_CURRENT_USER`. Almacene los valores en la subclave `\\Software\\CSHARP\\WriteRegistryValue`. Una vez que la nueva clave esté en su sitio, use el método `SetValue()` para especificar el nombre del valor y el valor actual. Este ejemplo almacena texto en el valor `Success` y un `DWord` en el valor `AttemptNumber`. Una vez asignados los valores, es aconsejable cerrar la clave por si se produce un corte de luz o algún fallo similar. En este punto, se han producido cambios en el registro. Si abre la aplicación `RegEdit` y se dirige a la clave adecuada, debería ver los valores mostrados en la figura 24.9.

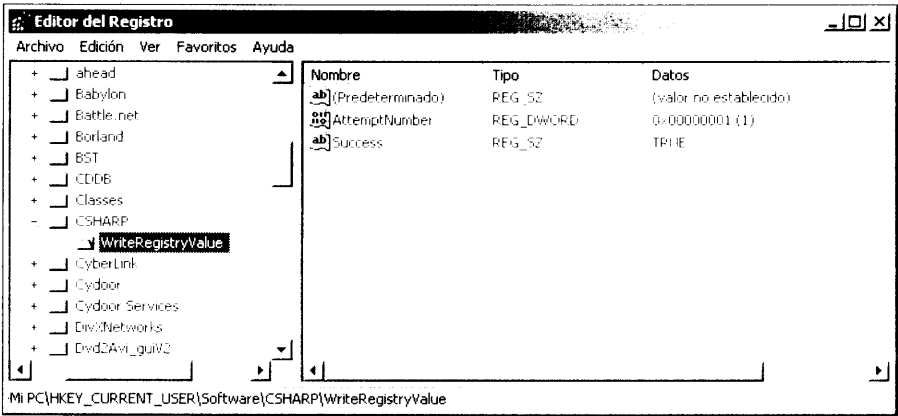


Figura 24.9. RegEdit indica que se han guardado sus valores

Como en el ejemplo anterior, crea un nuevo objeto `RegistryKey` y lee de nuevo los valores. Si el valor de `Success` es realmente `True`, se muestra la información en la pantalla, como se muestra en la figura 24.10.



Figura 24.10. Las claves que se leen de la consola se muestran en la consola.

Esta aplicación muestra una sencilla técnica para escribir valores en el registro. Este método ha demostrado ser útil para controlar la configuración de los programas, guardar la última posición y tamaño de la interfaz de las aplicaciones y acciones similares. Las posibilidades son ilimitadas.

Cómo enumerar claves del registro

La enumeración de claves de Registro es muy parecida a la utilidad `Buscar archivo` de Windows: permite buscar en el registro desde cualquiera de sus puntos

y recuperar todas las subclaves y valores debajo de ese punto inicial. Actualmente no hay ningún método en .NET para enumerar las claves de registro. Deberá crear sus propias funciones si las necesita. El conocimiento de la estructura de las claves que se desean enumerar hace que la tarea resulte mucho más sencillo, ya que puede usar un simple bucle. Si no se conoce la estructura de las entradas del registro, necesitará crear una función que pueda llamar y pasar la clave inicial cada vez que es invocada.

El listado 24.14 es un ejemplo de enumeración de claves de registro. Este ejemplo inspecciona el registro en busca de una lista de todo el software que tiene instalado en su equipo. Este programa enumera cualquier aplicación que aparece en la sección **Agregar o quitar programas** del Panel de control.

Listado 24.14. Cómo enumerar claves de Registro

```
using System;
using Microsoft.Win32;

namespace Installed
{
    class Class1
    {
        static void Main(string[] args)
        {
            RegistryKey myRegKey=Registry.LocalMachine;
            myRegKey=myRegKey.OpenSubKey
("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall");
            String [] subkeyNames = myRegKey.GetSubKeyNames();
            foreach (String s in subkeyNames)
            {
                RegistryKey UninstallKey=Registry.LocalMachine;
                UninstallKey=UninstallKey.OpenSubKey
("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\" +
s);
                try
                {
                    Object
oValue=UninstallKey.GetValue("DisplayName");
                    Console.WriteLine(oValue.ToString());
                }
                catch (NullReferenceException)
                {
                }
            }
        }
    }
}
```

Tras crear un objeto `RegistryKey`, abra la subclave `Uninstall`, que contiene una lista de todos los programas instalados. A partir de aquí, use `GetSubKeyNames`, que devuelve una matriz de cadenas de todas las subclaves.

Ahora que tiene una lista de subclaves, use el operador `foreach` para recorrer todos los elementos de la matriz de cadenas de subclave.

Al recorrer cada clave, se busca un valor llamado `DisplayName`. Este valor es el nombre que se muestra en la sección **Agregar o quitar programas** del Panel de control. Recuerde que no todas las claves tendrán este valor. Por tanto, debe encapsular el método `GetValue` con una instrucción `try...catch` para capturar todas las posibles excepciones. Cuando se encuentra un valor `DisplayName`, se recupera el valor y se muestra en la ventana. A continuación, la instrucción `foreach` pasa a la siguiente clave de registro de la matriz de cadenas.

Pulse **F5** para probar la aplicación. Probablemente verá una larga lista de aplicaciones desplazándose a medida que el programa inspecciona el registro (véase la figura 24.11).

Una cosa que no se intenta en este programa es ordenar las aplicaciones alfabéticamente. Los elementos del registro no se almacenan así, pero para solucionar esto, puede simplemente almacenar los resultados en una matriz de cadenas y llamar al método `Sort` para ordenar el resultado de la manera deseada.

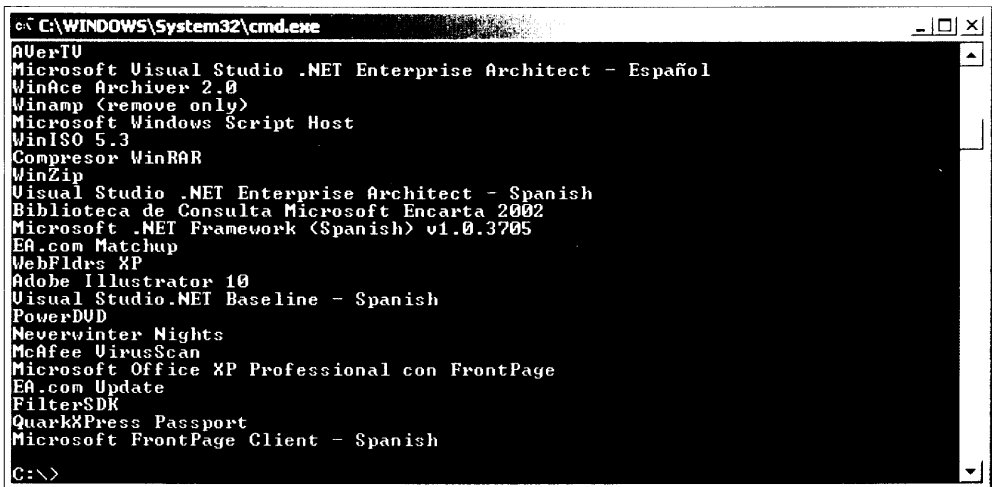
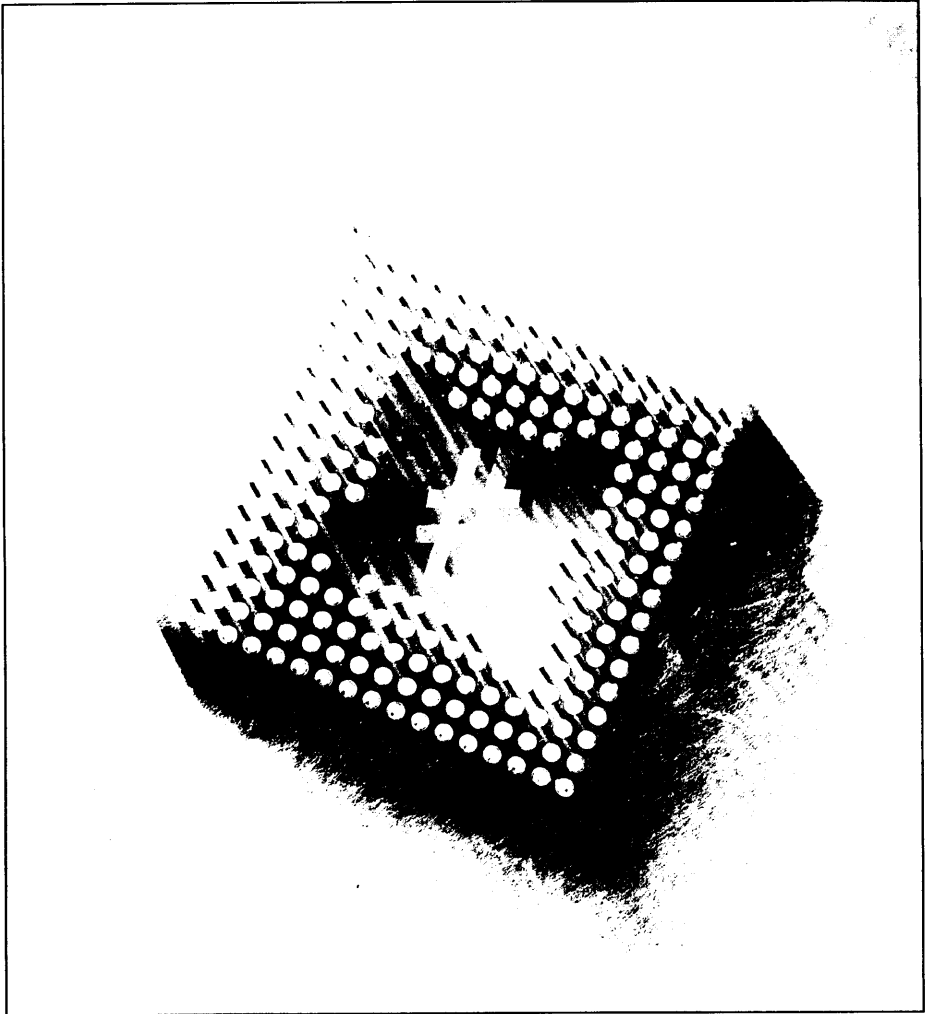


Figura 24.11. Cómo inspeccionar todas las aplicaciones instaladas con un enumerador de Registro

Resumen

.NET Framework ha reducido enormemente la cantidad de código y tiempo necesario para tratar efectivamente con archivos y con el registro de Windows. Entre los muchos beneficios de .NET Framework, ahora tiene acceso a componentes como `FileSystemWatcher` que permite examinar un sistema de archivos para buscar cambios realizados en cualquier archivo. Sin embargo, debe tener

cuidado al escribir aplicaciones que traten con el registro de Windows. porque si elimina por error claves de registro puede hacer que su sistema se vuelva inestable o incluso que deje de funcionar.



25 Cómo acceder a secuencias de datos

.NET Framework tiene clases que proporcionan un gran nivel de compatibilidad para la lectura y escritura de datos. Tradicionalmente, los lenguajes han proporcionado compatibilidad integrada para la lectura y escritura de archivos de disco y han confiado en las interfaces de programación de los sistemas operativos para la lectura y escritura de otros tipos de secuencias de datos, como conexiones de red o archivos de memoria. .NET Framework unifica la E/S de datos al proporcionar un conjunto de clases común que admite lecturas y escrituras de datos sin importar el sistema de almacenamiento subyacente usado para proporcionar el acceso a los datos. Todas estas clase pueden usarse desde el código C#.

En este capítulo aprenderá a usar secuencias. Aprenderá a usar lectores y escritores para leer y escribir datos en una secuencia y a realizar operaciones de archivo en segundo plano.

Jerarquía de clases de E/S de datos

La figura 25.1 muestra la jerarquía de clases para las clases básicas .NET Framework que se emplean al trabajar con E/S de datos. Las clases se agrupan en una de estas tres categorías: secuencias, escritores y lectores.

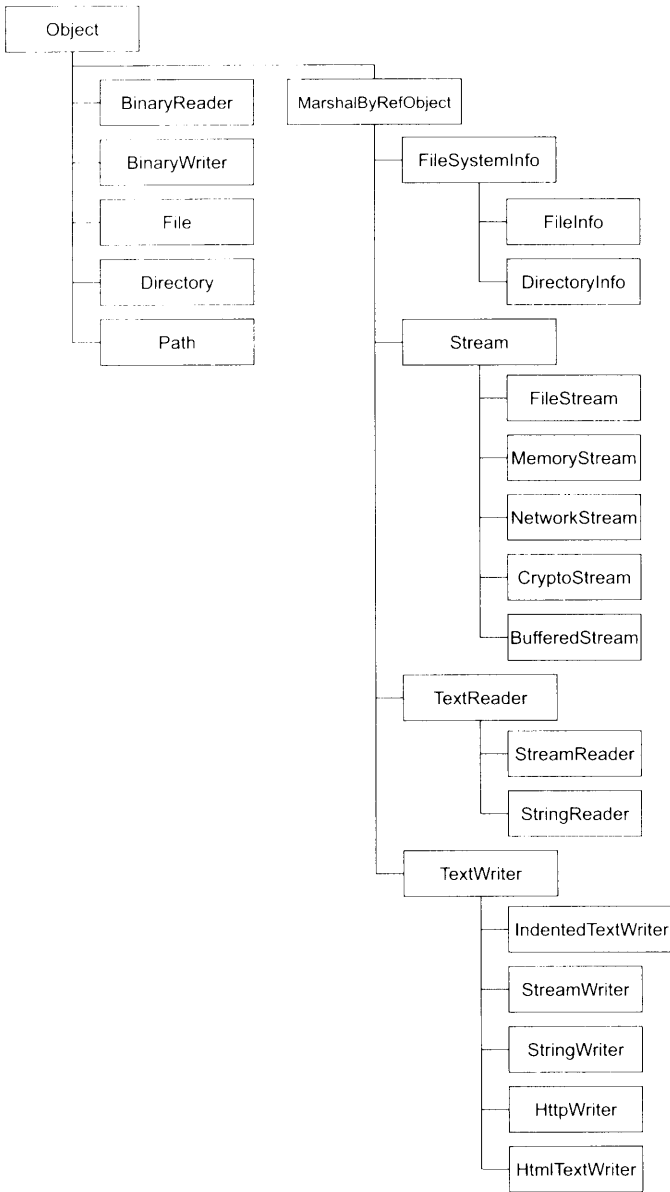


Figura 25.1. Jerarquía de clases de E/S de datos

Cómo usar secuencias

Las clases de secuencia proporcionan un mecanismo para hacer referencia a un contenedor de datos. Las clases de secuencia comparten una clase base común llamada `Stream`, que está definida en un espacio de nombres .NET Framework llamado `System.IO`.

La clase base `Stream` contiene propiedades y métodos que permiten a los invocadores trabajar con la secuencia de datos. .NET Framework dispone de varias clases que se derivan de la clase base `Stream`. Cada clase proporciona una implementación específica de una secuencia de datos usada para un entorno particular.

La clase `FileStream`, por ejemplo, proporciona una implementación que permite a los invocadores trabajar con secuencias de datos vinculadas a un archivo de disco.

Del mismo modo, la clase `NetworkStream` proporciona una implementación que permite a los invocadores trabajar con secuencias de datos a las que se accede mediante una red de comunicaciones.

Cómo usar escritores

Las secuencias admiten el acceso de datos en el nivel de byte. Incluyen métodos llamados `Read()` y `Write()`, que trabajan con una matriz de bytes que se procesan durante la llamada. Sin embargo, trabajar a nivel de byte puede no ser lo mejor para una aplicación. Suponga, por ejemplo, que una aplicación debe escribir una serie de números enteros en una secuencia. Como los números enteros en la implementación de 32 bits tienen un tamaño de cuatro bytes, el código C# necesitará convertir cada número entero en una cadena de cuatro bytes que pueda ser usada en una llamada a la implementación de `Write()` de la secuencia. .NET Framework incluye clases de escritor que permiten escribir varios tipos de datos de nivel superior en una secuencia.

Un escritor puede admitir muchas sobrecargas de un método `Write()`. Por ejemplo, un escritor puede aceptar datos como `int`, `long` o `double`. Las implementaciones de la clase `writer` convierten los tipos de datos en una serie de bytes y pasan esa secuencia de bytes convertidos a un objeto `Stream`. Este diseño de clase libera al código de tener que trabajar con secuencias de nivel de byte. El código de la aplicación de C# puede simplemente indicar, por ejemplo, "escribe este dato `long` sin signo en la secuencia" y permitir que la clase `writer` realice el trabajo necesario para obtener el valor almacenado en la secuencia como una serie de bytes.

Cómo usar lectores

Las clases de lectores complementan a las clase de escritores. Como las clases de escritores, las clases de lectores admiten la lectura de tipos de datos que va más allá de la simple matriz de bytes admitida por las clases de secuencia. En .NET Framework, hay una clase de lector para complementar a cada clase de escritor. Las clases de lector proporcionan varias sobrecargas de un método `Read()` que permite que el código de una aplicación lea varios tipos de datos, como cadenas, enteros, `long`, etc.

Cómo trabajar con secuencias

Las secuencias admiten dos métodos de E/S:

- E/S sincrónica. en la que las llamadas que realizan la secuencia de E/S no regresan al invocador hasta que la operación de E/S solicitada se ha completado.
- E/S asincrónica. en la que las llamadas que realizan la secuencia de E/S regresan al invocador antes de que la operación de E/S solicitada se haya completado y, posteriormente, informa al invocador de la conclusión de la operación.

E/S sincrónica

El listado 25.1 muestra una E/S de secuencia sincrónica. Crea un archivo y escribe 256 bytes de datos binarios en él. A continuación, lee los 256 bytes del archivo y se asegura de que la lectura de datos coincida con los datos escritos.

Listado 25.1. E/S de archivos sincrónica

```
using System;
using System.IO;

class FileTestClass
{
    private FileStream BinaryFile;
    private byte [] ByteArray;

    public FileTestClass()
    {
        BinaryFile = new FileStream("test.dat", FileMode.Create,
        FileAccess.ReadWrite);
        ByteArray = new byte [256];
    }

    public void WriteBytes()
    {
        int ArrayIndex;

        for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
            ByteArray[ArrayIndex] = (byte)ArrayIndex;
        BinaryFile.Write(ByteArray, 0, 256);
    }

    public bool ReadBytes()
    {
        int ArrayIndex;
```

```

        BinaryFile.Seek(0, SeekOrigin.Begin);
        BinaryFile.Read(ByteArray, 0, 256);
        for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
        {
            if(ByteArray[ArrayIndex] != (byte)ArrayIndex)
                return false;
        }
        return true;
    }
}

class MainClass
{
    static public void Main()
    {
        FileTestClass FileTest = new FileTestClass();
        bool ReadTest;

        FileTest.WriteBytes();
        ReadTest = FileTest.ReadBytes();
        if(ReadTest == true)
            Console.WriteLine("The readback test was successful.");
        else
            Console.WriteLine("The readback test failed.");
    }
}

```

El listado 25.1 implementa dos clases C#: `FileTestClass`, que contiene el código de E/S de la secuencia y `MainClass`, que contiene el método `Main()` de la aplicación. El método `Main()` crea un objeto de la clase `FileTestClass` y pide al objeto que escriba y lea datos.

La clase `FileTestClass` contiene un miembro privado que representa un objeto `FileStream`. El constructor de la clase crea un nuevo objeto `FileStream` mediante un constructor que acepta tres argumentos:

- La ruta de la secuencia de archivo con la que se va a trabajar.
- Una especificación de modo para la operación de archivos.
- Una especificación de modo para el acceso a archivos.

La especificación de modo para la operación de archivos está representada por una enumeración llamada `FileMode`. La enumeración `FileMode` se incluye en el espacio de nombres `System.IO` de .NET y admite los siguientes miembros de enumeración:

- `Append`, que ordena a la secuencia de archivos que abra el archivo indicado si lo encuentra. Si el archivo indicado existe, la clase de secuencia de archivos se inicializa para escribir datos al final del archivo. Si el archivo indicado no existe, la clase crea un nuevo archivo con el nombre especificado.

- `Create`, que ordena a la secuencia de archivos que cree el archivo indicado. Si el archivo ya existe, se sobrescribe.
- `CreateNew`, que, como `Create`, ordena a la secuencia de archivos que cree el archivo indicado. La diferencia entre `CreateNew` y `Create` es el modo de trabajar con los archivos existentes. Si el archivo ya existe cuando se especifica `CreateNew` como modo de archivos, la secuencia de archivos inicia una excepción de la clase `IOException`.
- `Open`, que ordena a la secuencia de archivos que abra el archivo indicado.
- `OpenOrCreate`, que ordena a la secuencia de archivos que cree el archivo indicado. Si el archivo ya existe, el objeto `FileStream` abre el archivo indicado.
- `Truncate`, que ordena a la secuencia de archivos que abra el archivo indicado y, a continuación, lo trunca para que su tamaño sea cero bytes.

La especificación del modo de acceso a archivos está representada por una enumeración llamada `FileAccess`. La enumeración `FileAccess` también se encuentra en el espacio de nombres `.NET System.IO` de `.NET` y admite los siguientes miembros de enumeración:

- `Read`, que especifica que la clase `FileStream` debe permitir el acceso de lectura al archivo especificado.
- `ReadWrite`, que especifica que la clase `FileStream` debe permitir el acceso de lectura y escritura al archivo especificado.
- `Write`, que especifica que la clase `FileStream` debe permitir el acceso de escritura al archivo especificado. Se pueden escribir datos en el archivo, pero no se pueden leer.

El constructor `FileTestClass` del listado 25.1 crea una nueva secuencia de archivos que gestiona un archivo llamado `test.dat`. El archivo se abre en modo de creación para el acceso de lectura y escritura.

El método `WriteBytes()` de `FileTestClass` se coloca en un búfer de 256 de bytes, que crea el constructor de la clase. Se ubica en el búfer de 256 bytes con valores que van desde 00 hexadecimal a FF hexadecimal. A continuación se escribe el búfer en la secuencia mediante el método de secuencia de archivos `Write()`. El método `Write()` acepta tres argumentos:

- Una referencia al búfer de bytes que contiene los datos que se van a escribir.
- Un número entero que especifica el elemento de matriz del primer byte del búfer que se va a escribir.
- Un número entero que especifica el número de bytes que se van a escribir.

El método `Write()` es sincrónico y no regresa hasta que los datos se han escrito realmente en la secuencia.

El método `ReadBytes()` de `FileTestClass` lee los 256 bytes escritos por `WriteBytes()` y compara los bytes con el patrón de bytes implementado por `WriteBytes()`.

La primera operación que realiza el método `ReadBytes()` supone trasladar el puntero de la secuencia al principio del archivo. La posición de las secuencias es un concepto importante y merece una especial atención. Las secuencias admiten el concepto de posicionamiento de fila. Una posición de secuencia hace referencia a una posición en la secuencia donde tendrá lugar la próxima operación de E/S. Por lo general, una posición de secuencia se sitúa al principio de una secuencia cuando se inicializa esa secuencia. A medida que se leen o escriben datos en la secuencia, la posición de la secuencia avanza hasta la posición inmediatamente después de la última operación.

La figura 25.2 refleja este concepto. Muestra una secuencia con seis bytes de datos.

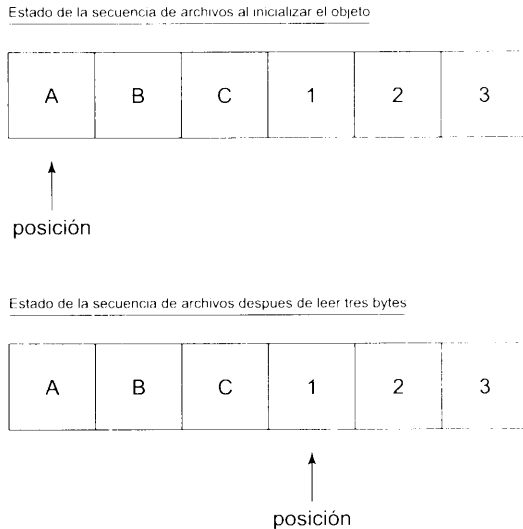


Figura 25.2. La secuencia de E/S se mueve a la siguiente posición de secuencia

Cuando la secuencia se abre por primera vez, la posición de secuencia apunta al primer byte de la secuencia. Esto aparece reflejado en el diagrama superior de la figura 25.2. Imagine que el código que gestiona la secuencia lee tres bytes del archivo. Se leen los tres bytes y la posición de secuencia apuntará al byte inmediatamente después de la última posición que se ha leído. Siguiendo con el ejemplo, la posición de secuencia apuntará al cuarto byte de la secuencia. Esto se refleja en la parte inferior del diagrama de la figura 25.2.

La relación con el código 25.1 tiene que ver con el hecho de que se crea un archivo y se escriben 256 bytes en él. Cuando se han escrito los bytes, se leen. Sin

embargo, es importante recordar dos conceptos de posicionamiento de secuencias:

- La posición del archivo se actualiza después de cada operación de lectura o escritura, para señalar una posición inmediatamente posterior a la última operación.
- Las operaciones de lectura y escritura empiezan en el byte al que hace referencia la posición de secuencia.

Cuando el código del listado 25.1 crea la nueva secuencia de archivos, la posición de archivo es situada al principio del archivo (vacío). Tras escribir los 256 bytes, la posición del archivo se actualiza para hacer referencia a la posición inmediatamente posterior a los 256 bytes. Si el código va a leer los 256 bytes inmediatamente después de la operación de escritura, la operación de lectura fallará porque la posición de secuencia apunta hacia el final del archivo tras la operación de escritura y la operación de lectura intentará leer 256 bytes a partir de esa posición, pero no hay bytes disponibles en esa posición. El código debe decir "antes de que empiece la operación de lectura, vuelve a apuntar el puntero del archivo hacia el principio de la secuencia para que la operación de lectura pueda realizarse."

Esta tarea la realiza un método de la clase `Stream` llamada `Seek()`. El método `Seek()` permite al código trasladar o buscar la posición de secuencia a cualquier posición disponible en la secuencia. El método `Seek()` recibe dos parámetros:

- Un número entero `long`, que especifica un desplazamiento de posición en bytes.
- Un valor de una enumeración llamada `SeekOrigin`, que especifica la posición de secuencia que debe usarse en el punto inicial para la operación de búsqueda.

La enumeración `SeekOrigin` se declara en el espacio de nombre `System.IO` y admite los siguientes valores:

- `Begin`, que indica que la operación de búsqueda debe realizarse respecto al principio de la secuencia.
- `Current`, que indica que la operación de búsqueda debe realizarse respecto a la actual posición de la secuencia.
- `End`, que indica que la operación de búsqueda debe realizarse respecto al final de la secuencia.

El método `Seek()` ajusta la posición de la secuencia para que apunte a la posición de secuencia a la que hace referencia la enumeración `SeekOrigin`, que se desplaza el número especificado de bytes. El desplazamiento de bytes

usado en el método `Seek()` puede ser positivo o negativo. El siguiente ejemplo usa un valor de desplazamiento positivo:

```
File.Seek(4, SeekOrigin.Begin);
```

La anterior línea ajusta el puntero de la secuencia para que apunte cuatro bytes más allá del principio de la secuencia. Los valores de desplazamiento positivo desplazan el puntero de la secuencia hacia el final de la secuencia. El siguiente ejemplo usa un valor de desplazamiento negativo:

```
File.Seek(-2, SeekOrigin.End);
```

Este ejemplo ajusta el puntero de la secuencia para que apunte dos bytes antes del final de la secuencia. Los valores de desplazamiento negativos desplazan el puntero de la secuencia hacia el principio de la secuencia.

El código del listado 25.1 usa el siguiente código de búsqueda antes de que se lean los 256 bytes:

```
BinaryFile.Seek(0, SeekOrigin.Begin);
```

Esta llamada ajusta el puntero hacia el principio de la secuencia. Cuando empieza la operación de lectura, comienza leyendo desde el principio de la secuencia.

El método `ReadBytes()` usa el método `FileStream` llamado `Read()` para realizar una lectura sincrónica de E/S en la secuencia. El método `Read()` admite tres argumentos:

- Una referencia al búfer de bytes que se usará para contener la lectura de bytes de la secuencia.
- Un número entero que especifica el elemento de matriz del primer byte en el búfer que contendrá la lectura de datos de la secuencia.
- Un número entero que especifica el número de bytes que se van a leer.

El método `Read()` es sincrónico y no regresa hasta que se han leído realmente los datos de la secuencia. Cuando la operación de lectura se completa, el código comprueba el patrón de bytes de la matriz para asegurarse de que concuerda con el patrón de bytes que se escribió.

E/S asincrónica

El listado 25.2 es una modificación del listado 25.1 que refleja la E/S asincrónica. A diferencia de la E/S sincrónica, en la que las llamadas para leer y escribir operaciones no regresan hasta que la operación está completa, las llamadas a operaciones de E/S asincrónicas regresan poco después de ser llamadas. La operación de E/S actual se realiza de forma oculta, en un subproceso distinto creado por la implementación de los métodos de E/S asincrónicos de .NET

Framework y cuando la operación se ha completado se advierte al código mediante un delegado. La ventaja de la E/S asincrónica es que el código principal no necesita depender de que se complete una operación de E/S. La realización de largas operaciones de E/S en segundo plano evita que la aplicación tenga que realizar otras tareas, como procesar los mensajes de Windows en aplicaciones de Windows Forms.

Cómo leer de forma asincrónica

El listado 25.2 mejora el listado 25.1 realizando la operación de lectura asincrónicamente. La operación de escritura sigue realizándose sincrónicamente. La secuencia se inicializa de la misma manera, sin importar cómo se realiza la E/S. Las secuencias pueden manipularse asincrónicamente para todas las operaciones de E/S, de forma sincrónica para todas las operaciones de E/S o en una combinación de forma sincrónica y asincrónica.

Listado 25.2. Escritura sincrónica, lectura asincrónica

```
using System;
using System.IO;
using System.Threading;

class FileTestClass
{
    private FileStream BinaryFile;
    private byte [] ByteArray;
    private IAsyncResult AsyncResultImplementation;
    private AsyncCallback ReadBytesCompleteCallback;

    public FileTestClass()
    {
        AsyncResultImplementation = null;
        BinaryFile = new FileStream("test.dat", FileMode.Create,
        FileAccess.ReadWrite);
        ByteArray = new byte [256];
        ReadBytesCompleteCallback = new
        AsyncCallback(OnReadBytesComplete);
    }

    public void WriteBytes()
    {
        int ArrayIndex;

        for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
            ByteArray[ArrayIndex] = (byte)ArrayIndex;
        BinaryFile.Write(ByteArray, 0, 256);
    }

    public void ReadBytes()
    {
        BinaryFile.Seek(0, SeekOrigin.Begin);
```

```

        AsyncResultImplementation =
BinaryFile.BeginRead(ByteArray, 0, 256,
ReadBytesCompleteCallback, null);
    }

public void OnReadBytesComplete(IAAsyncResult AsyncResult)
{
    int ArrayIndex;
    int BytesRead;
    int Failures;
    BytesRead = BinaryFile.EndRead(AsyncResult);
    Console.WriteLine("Bytes read.....: {0}", BytesRead);
    Failures = 0;
    for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
    {
        if(ByteArray[ArrayIndex] != (byte)ArrayIndex)
        {
            Console.WriteLine("Read test failed for byte at
offset {0}.", ArrayIndex);
            Failures++;
        }
    }
    Console.WriteLine("Read test failures: {0}", Failures);
}

public void WaitForReadOperationToFinish()
{
    WaitHandle WaitOnReadIO;

    WaitOnReadIO = AsyncResultImplementation.AsyncWaitHandle;
    WaitOnReadIO.WaitOne();
}
}

class MainClass
{
    static public void Main()
    {
        FileTestClass FileTest = new FileTestClass();

        FileTest.WriteBytes();
        FileTest.ReadBytes();
        FileTest.WaitForReadOperationToFinish();
    }
}
}

```

El código de las operaciones de escritura del listado 25.2 se controla de forma sincrónica y su código es idéntico al código de las operaciones de escritura del listado 25.1. Sin embargo, la operación de lectura es bastante diferente.

El código de las operaciones de lectura del listado 25.2 no empieza con una llamada al método sincrónico `Read()` de la secuencia sino con una llamada al método asincrónico `BeginRead()`. Esta llamada admite cinco parámetros. Los

tres primeros concuerdan con los parámetros admitidos con el método sincrónico `Read()`, pero los dos últimos parámetros son nuevos:

- Una referencia al búfer de bytes que se usará para contener la lectura de bytes de la secuencia.
- Un número entero que especifica el elemento de matriz del primer byte en el búfer que contendrá la lectura de datos de la secuencia.
- Un número entero que especifica el número de bytes que se van a leer.
- Datos específicos de llamada.

El delegado `callback` debe ser un objeto de una clase llamada `AsyncCallback`. La clase `AsyncCallback` se declara en el espacio de nombres `System` de .NET Framework y gestiona un método que no devuelve nada y admite una referencia a un interfaz llamada `IAsyncResult`. El listado 25.2 crea una instancia de este delegado en el constructor de la clase `FileTestClass`:

```
ReadBytesCompleteCallback = new AsyncCallback(OnReadBytesComplete);
```

La clase `FileTestClass` del listado 25.2 incluye un nuevo método llamado `OnReadBytesComplete()`, que se usa como método delegado. El objeto `Stream` invoca a este delegado cuando se completa la operación de lectura.

La interfaz `IAsyncResult`, que se emplea como parámetro para el delegado `AsyncCallback`, se define en el espacio de nombres `System` de .NET Framework. Admite cuatro propiedades que pueden usarse para obtener más información sobre la naturaleza de la operación asincrónica:

- `AsyncState`, que es una referencia al objeto que se proporcionó como el último parámetro del método `BeginRead()`. Los métodos de E/S asincrónicos permiten asociar datos con una operación específica del último parámetro a un método de E/S. En la propiedad `AsyncState` hay disponible una copia de estos datos. El listado 25.2 no necesita que los datos se asocien a la llamada, por lo que pasa un valor `null` como último parámetro para `BeginRead()`. Como resultado, la propiedad `AsyncState` también tiene un valor `null`. Quizás quiera usar estos datos para, por ejemplo, diferenciar una llamada de E/S de otra. Por ejemplo, puede usar la misma referencia de delegado en varias llamadas de E/S asincrónicas y quizás quiera pasar junto a ella los datos que diferencian una llamada de otra.
- `AsyncWaitHandle`, que es una referencia a un objeto de clase `WaitHandle`. La clase `WaitHandle` se declara en el espacio de nombres `System.Threading` de .NET Framework. Este objeto encapsula una sincronización de primitivos y sirve de clase base para los primitivos específicos, como mutex y semáforos. El código puede esperar en este

control para determinar cuándo la operación de lectura está realmente terminada. El código del listado 25.2 hace precisamente eso.

- `CompletedSynchronously`, que es un booleano cuyo valor es `True` si la llamada `BeginRead()` se completa de forma sincrónica y `False` en caso contrario. Casi todas las implementaciones de secuencia devuelven `False` para esta propiedad cuando la interfaz hace referencia a una operación de E/S asincrónica.
- `IsCompleted`, que es un booleano cuyo valor es `True` si el objeto `Stream` ha completado la operación asincrónica. Hasta ese momento, la propiedad tiene un valor `False`. El código puede destruir todos los recursos relacionados con secuencias después de que la propiedad `IsCompleted` devuelve `True`.

La llamada a `BeginRead()` también devuelve una implementación de la interfaz `IAsyncCallback`. El código del listado 25.2 captura la referencia de interfaz para su uso posterior.

El método `AsyncCallback`, que, en el listado 25.2, es el método `OnReadBytesComplete()`, es invocado por el objeto `Stream` cuando se completa la operación asincrónica. La implementación que se muestra en el listado 25.2 empieza con una llamada a `EndRead()`, que devuelve el número de bytes que se leen realmente desde la operación. Este número debe ser igual al número de bytes que se solicitó leer mediante `BeginRead()`.

TRUCO: La llamada a `EndRead()` en el listado 25.2 se muestra para que se pueda encontrar el número de bytes afectados por la operación asincrónica. Si el código no necesita este valor, no es necesario llamar a `EndRead()`.

El resto de la implementación del método `OnReadBytesComplete()` comprueba el patrón de bytes leídos por la operación de E/S y envía sus resultados a la consola. El método `Main()` del listado 25.2 agrega una nueva llamada de método al código del listado 25.1, que es para un método privado del objeto `FileTestClass` llamado `WaitForReadOperationToFinish()`. Como la operación de lectura asincrónica es la última operación del código, la aplicación puede salir antes de que se complete la operación de lectura. Recuerde que el procesamiento de la operación de E/S asincrónica se realiza en un subproceso diferente. Si el proceso principal sale antes de que el subproceso de E/S pueda terminar, el código de `OnReadBytesComplete()` no tiene oportunidad de terminar. El método `WaitForReadOperationToFinish()` garantiza que la operación se complete antes de regresar a su invocador.

El método `WaitForReadOperationToFinish()` usa el temporizador de espera en la implementación de la interfaz `IAsyncCallback` para realizar

su trabajo. El método llama al método `WaitHandle WaitOne()` para que espere hasta que el temporizador de espera esté marcado. La llamada a `WaitOne()` no regresa hasta que el temporizador de espera está marcado. El objeto `Stream` marca el temporizador de espera sólo después de que la operación de E/S se complete. Después de que la llamada a `WaitOne()` regrese, puede estar seguro de que toda la operación se ha completado.

Cómo escribir de forma asincrónica

Las operaciones de E/S asincrónicas son parecidas a las operaciones de E/S sincrónicas. La única diferencia es que se emplea el método `BeginWrite()` en lugar del método `BeginRead()`. El listado 25.3 perfecciona el listado 25.2 al implementar una operación de escritura asincrónica.

Listado 25.3. Escritura asincrónica, lectura asincrónica

```
using System;
using System.IO;
using System.Threading;

class FileTestClass
{
    private FileStream BinaryFile;
    private byte [] ByteArray;
    private IAsyncResult AsyncReadResultImplementation;
    private IAsyncResult AsyncWriteResultImplementation;
    private AsyncCallback ReadBytesCompleteCallback;
    private AsyncCallback WriteBytesCompleteCallback;

    public FileTestClass()
    {
        AsyncReadResultImplementation = null;
        BinaryFile = new FileStream("test.dat", FileMode.Create,
FileAccess.ReadWrite);
        ByteArray = new byte [256];
        ReadBytesCompleteCallback = new
AsyncCallback(OnReadBytesComplete);
        WriteBytesCompleteCallback = new
AsyncCallback(OnWriteBytesComplete);
    }

    public void WriteBytes()
    {
        int ArrayIndex;

        for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
            ByteArray[ArrayIndex] = (byte)ArrayIndex;
        AsyncWriteResultImplementation =
BinaryFile.BeginWrite(ByteArray, 0, 256,
WriteBytesCompleteCallback, null);
    }
}
```

```

public void ReadBytes()
{
    WaitForWriteOperationToFinish();
    BinaryFile.Seek(0, SeekOrigin.Begin);
    AsyncReadResultImplementation =
BinaryFile.BeginRead(ByteArray, 0, 256,
ReadBytesCompleteCallback, null);
}

public void OnReadBytesComplete(IAsyncResult AsyncResult)
{
    int ArrayIndex;
    int BytesRead;
    int Failures;

    BytesRead = BinaryFile.EndRead(AsyncResult);
    Console.WriteLine("Bytes read.....: {0}", BytesRead);
    Failures = 0;
    for(ArrayIndex = 0; ArrayIndex < 256; ArrayIndex++)
    {
        if(ByteArray[ArrayIndex] != (byte)ArrayIndex)
        {
            Console.WriteLine("Read test failed for byte at
offset {0}.", ArrayIndex);
            Failures++;
        }
    }
    Console.WriteLine("Read test failures: {0}", Failures);
}

public void WaitForReadOperationToFinish()
{
    WaitHandle WaitOnReadIO;

    WaitOnReadIO =
AsyncReadResultImplementation.AsyncWaitHandle;
    WaitOnReadIO.WaitOne();
}

public void OnWriteBytesComplete(IAsyncResult AsyncResult)
{
    BinaryFile.EndWrite(AsyncResult);
}

private void WaitForWriteOperationToFinish()
{
    WaitHandle WaitOnWriteIO;

    WaitOnWriteIO =
AsyncWriteResultImplementation.AsyncWaitHandle;
    WaitOnWriteIO.WaitOne();
}
}

```

```

class MainClass
{
    static public void Main()
    {
        FileTestClass FileTest = new FileTestClass();

        FileTest.WriteBytes();
        FileTest.PeadBytes();
        FileTest.WaitForReadOperationToFinish();
    }
}

```

El método `EndWrite()` no devuelve un valor, a diferencia del método `EndRead()`. Sin embargo, los dos métodos se bloquean hasta que se termina la operación de E/S.

Escritores y lectores

.NET Framework tiene muchas clases escritoras y lectoras que facilitan el trabajo con datos más complejos que simples secuencias de bytes. Los lectores y los escritores encapsulan una secuencia y proporcionan un nivel de conversión que convierte los valores en sus secuencias de bytes equivalentes (para los escritores) y viceversa (para los lectores). Las clases lectoras y escritoras de .NET suelen tener un nombre que refleja el tipo de cambio de formato que realizan. Por ejemplo, la clase `HtmlTextWriter` escribe valores destinados para la información de respuesta de HTTP enviada por ASP.NET y la clase `StringReader` lee valores escritos usando su representación de cadena.

Las clases escritoras y lectoras también controlan varios esquemas de codificación, algo imposible de realizar mediante objetos de secuencia de bajo nivel. Por ejemplo, las clases derivadas de la clase abstracta `TextWriter`, permiten al código C# escribir texto y codificarlo en la secuencia mediante los algoritmos codificadores ASCII, Unicode, UTF7 o UTF8.

Cómo escribir secuencias con `BinaryWriter`

El listado 25.4 muestra la clase `BinaryWriter` en funcionamiento. La función de la clase `BinaryWriter` es convertir tipos de datos de C# en series de bytes que puedan escribirse en una secuencia subyacente.

Listado 25.4. Cómo trabajar con la clase `BinaryWriter`

```

using System;
using System.IO;

class FileTestClass
{

```

```

private BinaryWriter Writer;
private FileStream BinaryFile;

public FileTestClass()
{
    BinaryFile = new FileStream("test.dat", FileMode.Create,
FileAccess.ReadWrite);
    Writer = new BinaryWriter(BinaryFile);
}

public void WriteBinaryData()
{
    Writer.Write('a');
    Writer.Write(123);
    Writer.Write(456.789);
    Writer.Write("test string");
}
}

class MainClass
{
    static public void Main()
    {
        FileTestClass FileTest = new FileTestClass();

        FileTest.WriteBinaryData();
    }
}

```

El código del listado 25.4 tiene un diseño de clases igual al del listado 25.3. El código contiene una clase `MainClass` y una clase `FileTestClass`. El constructor de la clase `FileTestClass` del listado 25.4 crea una secuencia de archivos y, a continuación, un objeto `BinaryWriter`. El constructor del objeto `BinaryWriter`, que establece la relación entre el escritor binario y la secuencia en la que escribe sus datos, recibe una referencia a la secuencia de archivos. En el listado 25.4, todos los datos escritos en el escritor binario terminan en la secuencia de archivos establecida en el constructor. El método `WriteBinaryData()` escribe un carácter, un número entero, uno doble y una cadena, en la secuencia subyacente. La clase `BinaryWriter` implementa varias sobrecargas de un método llamado `Write()`. Las sobrecargas del método `Write()` admiten la escritura en la clase escritora de los siguientes tipos de datos:

- Booleanos
- Bytes
- Matrices de bytes
- Caracteres
- Matrices de caracteres
- Valores decimales

- Valores dobles
- Valores enteros cortos con y sin firma
- Valores enteros con y sin firma
- Valores enteros ampliados con y sin firma
- Sbytes
- Valores de coma flotante
- Cadenas

Si compila y ejecuta el código del listado 25.4, se crea un archivo llamado `test.dat`. Puede examinar los contenidos del nuevo archivo en un editor hexadecimal para comprobar que las representaciones binarias de los valores se han escrito en el archivo.

Cómo leer de secuencias con `BinaryReader`

El listado 25.5 agrega la clase `BinaryReader` al código del listado 25.5. Esta clase vuelve a ensamblar los bytes de la secuencia convirtiéndolos a sus tipos de datos originales y devuelve los valores al invocador.

Listado 25.5. Cómo trabajar con la clase `BinaryReader`

```
using System;
using System.IO;

class FileTestClass
{
    private BinaryReader Reader;
    private BinaryWriter Writer;
    private FileStream BinaryFile;

    public FileTestClass()
    {
        BinaryFile = new FileStream("test.dat", FileMode.Create,
FileAccess.ReadWrite);
        Writer = new BinaryWriter(BinaryFile);
        Reader = new BinaryReader(BinaryFile);
    }

    public void ReadBinaryData()
    {
        char ReadCharacter;
        double ReadDouble;
        int ReadInteger;
        string ReadString;

        BinaryFile.Seek(0, SeekOrigin.Begin);
        ReadCharacter = Reader.ReadChar();
```

```

    ReadInteger = Reader.ReadInt32();
    ReadDouble = Reader.ReadDouble();
    ReadString = Reader.ReadString();

    Console.WriteLine("Character: {0}", ReadCharacter);
    Console.WriteLine("Integer: {0}", ReadInteger);
    Console.WriteLine("Double: {0}", ReadDouble);
    Console.WriteLine("String: {0}", ReadString);
}

public void WriteBinaryData()
{
    Writer.Write('a');
    Writer.Write(123);
    Writer.Write(456.789);
    Writer.Write("test string");
}

}

class MainClass
{
    static public void Main()
    {
        FileTestClass FileTest = new FileTestClass();

        FileTest.WriteBinaryData();
        FileTest.ReadBinaryData();
    }
}

```

A diferencia de la clase `BinaryWriter`, que contiene un método sobrecargado para todas las operaciones, la clase `BinaryReader` contiene un método de lectura distinto para cada tipo de datos. El código del listado 25.5 usa algunos de estos métodos de lectura, como `ReadChar()` y `ReadInt32()`, para leer valores de la secuencia escrita en el método `WriteBinaryData()`. Los valores que se leen de la secuencia se envían a la consola. Al ejecutar el listado 25.5 se obtiene el siguiente resultado en la consola.

```

Character: a
Integer: 123
Double: 456.789
String: test string

```

Cómo escribir XML con un formato correcto mediante la secuencia `XmlWriter`

Las secuencias pueden hacer más cosas aparte de leer y escribir datos en varias secuencias de datos. También pueden agregar valor a los datos que se

envían a través de la secuencia. Un buen ejemplo de esta tecnología es la clase `XmlWriter`, que encapsula en elementos XML con formato correcto los datos que se envían a una secuencia. El resultado es un documento XML con formato correcto con el que puede trabajar cualquier procesador de documentos XML, como muestra el listado 25.6.

Listado 25.6. Cómo escribir XML con la clase `XmlWriter`

```
using System;
using System.IO;
using System.Xml;

class XMLStreamWriterClass
{
    private XmlTextWriter XmlWriter;

    public void WriteXML()
    {
        XmlWriter = new XmlTextWriter(Console.Out);

        XmlWriter.WriteStartDocument();
        XmlWriter.WriteComment("This XML document was
automatically generated by C# code.");
        XmlWriter.WriteStartElement("BOOK");
        XmlWriter.WriteElementString("TITLE", "C# Bible");
        XmlWriter.WriteElementString("AUTHOR", "Jeff Ferguson");
        XmlWriter.WriteElementString("PUBLISHER", "Wiley");
        XmlWriter.WriteEndElement();
        XmlWriter.WriteEndDocument();
    }
}

class MainClass
{
    static public void Main()
    {
        XMLStreamWriterClass XMLStreamWriter = new
XMLStreamWriterClass();

        XMLStreamWriter.WriteXML();
    }
}
```

El código del listado 25.6 crea una nueva instancia de la clase `XmlWriter` y la asocia a la secuencia de salida de la consola. El código simplemente llama a varios métodos de la clase `XmlWriter` para producir datos y los métodos rodean a esos datos con nombres de elementos XML que se especifican cuando se llama al método.

Observe la siguiente línea del listado 25.6:

```
XmlWriter.WriteElementString("AUTHOR", "Jeff Ferguson");
```

Esta llamada ordena a la clase `XmlWriter` que escriba un elemento XML llamado `<AUTHOR>` cuyo valor es `<Brian Patterson>`, al dispositivo de salida de la secuencia. La implementación del método proporciona automáticamente la etiqueta de cierre XML.

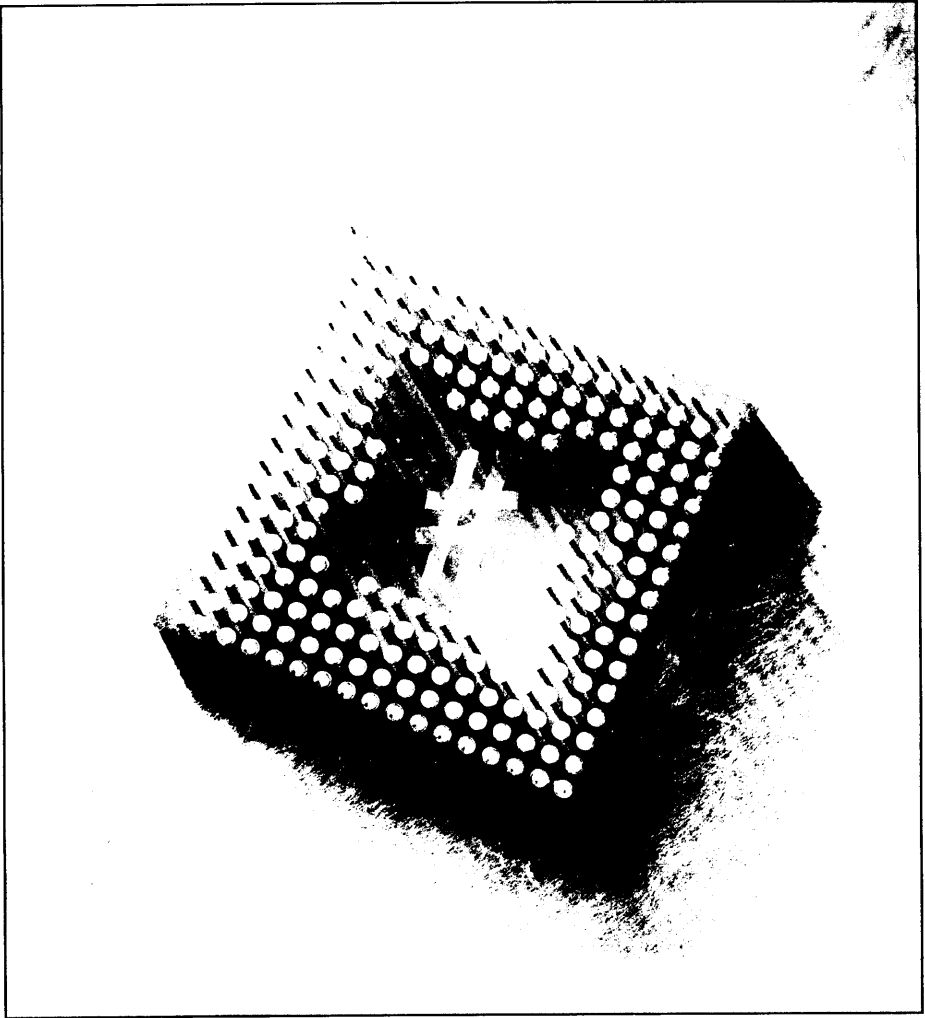
Al compilar y ejecutar el código del listado 25.6, se envía el siguiente documento XML con formato correcto a la consola de la aplicación:

```
<?xml version="1.0" encoding="IBM437"?>
<!--This XML document was automatically generated by C# code.--
>
<BOOK>
<TITLE>C# Bible</TITLE>
<AUTHOR>Jeff Ferguson</AUTHOR>
<PUBLISHER>Wiley</PUBLISHER>
</BOOK>
```

Resumen

Las secuencias proporcionan compatibilidad con la E/S sincrónica y asincrónica en las aplicaciones de C#. Las secuencias trabajan en el nivel de bytes y necesitan que se lean y escriban bloques de bytes. Los lectores y escritores encapsulan secuencias y proporcionan acceso a los datos en un nivel superior. Puede usar lectores y escritores para trabajar con los tipos de datos estándar de C#. lo que permite que los lectores y los escritores hagan conversiones entre los valores de tipos de datos y sus representaciones de bytes.

Su código C# seguramente trabajará con lectores y escritores, ya que proporcionan compatibilidad para trabajar con los tipos de datos estándar sin tener que ocuparse de realizar la conversión entre un valor de un tipo de dato y su representación binaria. Sin embargo, las secuencias también están disponibles si cree que necesita trabajar con ellas directamente. Quizás también quiera trabajar con secuencias si los datos que está leyendo están en un formato propietario que no es compatible con las clases lectoras y escritoras estándar de .NET Framework. También puede considerar la posibilidad de crear sus propias clases lectoras, derivadas de las clases base `TextReader` o `StreamReader` y usarlas para leer la secuencia de formato propietario.



26 **Cómo dibujar con GDI+**

En Windows, el acceso mediante programación al subsistema de gráficos se consiguió por primera vez usando los API GDI disponibles desde Windows 3.1. GDI ofrecía a los programadores la posibilidad de controlar cualquier tipo de elemento de interfaz de usuario y esta función ha sido reconstruida desde cero en .NET Framework. GDI+ ha reemplazado a GDI como el API que se usa para acceder a los subsistemas de gráficos de Windows. Con GDI+ puede acceder a fuentes, manipular cualquier tipo de imagen y trabajar con formas en las aplicaciones de C#. Para conseguir una visión global de cómo usar GDI+ en sus aplicaciones, debe comprender el modo de empleo de los objetos `Graphics`, `Pen`, `Brush` y `Color`. Con estos cuatro objetos, puede conseguir casi cualquier cosa que necesite hacer con las GUI y las imágenes de .NET. Este capítulo estudia estos objetos y le familiariza con el uso de GDI+ en C#. Las clases disponibles en GDI+ podrían llenar un libro de mil páginas, por lo que debe seguir usando SDK como referencia para la más compleja y menos usada funcionalidad gráfica que no se trata en este capítulo.

Cómo trabajar con gráficos

Al trabajar con GDI+ en .NET, el principal objeto con el que se debe trabajar es el objeto `Graphics`. Este objeto es la superficie real que se usa para pintar

formas, trabajar con imágenes o mostrar texto. Visual Basic 6 y versiones anteriores incorporaban una limitada compatibilidad para trabajar con gráficos, lo que dificultaba a los programadores de VB la tarea de escribir aplicaciones gráficas personalizadas.

Lo que VB hacía era mantener un registro de cómo se dibujaban en pantalla los formularios y los objetos de los formularios. La propiedad `AutoRedraw` permitía que los formularios dejaran a Windows mantener un registro de lo que estaba en la parte superior de las demás ventanas y, en caso de necesidad, dibujar de nuevo automáticamente un formulario si otro estaba encima de él durante un cierto periodo de tiempo. No hacía falta que se ocupara del proceso real de dibujar el formulario.

En .NET, sucede todo lo contrario. El objeto `Graphics` no recuerda cuándo fue dibujado ni qué fue lo que dibujó. Por tanto, es necesario dibujar de nuevo los objetos tantas veces como resulte necesario si otras ventanas están encima de una ventana concreta. Esto puede parecer pesado, pero la variable `PaintEventArgs` del evento `Paint` de un formulario puede controlar el proceso perfectamente. Si el código de dibujo se mantiene allí, cada vez que Windows pinte el formulario, los objetos se generarán correctamente.

El siguiente fragmento de código recibe una referencia a un objeto `Graphics` mediante la variable `PaintEventArgs` del evento `Paint` de un formulario:

```
private void Form1_Paint(object sender,
    System.Windows.Forms.PaintEventArgs p)
{
    Graphics g = p.Graphics;
}
```

También puede crear un objeto `Graphics` mediante el método `CreateGraphics` de un control o formulario. El siguiente código muestra el método `CreateGraphics`:

```
private void createManually()
{
    Graphics g;
    g = this.CreateGraphics;
}
```

El tercer y último modo de crear un objeto `Graphics` es pasar un archivo de imagen directamente al objeto al instanciarlo, como muestra el siguiente código al tomar una imagen de mapa de bits del sistema de archivos:

```
private void createFromFile()
{
    Graphics g;
    Bitmap b;
    b = new Bitmap(@"C:\Enterprise.bmp");
    g = Graphics.FromImage(b);
}
```

Si ha estado agregando estos fragmentos a un formulario de Windows, es obvio que no sucede nada cuando se ejecuta alguno de estos códigos. Para implementar realmente alguna funcionalidad, debe usar miembros de la clase `Graphics` para hacer que sucedan cosas.

NOTA: Si crea un objeto `Graphics` mediante el método `CreateGraphics`, debe llamar a `Dispose` en ese objeto después de usarlo. Así se asegura de que el objeto `Graphics` se eliminará de la memoria.

La tabla 26.1 enumera las propiedades de la clase `Graphics` y la tabla 26.2 enumera los métodos disponibles de la clase `Graphics`. La clase `Graphics` se incluye en el espacio de nombres `System.Drawing`, que se agrega como referencia por defecto cuando se crea una nueva aplicación Windows Forms. Esto no significa que no se puedan usar objetos `Graphics` en ASP.NET; de hecho, en ASP.NET se pueden escribir aplicaciones de proceso de imágenes extremadamente depuradas usando objetos `Graphics`.

Tabla 26.1. Propiedades de la clase `Graphics`

Propiedad	Descripción
<code>Clip</code>	Obtiene o establece un objeto <code>Region</code> que delimita la región de dibujo de este objeto <code>Graphics</code>
<code>ClipBounds</code>	Obtiene la estructura <code>RectangleF</code> que delimita la región de recorte de este objeto <code>Graphics</code>
<code>CompositingMode</code>	Obtiene un valor que especifica cómo se dibujan las imágenes compuestas en el objeto <code>Graphics</code>
<code>CompositingQuality</code>	Obtiene o establece la calidad de la generación de las imágenes compuestas que se dibujan en el objeto <code>Graphics</code>
<code>DpiX</code>	Obtiene la resolución horizontal de este objeto <code>Graphics</code>
<code>DpiY</code>	Obtiene la resolución vertical de este objeto <code>Graphics</code>
<code>InterpolationMode</code>	Obtiene o establece el modo de interpolación asociado al objeto <code>Graphics</code>
<code>IsClipEmpty</code>	Obtiene un valor que indica si la región de recorte de este objeto <code>Graphics</code> está vacía
<code>IsVisibleClipEmpty</code>	Obtiene un valor que indica si la región visible de recorte de este objeto <code>Graphics</code> está vacía

Propiedad	Descripción
PageScale	Obtiene o establece la relación de escala entre las unidades universales y las unidades de página de este objeto <code>Graphics</code>
PageUnit	Obtiene o establece la unidad de medida usada para las coordenadas de página de este objeto <code>Graphics</code>
PixelOffsetMode	Obtiene o establece un valor que especifica cómo se desplazan los píxeles durante el procesamiento de este objeto <code>Graphics</code>
RenderingOrigin	Obtiene o establece el origen de la generación de este objeto <code>Graphics</code> para la interpolación y los pinceles de trama
SmoothingMode	Obtiene o establece la calidad de la generación del objeto <code>Graphics</code>
TextContrast	Obtiene o establece el valor de la corrección gamma para la generación de texto
TextRenderingHint	Obtiene o establece el modo de generación para el texto asociado a este objeto <code>Graphics</code>
Transform	Obtiene o establece la transformación universal de este objeto <code>Graphics</code>
VisibleClipBounds	Obtiene o establece el rectángulo delimitador de este objeto <code>Graphics</code>

Tabla 26.2. Métodos de la clase `Graphics`

Método	Descripción
AddMetafileComment	Agrega un comentario al objeto <code>Metafile</code> actual
BeginContainer	Guarda un contenedor <code>Graphics</code> con el estado actual de este objeto <code>Graphics</code> y abre y usa un nuevo contenedor de gráficos.
Clear	Limpia toda la superficie de dibujo y la rellena con el color de fondo especificado
DrawArc	Dibuja un arco que representa una porción de una elipse especificada por un par de coordenadas, un valor de altura y un valor de anchura
DrawBezier	Dibuja una curva spline Bézier definida por cuatro estructuras <code>Point</code>

Método	Descripción
DrawBeziers	Dibuja una serie de curvas spline Bézier a partir de una matriz de estructuras <code>Point</code>
DrawClosedCurve	Dibuja una curva spline cardinal cerrada definida por una matriz de estructuras <code>Point</code>
DrawCurve	Dibuja una curva spline cardinal mediante una matriz de estructuras <code>Point</code> especificada
DrawEllipse	Dibuja una elipse definida por un rectángulo delimitador especificada por un par de coordenadas, un valor de altura y un valor de anchura
DrawIcon	Dibuja la imagen representada por el objeto <code>Icon</code> especificado en las coordenadas especificadas
DrawIconUnstretched	Dibuja la imagen representada por el objeto <code>Icon</code> especificado sin escalar la imagen
DrawImage	Dibuja el objeto <code>Image</code> especificado en la posición especificada y con el tamaño original
DrawImageUnscaled	Dibuja el objeto <code>Image</code> especificado con su tamaño original en la posición especificada por un par de coordenadas
DrawLine	Dibuja una línea que conecta los dos puntos especificados por pares de coordenadas
DrawLines	Dibuja una serie de segmentos de línea que conectan una matriz de estructuras <code>Point</code>
DrawPath	Dibuja un objeto <code>GraphicsPath</code>
DrawPie	Dibuja una forma circular definida por una elipse especificada por un par de coordenadas, un valor de altura y un valor de anchura y dos líneas radiales
DrawPolygon	Dibuja un polígono definido por una matriz de estructuras <code>Point</code>
DrawRectangle	Dibuja un rectángulo especificado por un par de coordenadas, un valor de alto y un valor de ancho
DrawRectangles	Dibuja una serie de rectángulos especificados por estructuras <code>Rectangle</code>
DrawString	Dibuja la cadena de texto especificada en la posición especificada con los objetos <code>Brush</code> y <code>Font</code> especificados
EndContainer	Cierra el contenedor de gráficos activo y restaura el estado que tenía este objeto <code>Graphics</code> al esta-

Método	Descripción
	do guardado por una llamada al método <code>BeginContainer</code>
<code>EnumerateMetafile</code>	Envía los registros del objeto <code>Metafile</code> especificado, de uno en uno, a un método de devolución de llamada para su presentación en un punto determinado
<code>ExcludeClip</code>	Actualiza la región de recorte de este objeto <code>Graphics</code> con el fin de excluir el área especificada por una estructura <code>Rectangle</code>
<code>FillClosedCurve</code>	Rellena el interior de una curva spline cardinal cerrada, definida por una matriz de estructuras <code>Point</code>
<code>FillEllipse</code>	Rellena el interior de una elipse definida por un rectángulo de delimitación especificado por un par de coordenadas, un valor de altura y un valor de anchura
<code>FillPath</code>	Rellena el interior de un objeto <code>GraphicsPath</code>
<code>FillPie</code>	Rellena el interior de una sección de gráfico circular definida por una elipse, determinada por un par de coordenadas, unos valores de anchura y altura y dos líneas radiales
<code>FillPolygon</code>	Rellena el interior de un polígono definido por una matriz de puntos, especificados por estructuras <code>Point</code>
<code>FillRectangle</code>	Rellena el interior de un rectángulo especificado por un par de coordenadas, un valor de anchura y un valor de altura
<code>FillRectangles</code>	Rellena el interior de una serie de rectángulos especificados por estructuras <code>Rectangle</code>
<code>FillRegion</code>	Rellena el interior de un objeto <code>Region</code>
<code>Flush</code>	Fuerza la ejecución de todas las operaciones de gráficos pendientes y devuelve inmediatamente el control sin esperar a que finalicen las operaciones
<code>FromHdc</code>	Crea un nuevo objeto <code>Graphics</code> a partir del identificador especificado en un contexto de dispositivo
<code>FromHwnd</code>	Crea un nuevo objeto <code>Graphics</code> a partir del identificador especificado de una ventana
<code>FromImage</code>	Crea un nuevo objeto <code>Graphics</code> a partir del objeto <code>Image</code> especificado

Método	Descripción
GetHalftonePalette	Obtiene un identificador de la paleta actual de semitonos de Windows
GetHdc	Obtiene el identificador del contexto de dispositivo asociado a este objeto <code>Graphics</code>
GetNearestColor	Obtiene el color más próximo a la estructura <code>Color</code> especificada
IntersectClip	Actualiza la región de recorte de este objeto <code>Graphics</code> como la intersección de la región de recorte actual y la estructura <code>Rectangle</code> especificada
IsVisible	Indica si el punto especificado por un par de coordenadas está contenido en la región de recorte visible de este objeto <code>Graphics</code>
MeasureCharacterRanges	Obtiene una matriz de objetos <code>Region</code> , cada uno de los cuales delimita un intervalo de posiciones de caracteres dentro de la cadena especificada
MeasureString	Mide la cadena especificada al dibujarla con el objeto <code>Font</code> especificado
MultiplyTransform	Multiplica la transformación universal de este objeto <code>Graphics</code> y especificada en el objeto <code>Matrix</code>
ReleaseHdc	Libera un identificador de contexto de dispositivo obtenido mediante una llamada anterior al método <code>GetHdc</code> de este objeto <code>Graphics</code>
ResetClip	Restablece la región de recorte de este objeto <code>Graphics</code> en una región infinita
ResetTransform	Restablece la matriz de transformación universal de este objeto <code>Graphics</code> en la matriz de identidades
Restore	Restaura como estado de este objeto <code>Graphics</code> el estado representado por un objeto <code>GraphicsState</code>
RotateTransform	Aplica la rotación especificada a la matriz de transformación de este objeto <code>Graphics</code>
Save	Guarda el estado actual de este objeto <code>Graphics</code> e identifica el estado guardado con un objeto <code>GraphicsState</code>
ScaleTransform	Aplica la operación de cambio de escala especificada a la matriz de transformación de este objeto <code>Graphics</code> , anteponiéndola a esta última

Método	Descripción
SetClip	Establece la región de recorte de este objeto <code>Graphics</code> en la propiedad <code>Clip</code> del objeto <code>Graphics</code> especificado
TransformPoints	Transforma una matriz de puntos de un espacio de coordenadas a otro utilizando las transformaciones universal y de página actuales de este objeto <code>Graphics</code>
TranslateClip	Convierte la región de recorte de este objeto <code>Graphics</code> usando las cantidades especificadas en las direcciones horizontal y vertical
TranslateTransform	Antepone la conversión especificada a la matriz de transformación de este objeto <code>Graphics</code>

Como puede ver, la clase `Graphics` proporciona todos los métodos posibles que pueda necesitar para trabajar con cualquier tipo de elemento GUI. El listado 26.1 usa muchos de los métodos de la clase `Graphics` para producir el resultado que se muestra en la figura 26.1.

NOTA: Como no hay una propiedad `AutoRedraw`, todavía necesita un modo de volver a dibujar un formulario si se le asigna un nuevo tamaño. Si se usa el método `SetStyles` y se pasa el estilo `ControlStyles.ResizeRedraw` correctamente, se llamará al método `Paint` de un formulario para corregir su estilo. Tras la llamada, el objeto `InitializeComponent` de su formulario debe escribir `SetStyle(ControlStyles.ResizeRedraw, true)` para garantizar que se llamará al evento `Paint` cuando el formulario cambie de tamaño. Busque `SetStyle` en .NET Framework SDK para aprender más sobre lo que puede hacer con el método `SetStyle`.

Listado 26.1. Cómo usar métodos de la clase `Graphics`

```
private void drawLine()
{
    /* crea un objeto Graphics que puede ser recuperado
    * para cada una de las muestras */

    Graphics g;
    g = this.CreateGraphics();

    // Use el objeto Pen para crear una línea

    Pen p;
```

```

p = new Pen(Color.Red, 50);

/* DrawLine es un metodo sobrecargado,
 * pasa las coordenadas x1, y1, x2, y2 */

g.DrawLine(p, 100F, 100F, 500F, 100F);

// dibuje un icono del sistema de archivos

Icon i;
i = new Icon(@"C:\Desktop.ico");

// llame a DrawIcon y pasa las coordenadas x e y

g.DrawIcon(i, 150, 15);

// dibuje un rectangulo

Pen p2;
p2 = new Pen(Color.PapayaWhip, 7);

/* dibuje un rectangulo pasando x, y,
 * altura y anchura */

g.DrawRectangle(p2, 50, 50, 100, 100);
}

```

Si llama a este método desde una aplicación Windows Forms, su resultado se parecerá a lo que aparece en la figura 26.1.

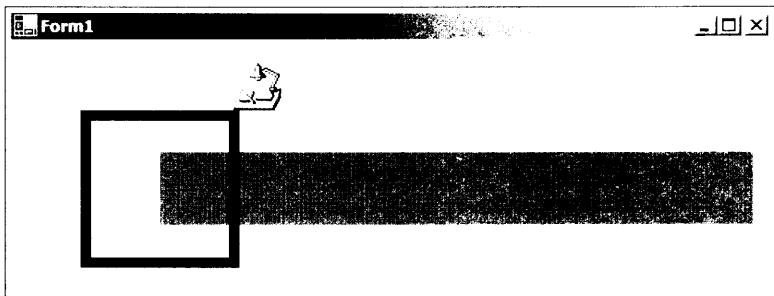


Figura 26.1. Resultado del uso de miembros de la clase Graphics

La clase `Graphics` no hace nada por sí misma. Para crear líneas, rectángulos, imágenes y fuentes, debe usar otros objetos junto al objeto `Graphics`. El listado 26.1 crea un objeto `Pen` para usarlo en conjunción con el objeto `Graphics` para dibujar una línea roja en el formulario. También crea un objeto `Icon`, que es usado por el objeto `Graphics` para dibujar el icono de escritorio en el formulario. Como ya se dijo antes, puede realizar numerosas tareas con `GDI+`: dibujar formas y líneas, manipular imágenes y trabajar con fuentes. Las siguientes secciones profundizan en este tema, describiendo cómo puede usar objetos como

Pen, Brush e Image en combinación con miembros de la clase Graphics para aprovechar al máximo la inmensa colección de utilidades GDI+ de .NET.

Cómo trabajar con Image en GDI+

Si necesita procesar imágenes existentes en el sistema de archivos, la clase Image le brinda la posibilidad de procesar imágenes en superficies creadas con el objeto Graphics. La clase Image se incluye en el espacio de nombres System.Drawing y es una clase abstracta que le ofrece toda la funcionalidad que necesita para usar mapas de bits, iconos y metarchivos con un objeto Graphics para procesar objetos Image predefinidos en un formulario. Las imágenes procesadas pueden proceder directamente del sistema de archivos o de una secuencia de memoria. En cualquier caso, está tratando con algún tipo de fuente de imagen. Las imágenes pueden ser de tipo JPG, ICO o BMP.

En el listado 26.2, se abre un archivo JPG del disco local para que aparezca en un formulario. Este ejemplo es ligeramente diferente del que vimos anteriormente. En éste, se sobrecarga el evento OnPaint del formulario para que la imagen no sea eliminada si se sitúa otra ventana sobre la suya. Este ejemplo muestra cómo implementar llamadas al método Dispose en el objeto Graphics que se usa para pintar la imagen JPG cuando se destruye el formulario.

Listado 26.2. Cómo usar imágenes con GDI+

```
namespace RenderJPG
{
    public class Form1 : System.Windows.Forms.Form
    {
        private System.ComponentModel.Container
            components = null;

        // declara la variable de imagen
        private Image img;

        public Form1()
        {
            InitializeComponent();

            // abre la imagen
            img = new Bitmap(@"C:\money.jpg");
            //
        }

        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                // Llama a DISPOSE sobre el objeto Img
            }
        }
    }
}
```

```

        img.Dispose();
        //
        if (components != null)
            {
                components.Dispose();
            }
    }
    base.Dispose( disposing );
}
static void Main()
{
    Application.Run(new Form1());
}

// sobrecarga el evento OnPaint
protected override void OnPaint(PaintEventArgs p)
{
    Graphics g = p.Graphics;
    g.DrawImage(img, 0,0);
}
}
}

```

Si se ejecuta esta aplicación se producirá un resultado parecido al que aparece en la figura 26.2.

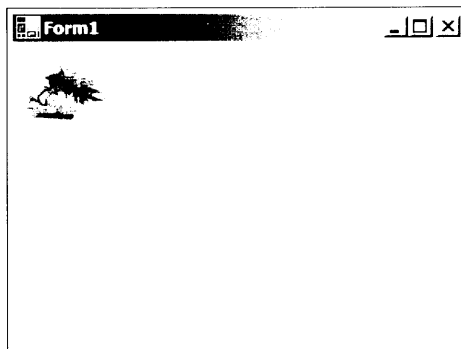


Figura 26.2. Resultado del listado 26.2 usando un JPG con GDI+

El método `DrawImage` usado para dibujar la imagen en el formulario tiene casi 20 constructores sobrecargados. Básicamente, cada uno indica al método cómo dibujar la imagen, como coordenadas o como altura y anchura. Con un simple cambio en el método `DrawImage`, puede rellenar todo el formulario con el mapa de bits. Si pasa la constante `ClientRectangle` a `DrawImage`, como muestra el fragmento de código, obtendrá un resultado con un aspecto similar al de la figura 26.3, con todo el mapa de bits rellenando la pantalla:

```

// sobrecarga del evento OnPaint
protected override void OnPaint(PaintEventArgs p)
{

```



```

Graphics g = p.Graphics;
g.DrawImage(img, ClientRectangle);
}

```

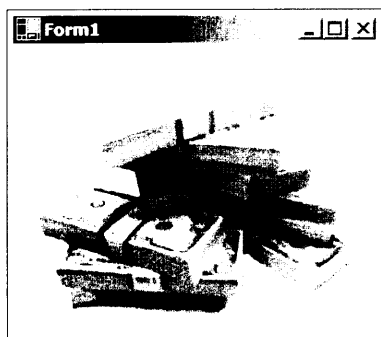


Figura 26.3. La imagen rellena todo el formulario

También puede devolver propiedades a una imagen sin mostrarla. El siguiente evento `Load` examina algunas de las propiedades disponibles de la imagen `money.jpg` que abrimos antes:

```

private void Form1_Load(object sender, System.EventArgs e)
{
    MessageBox.Show
        (img.PhysicalDimension.ToString() );
    MessageBox.Show
        (img.Height.ToString() );
    MessageBox.Show
        (img.Width.ToString() );
    MessageBox.Show
        (img.RawFormat.ToString() );
    MessageBox.Show
        (img.Size.ToString() );
}

```

La tabla 26.3 describe cada una de las propiedades disponibles para imágenes a través de la clase `Image`.

Tabla 26.3. Propiedades de la clase `Image`

Propiedad	Descripción
Flags	Obtiene indicadores de atributo para este objeto <code>Image</code>
FrameDimensionsList	Obtiene una matriz GUID que representa las dimensiones de los marcos de este objeto <code>Image</code>
Height	Obtiene la altura de este objeto <code>Image</code>

Propiedad	Descripción
HorizontalResolution	Obtiene la resolución horizontal, en píxeles por pulgada, de este objeto <code>Image</code>
Palette	Obtiene o establece la paleta de colores de este objeto <code>Image</code>
PhysicalDimension	Obtiene la anchura y altura de este objeto <code>Image</code>
PixelFormat	Obtiene el formato de píxeles de este objeto <code>Image</code>
PropertyIdList	Obtiene una matriz de los identificadores de propiedad almacenados en este objeto <code>Image</code>
PropertyItems	Obtiene una matriz de objetos <code>PropertyItem</code> que describe este objeto <code>Image</code>
RawFormat	Obtiene el formato de este objeto <code>Image</code>
Size	Obtiene la anchura y altura de este objeto <code>Image</code>
VerticalResolution	Obtiene la resolución vertical, en píxeles por pulgada, de este objeto <code>Image</code>
Width	Obtiene el ancho de este objeto <code>Image</code>

También puede usar varios métodos de la clase `Image`, que le permite manipular imágenes de un modo prácticamente ilimitado. El siguiente código voltea la imagen 90 grados:

```
img.RotateFlip(RotateFlipType.Rotate90FlipY);
```

La enumeración `RotateFlipType` permite especificar cómo quiere girar o voltear una imagen sobre una superficie de gráficos.

La tabla 26.4 enumera los métodos restantes de la clase `Image` que puede usar para manipular una imagen.

Tabla 26.4. Métodos de la clase `Image`

Método	Descripción
Clone	Crea una copia exacta de este objeto <code>Image</code>
FromFile	Crea un objeto <code>Image</code> a partir del archivo especificado
FromHbitmap	Crea un objeto <code>Bitmap</code> a partir de un identificador de Windows
FromStream	Crea un objeto <code>Image</code> a partir de la secuencia de datos especificada

Método	Descripción
GetBounds	Obtiene un rectángulo delimitador para este objeto <code>Image</code> en las unidades especificadas
GetEncoderParameterList	Devuelve información sobre los parámetros que admite el codificador de imágenes especificado
GetFrameCount	Devuelve el número de marcos de la dimensión especificada
GetPixelFormatSize	Devuelve la profundidad de color (número de bits por píxel) del formato de píxel especificado
GetPropertyItem	Obtiene el elemento de propiedad especificado de este objeto <code>Image</code>
GetThumbnailImage	Devuelve la vista en miniatura de este objeto <code>Image</code>
IsAlphaPixelFormat	Devuelve un valor que indica si el formato de píxel de este objeto <code>Image</code> contiene información alfa
IsCanonicalPixelFormat	Devuelve un valor que indica si el formato de píxel es canónico
IsExtendedPixelFormat	Devuelve un valor que indica si el formato de píxel es extendido
RemovePropertyItem	Quita el elemento de propiedad especificado de este objeto <code>Image</code>
RotateFlip	Este método gira, voltea o gira y voltea el objeto <code>Image</code>
Save	Guarda este objeto <code>Image</code> en el objeto <code>Stream</code> con el formato especificado
SaveAdd	Agrega la información del objeto <code>Image</code> especificado a este objeto <code>Image</code> . El objeto <code>EncoderParameters</code> especificado determina cómo se incorpora la nueva información a la imagen existente
SelectActiveFrame	Selecciona el marco que especifica la dimensión y el índice
SetPropertyItem	Establece el elemento de propiedad especificado en el valor especificado

Como ha podido ver en esta sección, la clase `Image` ofrece funciones muy sólidas cuando se usan con un objeto `Graphics`. En la siguiente sección aprenderá a usar lápices y pinceles para trabajar con imágenes y dibujar formas y líneas.

Cómo trabajar con lápices y pinceles

Como vimos en la clase `Image`, el espacio de nombres `System.Drawing` ofrece todo lo que necesitamos para trabajar con imágenes procedentes de una secuencia o del sistema de archivos. .NET Framework también ofrece compatibilidad integrada para trabajar con formas, líneas e imágenes mediante las clases `Pen` y `Brush`. Esta sección muestra cómo trabajar con las clases `Pen` y `Brush` para manipular formas, líneas e imágenes y lograr los efectos deseados.

Cómo usar la clase `Pen`

La clase `Pen` permite dibujar líneas y curvas sobre una superficie de gráficos. El espacio de nombres que contiene las funciones que usan las clases `Pen` y `Brush` es el espacio de nombres `System.Drawing.Drawing2D`, de modo que asegúrese de agregarlo junto a la instrucción de uso de sus archivos de clase. Al establecer varias propiedades en una instancia de `Pen`, puede modificar la apariencia de lo que muestra el lápiz. Al invocar métodos de la clase `Graphics`, puede indicar el tipo de forma que quiere mostrar.

El siguiente código establece las propiedades `Color` y `DashStyle` para crear una elipse similar a la que aparece en la figura 26.4.

```
private void Form1_Load(object sender,
    System.EventArgs e)
{
    Pen p = new Pen(Color.Blue, 10);
    p.DashStyle = DashStyle.DashDot ;
    Graphics g = this.CreateGraphics();
    g.DrawEllipse(p, 10, 15, 105, 250);
}
```

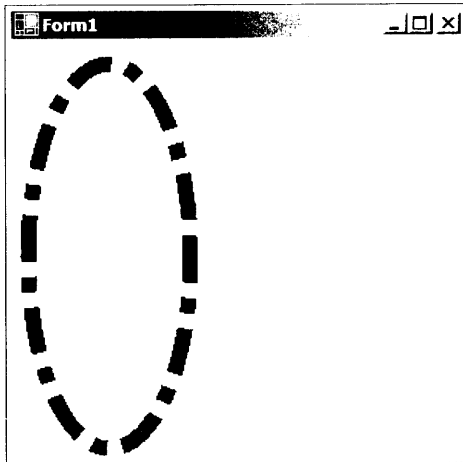


Figura 26.4. Dibujo de una elipse usando las propiedades `Color` y `DashStyle`

La tabla 26.5 recoge los valores posibles de la enumeración usada para establecer el estilo de la línea discontinua de la elipse.

Tabla 26.5. Enumeración DashStyle

Valor	Descripción
Custom	Especifica un estilo de guión personalizado definido por el usuario
Dash	Especifica una línea formada por guiones
DashDot	Especifica una línea formada por un modelo de guión y punto que se repite
DashDotDot	Especifica una línea formada por un modelo de guión, punto y punto que se repite
Dot	Especifica una línea formada por puntos
Solid	Especifica una línea continua

También puede personalizar líneas con las propiedades `StartCap` y `EndCap` usando la enumeración `LineCap`, que también se incluye en el espacio de nombres `System.Drawing.Drawing2D`. El listado 26.3 muestra algunas variaciones que usa la enumeración `LineCap` para dibujar diferentes tipos de líneas, cuyo resultado puede ver en la figura 26.5.

Listado 26.3. Cómo usar la enumeración `LineCap`

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    Pen p = new Pen(Color.Brown, 15);

    // establece la flecha
    p.StartCap = LineCap.ArrowAnchor ;
    p.EndCap = LineCap.ArrowAnchor ;
    g.DrawLine(p,30, 30, Width-50, 30);

    // extremos redondeados
    p.StartCap = LineCap.Round ;
    p.EndCap = LineCap.Round ;
    g.DrawLine(p,30, 80, Width-50, 80);

    // delimitador redondo
    p.StartCap = LineCap.RoundAnchor ;
    p.EndCap = LineCap.RoundAnchor ;
    g.DrawLine(p,30, 120, Width-50, 120);

    // triangulo
```

```

p.StartCap = LineCap.Triangle ;
p.EndCap = LineCap.Triangle ;
g.DrawLine(p,30, 150, Width-50, 150);

// delimitador cuadrado
p.StartCap = LineCap.SquareAnchor;
p.EndCap = LineCap.SquareAnchor ;
g.DrawLine(p,30, 190, Width-50, 190);
}

```

La figura 26.5 muestra el resultado de la ejecución del código anterior usando la enumeración `LineCap`.

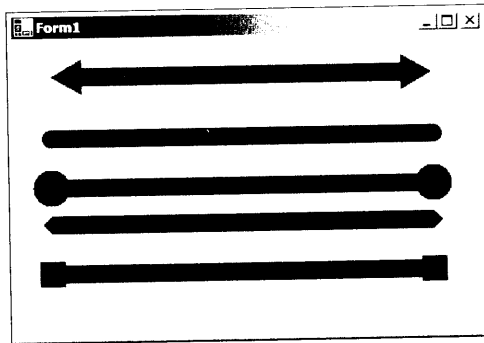


Figura 26.5. Cómo usar la enumeración `LineCap`

Cómo usar la clase `Brush`

El uso de la clase `Brush` en conjunción con un objeto `Graphics` permite procesar imágenes y objetos sólidos sobre una superficie de gráficos. El siguiente código muestra cómo crear una elipse continua rellena:

```

protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    SolidBrush sb = new SolidBrush(Color.Black);
    g.FillEllipse(sb, ClientRectangle);
}

```

Si se ejecuta el código anterior se produce una imagen como la que aparece en la figura 26.6.

Se pueden crear varios tipos de pincel. Un `SolidBrush`, el que se usó en el anterior ejemplo, rellena una forma con un color sólido. El uso de un `HatchBrush` permite improvisar la apariencia de sus gráficos. `HatchBrush` usa las enumeraciones `HatchStyle` y `HatchFill` para mostrar los diferentes tipos de patrones. El listado 26.4 dibuja algunas de las variaciones de `HatchBrush` mediante la enumeración `HatchStyle`. Esta enumeración tiene más de 40 miembros, de

modo que merece la pena buscarla en el SDK de .NET Framework. Si alguna vez necesita crear algún tipo de patrón de dibujo, puede encontrar una ayuda vital.

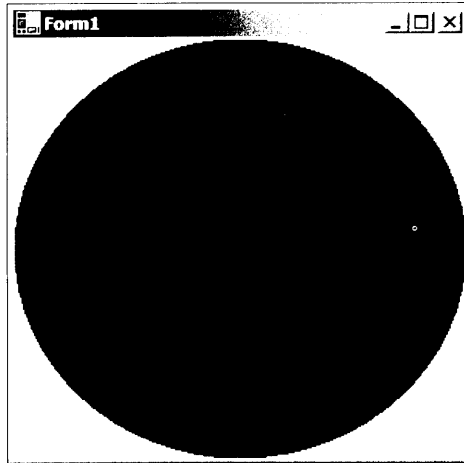


Figure 26.6. Elipse sólida creada con pincel

Listado 26.4. Cómo usar la clase HatchBrush con HatchStyles

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;

    HatchBrush hb =
        new HatchBrush
            (HatchStyle.Plaid,
             Color.AntiqueWhite, Color.Black);
    g.FillEllipse(hb, 30, 30, Width-50, 30);

    HatchBrush hb2 = new HatchBrush
        (HatchStyle.LargeCheckerBoard,
         Color.AntiqueWhite, Color.Black);
    g.FillEllipse(hb2, 30, 80, Width-50, 30);

    HatchBrush hb3 =
        new HatchBrush
            (HatchStyle.DashedHorizontal,
             Color.AntiqueWhite, Color.Black);
    g.FillEllipse(hb3, 30, 130, Width-50, 30);

    HatchBrush hb4 =
        new HatchBrush
            (HatchStyle.ZigZag,
             Color.AntiqueWhite, Color.Black);
    g.FillEllipse(hb4, 30, 180, Width-50, 30);
}
```

Si se ejecuta el código anterior se produce una imagen como la que aparece en la figura 26.7.

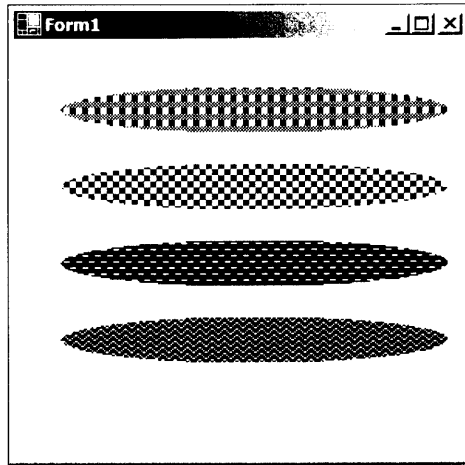


Figura 26.7. HatchBrush con diferentes HatchStyles

La tabla 26.6 describe cada uno de los tipos de lápiz disponibles en la enumeración `PenType` que puede usar con la clase `Brush`. Ya hemos visto `HatchFill` y `SolidColor` en funcionamiento.

Basándose en sus descripciones, probablemente pueda imaginar los otros tipos de pincel sin verlos en funcionamiento.

Tabla 26.6. Enumeración `PenType`

Miembro	Descripción
<code>HatchFill</code>	Especifica un relleno de trama
<code>LinearGradient</code>	Especifica un relleno de degradado lineal
<code>PathGradient</code>	Especifica un relleno de degradado del trazado
<code>SolidColor</code>	Especifica un relleno sólido
<code>TextureFill</code>	Especifica un relleno de textura de mapa de bits

El trabajo con texto y fuentes también requiere emplear un objeto `Brush` junto con un objeto `Graphics`. Para usar texto, se crea una instancia de la clase `Font`, que se incluye en el espacio de nombre `System.Drawing` y establecer las propiedades aspecto, estilo y tamaño de texto y luego se llama al método `DrawString` desde el objeto `Graphics` que contendrá el pincel. El listado 26.5 dibuja la frase `C# is cool` en el formulario en uso y produce algo parecido a la imagen de la figura 26.8.

Listado 26.5. Cómo usar el método DrawString

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    e.Graphics.FillRectangle(
        new SolidBrush(Color.White),
        ClientRectangle);

    g.DrawString("C# is cool", this.Font,
        new SolidBrush(Color.Black), 15, 15);
}
```

NOTA: En .NET, las fuentes que se usan en un objeto Form se heredan de la misma forma. En este ejemplo, la propiedad fuente del formulario recibe el valor 24, de modo que cuando se pasa el valor `this.Font` al método `DrawString`, se usa el actual tamaño de fuente de formulario.

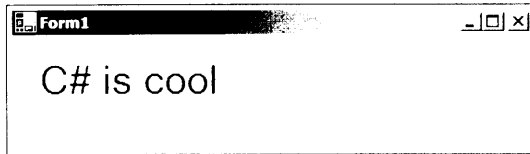


Figura 26.8. Cómo usar el método `DrawString` y la clase `Font` para producir texto

La tabla 26.7 enumera las propiedades disponibles de la clase `Font`. Al establecer o recuperar estas propiedades en sus objetos `Font`, puede controlar completamente el aspecto del texto en la pantalla.

Tabla 26.7. Font Class Properties

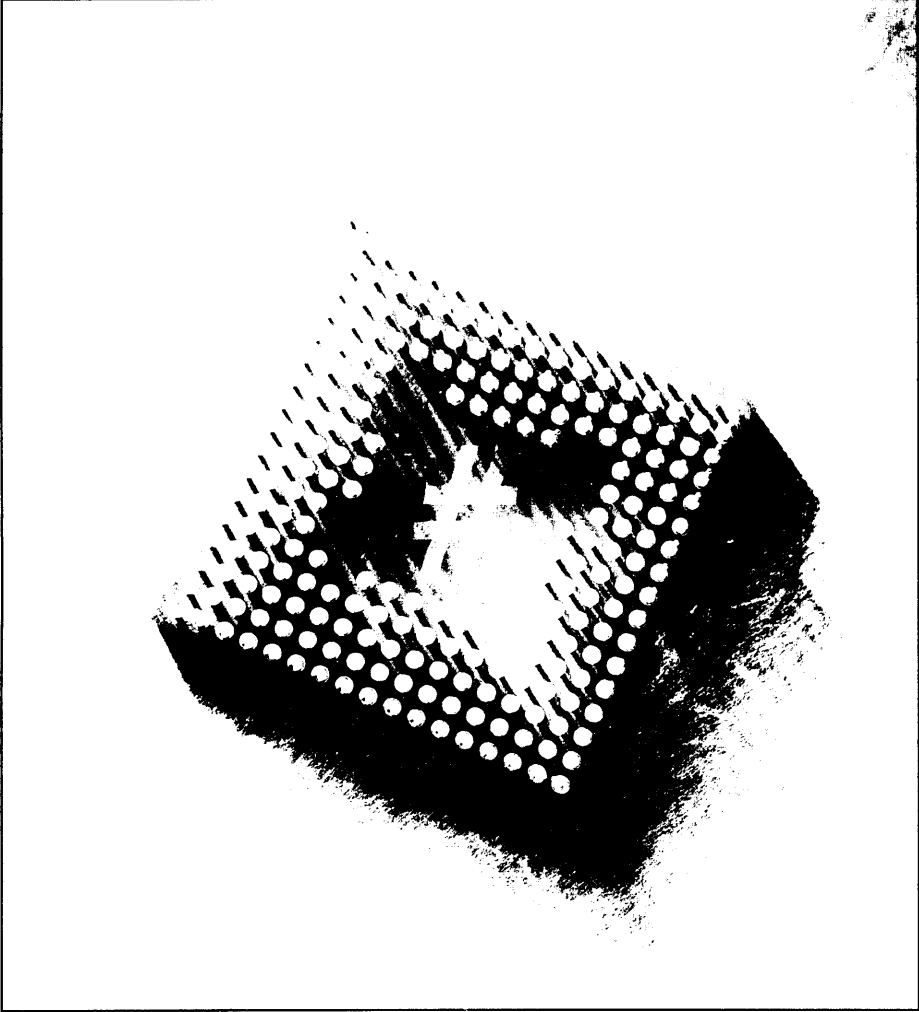
Propiedad	Descripción
<code>Bold</code>	Obtiene un valor que indica si este objeto <code>Font</code> está en negrita
<code>FontFamily</code>	Obtiene el objeto <code>FontFamily</code> asociado a este objeto <code>Font</code>
<code>GdiCharSet</code>	Obtiene un valor de bytes que especifica el conjunto de caracteres GDI que utiliza este objeto <code>Font</code>
<code>GdiVerticalFont</code>	Valor booleano que indica si este objeto <code>Font</code> se deriva de una fuente vertical de GDI
<code>Height</code>	Devuelve la altura de este objeto <code>Font</code>
<code>Italic</code>	Obtiene un valor que indica si este objeto <code>Font</code> está en cursiva

Propiedad	Descripción
Name	Obtiene el nombre del tipo de letra de este objeto <code>Font</code>
Size	Obtiene el tamaño eme de este objeto <code>Font</code> en unidades de diseño
SizeInPoints	Obtiene el tamaño, en puntos, de este objeto <code>Font</code>
Strikeout	Obtiene un valor que indica si este objeto <code>Font</code> especifica una línea horizontal de tachado de la fuente
Style	Obtiene la información de estilo de este objeto <code>Font</code>
Underline	Obtiene un valor que indica si este objeto <code>Font</code> está subrayado
Unit	Obtiene la unidad de medida de este objeto <code>Font</code>

Resumen

GDI+ ofrece una consistente matriz de clases que le permite escribir cualquier tipo de soporte gráfico en sus aplicaciones. Este capítulo presentó una vista general de las funciones de GDI+, pero puede hacer muchas más cosas con los espacios de nombre `System.Drawing` y `System.Drawing.Drawing2D` que pueden explicarse en un sólo capítulo.

Para manipular o crear gráficos usando GDI+, primero debe crear un objeto `Graphics` que le proporciona una superficie en la que pintar. Una vez creado el objeto `Graphics`, puede usar lápices, pinceles, mapas de bits o fuentes para procesar el tipo de imagen deseado.



27 **Cómo construir servicios Web**

Los servicios Web son, probablemente, el rasgo más innovador y apasionante de la iniciativa .NET de Microsoft y probablemente afecte al modo en que las empresas interactúan mediante las aplicaciones de ordenador. Pero, ¿qué es exactamente un servicio Web? A grandes rasgos, un servicio Web es simplemente un componente de servidor que puede ser invocado en Internet. Este componente de servidor normalmente realiza un servicio fundamental de negocios, como la autenticación del usuario, la validación de tarjetas de crédito, el cálculo del precio de un seguro de derivados, la tramitación de una solicitud de compra de acciones o el cálculo del precio de un envío el mismo día. Obviamente, la lista de posibles servicios Web es tan variada como la lista de posibles oportunidades de negocio. Los servicios Web permiten a las aplicaciones invocar servicios de negocios mediante un mecanismo basado en estándares (usando XML y HTTP) y como verá en este capítulo, el modo de realizar esto supone un importante avance en la interoperabilidad de las aplicaciones. La mayoría de los estándares usados para crear servicios Web se realizan con XML. Si no está familiarizado con XML, puede leer una breve introducción en el apéndice de este libro.

En este capítulo se estudiará qué estándares de XML controlan la definición y el uso de los servicios Web. A continuación crearemos un servicio Web mediante Visual Studio .NET. Por último, usaremos este servicio Web en un segundo proyecto de Visual Studio .NET.

Funcionamiento de los servicios Web

Al definir los servicios Web hay que contar con dos tecnologías opcionales: el *mecanismo de descubrimiento* y la *descripción de servicio*. Se pueden evitar estas dos tecnologías usando otros medios de comunicación: por ejemplo, si hay comunicación (llamadas de teléfono) entre los programadores del servicio Web y los programadores del cliente que accederá al servicio Web. El mecanismo de descubrimiento usa un documento XML de servidor para permitir que las aplicaciones cliente detecten la existencia de un servicio Web y encuentren una descripción detallada de ese servicio. Al principio, Microsoft propuso usar DISCO (descubrimiento de servicios Web) para este mecanismo de descubrimiento, pero desde entonces UDDI (Descripción, descubrimiento e integración universales) se ha convertido en el estándar real para los descubrimientos. Puede encontrar más información sobre UDDI en <http://www.uddi.org>. La descripción de servicio describe las entradas y salidas del servicio Web. Las descripciones de servicio usan el estándar Lenguaje de descripción de servicio Web (WSDL), descrito en este mismo capítulo. UDDI (o DISCO, su despreciado predecesor ahora) y WSDL son partes esenciales que pueden usarse para crear detallada documentación sobre cómo invocar un servicio Web. Como estas tecnologías están estandarizadas, las descripciones también pueden ser leídas por aplicaciones. Sin embargo, después de que se ha implementado un cliente, no hay necesidad de usar el mecanismo de descubrimiento ni de hacer referencia a la descripción de servicio. El uso de estas tecnologías no es indispensable para crear o usar servicios Web orientados a una aplicación concreta.

En la invocación real de un servicio Web toman parte tres tecnologías: el *protocolo de conexión*, el *formato de mensaje* y el *mecanismo de invocación*. Sólo los dos primeros están especificados en los servicios Web estándares. El protocolo de conexión es el mecanismo de transporte que se emplea para establecer la comunicación entre el cliente y el servidor. Por lo general, suele ser HTTP, el protocolo de Internet basado en TCP/IP. El formato de mensaje es el formato que se emplea para invocar un servicio Web. Un servicio Web puede ser invocado mediante HTTP puro o con un mensaje XML en un formato específico llamado Protocolo de acceso simple a objetos (SOAP). La tercera tecnología, que controla cómo se llaman a los componentes de servidor, no está especificada por los estándares del servicio Web. Éste es un detalle de implementación que queda a la elección de la persona que implementa el servicio Web. En otras palabras, el programador que crea el servicio Web elige la tecnología usada para llamar al código de negocio en el servidor: un programador de Visual Basic puede usar COM+ para invocar un objeto COM, un programador de Java puede usar RMI para invocar un objeto Java y así sucesivamente.

Las dos partes de un servicio Web pueden describirse como el creador (cliente) y el consumidor (servidor). El creador desarrolla el componente de servidor y

muestra este servicio a quien corresponda. Por ejemplo, una institución financiera desarrolla un sistema de validación de tarjetas de crédito y lo muestra a los vendedores conectados. Mostrar un servicio Web significa publicar la URL que los usuarios necesitan para invocar al servicio Web. El consumidor puede usar el servicio expuesto enviando un mensaje de petición SOAP a la URL publicada. Al recibir una petición SOAP escrita en XML, el componente de servidor tras el servicio Web es invocado en el servidor del creador. Los resultados de esta invocación toman el formato de un mensaje de respuesta SOAP y se envían de vuelta al consumidor del servicio. La figura 27.1 muestra los distintos elementos que toman parte en un servicio Web.

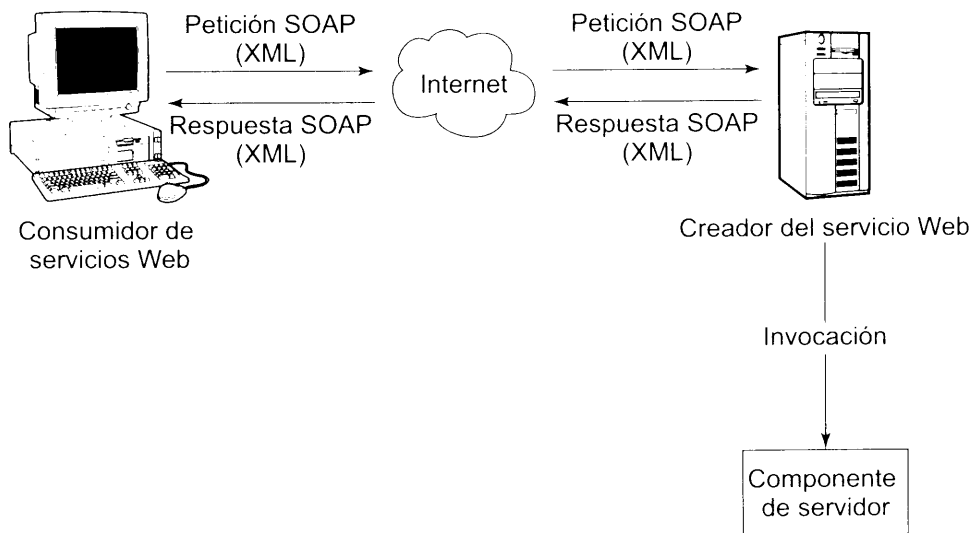


Figura 27.1. Los servicios Web constan dos partes: un consumidor y un creador

Los servicios Web usan un conjunto de estándares para definir cómo se realiza la interacción entre cliente y servidor. Estos estándares definen el mecanismo de transporte que se va a usar y el formato y contenido de la interacción. En el nivel de transporte, se usa el omnipresente protocolo de Internet HTTP. El servidor y el cliente se comunican entre sí mediante mensajes XML. El contenido de estos mensajes también está estandarizado y debe cumplir las reglas de SOAP (más adelante se explicarán estas reglas). La naturaleza de los servicios disponibles en un servicio Web puede ser descrita en un archivo XML cuyo contenido cumpla con las reglas del Lenguaje de descripción de servicio Web (WSDL). Por último, un cliente puede descubrir dinámicamente qué servicios Web están expuestos en un servidor recuperando el archivo XML cuyo contenido cumple con las reglas de DISCO.

Observe que aún no se ha mencionado ninguna tecnología específica de compañías. En ninguna parte se ha presupuesto el sistema operativo del cliente o el servidor, el lenguaje de programación usado para escribir el componente del ser-

vidor o el mecanismo usado para invocar el componente del servidor. Estas elecciones no tienen importancia para un servicio Web. Un cliente escrito en C# que se ejecuta en Windows XP, por ejemplo, puede invocar un servicio Web escrito en Java ejecutándose en Sun Solaris. De hecho, un cliente no tiene modo de saber qué tecnologías se usan para mostrar el servicio Web.

Las implicaciones de los servicios Web son enormes. Microsoft comenzó la programación basada en componentes con la introducción de los controles OLE (OCX), una versión depurada de los innovadores conceptos presentados por los Controles de Visual Basic (VBX). Los OCX se basan en el Modelo de objetos de componentes de Microsoft (COM) y funciona perfectamente. Sin embargo, COM y su contrapartida distribuida, el Modelo de objetos de componentes distribuido de Microsoft (DCOM), tienen un alcance limitado. Aparte de la familia de sistemas operativos Windows, muy pocos sistemas admiten COM/DCOM. Además, aunque la familia Windows tiene una amplia aceptación para aplicaciones de escritorio, en la modalidad de servidor se usan una gran variedad de sistemas operativos. Por ejemplo, muchas variedades del sistema operativo UNIX, como Solaris y Linux, tienen una importante presencia. Los servicios Web eliminan la necesidad de decidir el sistema operativo que se ejecuta en el servidor al integrar dos aplicaciones Web. Al usar servicios Web puede ensamblar aplicaciones Web usando componentes de otros servidores. Por ejemplo, puede usar un servicio Web de una compañía de tarjetas de para validar tarjetas de crédito, un servicio Web de una compañía distribuidora para determinar los gastos de envío y así sucesivamente. Ésta es la esencia y la oferta de los servicios Web: la siguiente generación de programación basada en componentes para la siguiente generación de aplicaciones distribuidas.

Servicios Web y Visual Studio .NET

Si los servicios Web son independientes de la plataforma en la que se ejecutan, ¿qué espera conseguir con esto Microsoft? La respuesta es sencilla: Microsoft ha anunciado públicamente que intentará hacer de Windows el mejor sistema operativo para hospedar servicios Web y de Visual Studio .NET el mejor entorno de desarrollo para crear servicios Web.

Como este libro trata de C#, nuestro estudio de los servicios Web se va a centrar en C# y en su entorno de desarrollo integrado. En la sección práctica de este capítulo podrá juzgar por sí mismo lo flexible y sencillo de usar que es el entorno de desarrollo Visual Studio .NET.

Visual Studio .NET logra un fantástico trabajo simplificando la creación y consumo de servicios Web. La mayor parte de las tareas que odian los programadores (por ejemplo, crear documentos XML) se realizan automáticamente, sin requerir un gran esfuerzo por parte del programador. Todo lo que tiene que hacer al programar es declarar su intención de mostrar un fragmento de código como un

servicio Web y la herramienta se encarga de casi todo el trabajo. De hecho, la herramienta hace un trabajo tan bueno que nunca verá XML al construir un servicio Web y un consumidor de servicio Web. De hecho, esto es exactamente lo que va a hacer en este capítulo. Sin embargo, antes de empezar, observe el concepto que hace posible toda esta automatización: la programación basada en atributos.

La programación basada en atributos es un potente concepto que permite a Visual Studio .NET automatizar una gran cantidad de pesadas tareas de programación (como crear un documento WSDL para un servicio Web). Simplemente tiene que marcar un fragmento de código, como una clase o un método, de un modo especial para indicar lo que quiere hacer con él. Como resultado, Visual Studio .NET genera los archivos necesarios para implementar su función. Un breve ejemplo servirá para mostrar cómo funciona al transformar una clase en un servicio Web.

El listado 27.1 muestra cómo puede implementar un sencillo juego. De acuerdo, este juego no es muy entretenido (el jugador siempre pierde), pero este capítulo trata sobre la programación de servicios Web, no sobre programación de juegos. Observe los elementos necesarios para convertir este fragmento de código en un servicio Web y lo que Visual Studio .NET genera durante este proceso. El principal objetivo de este ejercicio es que se haga una idea de la cantidad de trabajo necesario para convertir un fragmento de código completo en un servicio Web usando Visual Studio .NET.

Listado 27.1. Un sencillo juego

```
namespace MyFirstWebService
{
    public class GameWS
    {
        // Ejemplo de un sencillo juego
        // El juego de ejemplo devuelve la cadena "Sorry, you
        // lose!"
        // Para probar este juego, pulse F5
        public string Play(string opponentName)
        {
            return "Sorry " + opponentName + ", you lose!";
        }
    }
}
```

El primer paso para convertir este fragmento de código en un servicio Web es guardar el código en un nuevo archivo llamado `GameWS.asmx`. A continuación, realice estos cuatro pasos:

1. Agregue un título que indique tres cosas: que el archivo contiene un servicio Web, el lenguaje que usa y la clase que contiene la implementación:

```
<@ WebService Language="c#"
Class="MyFirstWebService.GameWS" >
```


2. Agregue una directiva `System.Web.Service` inmediatamente debajo del título del servicio Web:

```
using System.Web.Services;
```

3. Marque la clase como servicio Web y escoja el espacio de nombres XML asociado al servicio Web:

```
[WebService(Namespace="http://www.boutquin.com/GameWS/")]  
public class GameWS : System.Web.Services.WebService
```

4. Marque los métodos de la clase como accesibles desde la Web:

```
[WebMethod]  
public string Play(string opponentName)
```

El listado 27.2 muestra el resultado final. También se han cambiado los comentarios para reflejar los cambios al código original realizados.

Listado 27.2. Sencillo juego expuesto como servicio Web

```
< @ WebService Language="c#" Class="MyFirstWebService.GameWS"  
>  
  
using System.Web.Services;  
  
namespace MyFirstWebService  
{  
    [WebService(Namespace="http://www.boutquin.com/GameWS/")]  
    public class GameWS : System.Web.Services.WebService  
    {  
        // EJEMPLO DE SERVICIO WEB  
        // El metodo Play() devuelve la cadena "Sorry, you lose!"  
        // Para comprobar este servicio web, pulse F5  
        [WebMethod]  
        public string Play(string opponentName)  
        {  
            return "Sorry " + opponentName + ", you lose!";  
        }  
    }  
}
```

Al construir un proyecto de servicios Web en Visual Studio, se crea automáticamente un archivo de descripción de servicio que describe el servicio Web. Este archivo es un dialecto XML llamado Lenguaje de descripción de servicio Web (WSDL). Un archivo WSDL tiene este aspecto:

```
<?xml version="1.0" encoding="UTF-8"?>  
<methods href='http://www22.brinkster.com/boutquin/  
GameWS.asmx'>  
    <method name='Play' href='Play'>  
        <request>
```

```

        <param dt='string' >opponentName</param>
    </request>
    <response dt='string' />
</method>
</methods>

```

WSDL describe las funciones que están expuestas (el formato que se muestra es en realidad una simplificación del formato real, pero los conceptos siguen siendo los mismos).

Puede llamar al servicio mediante una URL (en este caso, `www22.brinkster.com/boutquin/GameWS.aspx/Play?opponentName=Pierre`) o enviar un mensaje XML con el formato apropiado a la URL (mediante una instrucción `post` o `get` HTTP). Este mensaje XML puede ser un mensaje SOAP como el siguiente:

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Play xmlns="http://www.boutquin.com/GameWS/">
      <opponentName>Pierre</opponentName>
    </Play>
  </soap:Body>
</soap:Envelope>

```

Al invocar el servicio, mediante la URL o enviando un mensaje SOAP, se produce una respuesta XML como la siguiente:

```

<?xml version="1.0" encoding="UTF-8"?>
  <string xmlns="http://www.boutquin.com/GameWS/">Sorry Pierre,
  you lose!</string>

```

Las siguientes secciones examinan las bases de SOAP y WSDL, tras lo cual pasaremos a estudiar los detalles de la creación e invocación de servicios Web.

Lenguaje de descripción de servicio Web (WSDL)

WSDL es el vocabulario XML usado para describir servicios Web. Esta descripción incluye información sobre cómo acceder a ellos. .NET Framework se ocupa de generar estos servicios Web por nosotros, de modo que no necesitamos saber demasiado sobre WSDL para usar servicios Web. Para ver el aspecto de un WSDL para un servicio puede añadir `?wsdl` a su URL y ver el resultado en un navegador compatible con XML: por ejemplo, `www22.brinkster.com/boutquin/GameWS.aspx?wsdl`.

Esta sección hace una breve descripción de este vocabulario XML. WSDL usa un espacio de nombre por defecto. `xmlns="http://schemas.xmlsoap.org/wsdl/"`. Usa un elemento raíz llamado `definitions` y contiene varias secciones. Una de estas secciones es la sección de servicio donde, evidentemente, se describen los servicios. El siguiente fragmento es el esqueleto de un archivo WSDL. Un archivo WSDL real usa varias declaraciones de espacio de nombre, que aquí omitimos por simplicidad:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns='http://schemas.xmlsoap.org/wsdl/'>

  <service name="GameWS" >

    <!--El servicio se describe aqui -->

  </service>

</definitions>
```

El primer atributo de una descripción de servicio define la posición desde la que se puede llamar al servicio. Esto se describe en el elemento `address` dentro de un elemento `port`. (Elemento y atributo son términos de XML, como puede ver en el apéndice.) El siguiente ejemplo muestra el atributo `binding`:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns='http://schemas.xmlsoap.org/wsdl/'>

  <service name="GameWS" >

    <port name="GameWSSoap" binding="s0:GameWSSoap">
      <soap:address
location="http://www22.brinkster.com/boutquin/GameWS.asmx" />
    </port>

  </service>

</definitions>
```

Si un servicio Web está expuesto mediante una instrucción `post` o `get` de HTTP, su posición se almacena en un elemento `http:address` element:

```
  <port name="GameWSHttpPost" binding="s0:GameWSHttpPost">
<http:address location="http://www22.brinkster.com/boutquin/
GameWS.asmx" />
  </port>
```

A continuación, debe definir los parámetros de entrada y salida. Puede hacerlo mediante los elementos de mensaje. En este elemento, se concede un nombre a cada mensaje; y en un mensaje, se describe cada parámetro (nombre y tipo de datos):

```

<message name = "PlayInput" >
  <part name="opponentName" element='xsd:string' />
  </message>
<message name = "PlayOutput" >
  <part name='Result' type='xsd:string' />
</message>

```

A continuación puede asociar los mensajes con el punto final de destino usando un elemento `portType`. En `portType`, se usa el nombre que asignó a este punto final de destino en el elemento `port`; y por cada uno de los servicios Web, crea una etiqueta de operación que contiene un elemento `input` y `output` que el mensaje usa como atributo:

```

<portType name="GameWSSoap">
  <operation name="Play">
    <input message="PlayInput" />
    <output message="PlayOutput" />
  </operation>
</portType>

```

Por último, el elemento `binding` describe los detalles específicos importantes del mecanismo de transporte usado. Un enlace SOAP, por ejemplo, debe especificar la acción de SOAP:

```

<binding name="GameWSSoap" type="s0:GameWSSoap">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <operation name="Play">
    <soap:operation soapAction="http://www.boutquin.com/
GameWS/Play" style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>

```

Cómo usar el Protocolo de acceso simple a objetos (SOAP)

SOAP es el dialecto de XML usado por los servicios Web. Especifica qué acción de servidor quiere invocar para pasar la información (es decir, los

parámetros) al servicio Web. SOAP también especifica cómo se devuelve la información desde el servicio Web (valores de devolución y excepciones).

Los mensajes SOAP siguen un formato estándar: un sobre externo que identifica el mensaje como un mensaje SOAP, un cuerpo que contiene la principal carga útil y un título opcional que ofrece información adicional sobre el mensaje. Puede usar el título para pasar información que no es una parte propiamente dicha de la invocación del servidor. Por ejemplo, puede pasar la fecha y hora de la solicitud o usar autenticación en este título. El cuerpo contiene un elemento cuyo nombre concuerda con el nombre del método del servidor que se está invocando.

Los elementos secundarios de este elemento tienen nombres que concuerdan con los parámetros:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>

    <!-- La información adicional se coloca aquí -->

  </soap:Header>
  <soap:Body>

    <MethodName>
      <Param1Name>value1</Param1Name>
      <Param2Name>value2</Param2Name>
      <!-- etc. -->
    </MethodName>

  </soap:Body>
</soap:Envelope>
```

El mismo formato se usa para enviar una respuesta. Los nombres de parámetro (en este caso, `Param1Name` y `Param2Name`) de la respuesta son, por supuesto, los parámetros de salida: o "devolución" cuando el método sólo devuelve un valor (por ejemplo `<return>value</return>`).

Cuando se produce algún error, la información del error se envía de vuelta en una sección de errores. La sección de errores se encuentra en el cuerpo SOAP:

```
<soap:Fault>

  <faultcode>x00</faultcode>
  <faultstring>description</faultstring>
  <runcode>Yes</runcode>

</soap:Fault>
```

Cómo crear servicios Web con Visual Studio .NET

La operación de crear (y, en la siguiente sección, acceder a) un servicio Web usando Visual Studio .NET es aparentemente simple. Ni siquiera será consciente de estar usando XML.

El servicio Web que crearemos en esta sección simplemente recuperará y mostrará una lista de libros. Esto simula la función de catálogo de una página Web comercial, aunque en el mundo real probablemente desearía introducir categorías para evitar que se devolviera una enorme lista de libros. Dado que el objetivo de este capítulo es mostrar los elementos necesarios para construir un servicio Web, el aspecto comercial ha sido simplificado.

El ejemplo que creamos aquí usa una tabla y un procedimiento almacenado. El código para crearlos aparece en el siguiente ejemplo (quizás también quiera colocar algunos datos de muestra en la tabla Books):

```
CREATE TABLE [Books] (
    [ISBN] [char] (14) NOT NULL ,
    [Title] [varchar] (150) NOT NULL ,
    [Price] [money] NOT NULL
)
GO

ALTER TABLE [dbo].[Books] WITH NOCHECK ADD
    CONSTRAINT [PK_Books] PRIMARY KEY CLUSTERED
    (
        [ISBN]
    ) ON [PRIMARY]
GO
CREATE PROCEDURE [pc_getBooks]
AS
SELECT
    [ISBN],
    [Title],
    [Price]
FROM [Books]
GO
```

Ya está preparado para construir un sencillo servicio Web que devuelve una lista de libros usando el procedimiento almacenado:

1. Abra Visual Studio .NET y seleccione Archivo>Nuevo proyecto.
2. Seleccione Servicio Web ASP.NET como el tipo de proyecto en el cuadro de diálogo Nuevo proyecto.
3. Escriba `BookSeller` como nombre del proyecto (véase la figura 27.2).

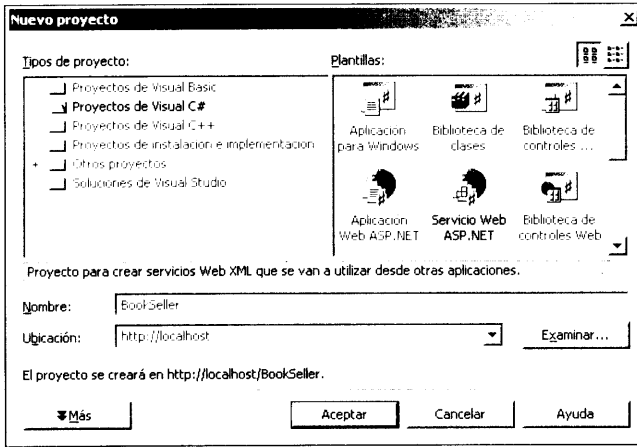


Figura 27.2. Un servicio Web es un tipo de proyecto

4. Renombre el servicio Web de `Service1.asmx` a `Books.asmx`.
5. Cambie a Vista de código haciendo clic en la ficha `Books.asmx.cs` y cambie todas las apariciones de `Service1` a `Books` (figura 27.3).

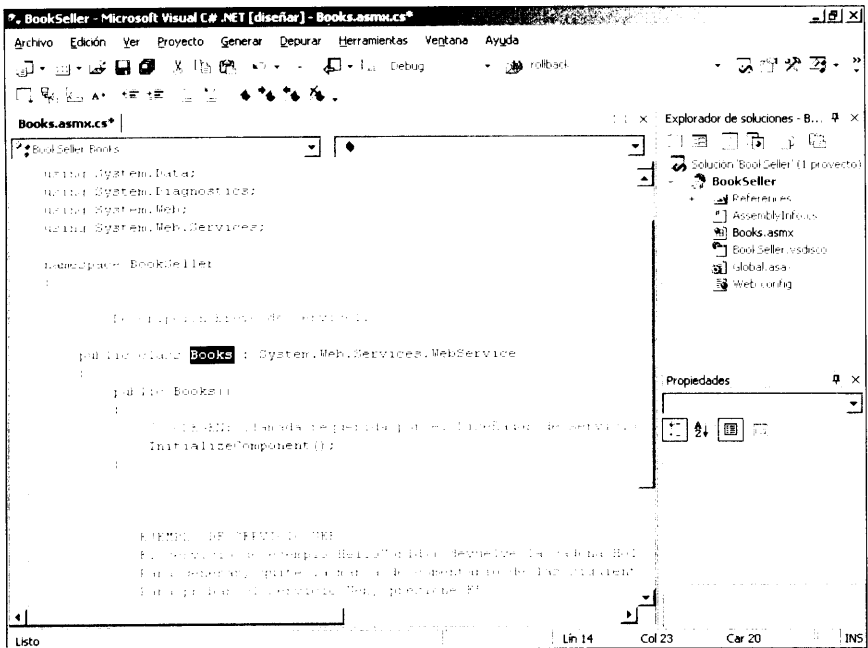


Figura 27.3. Puede cambiar el nombre del servicio por defecto a uno más descriptivo

6. Cambie la sección `using` para que contenga lo siguiente:

```
using System;
using System.Web.Services;
```

```
using System.Data;
using System.Data.OleDb;
```

7. Agregue los siguientes métodos y propiedad a la clase Books:

```
private static string oleDbConnectionString
{
    get
    {
        // NOTA: Usar la cuenta sa en aplicaciones de
        // producción
        // es, obviamente, una practica muy desaconsejable.
        // Además, dejar la
        // contraseña para la cuenta sa en blanco es igualmente
        // inadmisibile.
        return "Provider=SQLOLEDB.1;"
            +"User ID=sa;Initial Catalog=WebService_1_0_0;Data
            Source=localhost;";
    }
}

// EJEMPLO DE SERVICIO WEB
[WebMethod]
public DataSet getList()
{
    // Establece las cadenas de instrucción SQL
    string strSQLSelect = "[pc_getBooks]";

    // Crea objetos OleDb
    OleDbConnection databaseConnection = new
    OleDbConnection(oleDbConnectionString);
    OleDbCommand selectCommand = new
    OleDbCommand(strSQLSelect, databaseConnection);
    OleDbDataAdapter dsCmd = new OleDbDataAdapter();
    DataSet resultDataSet = new DataSet();

    // Estamos trabajando con un procedimiento almacenado (es
    // decir, NO con un instrucción SQL) selectCommand.CommandType
    = CommandType.StoredProcedure;

    try
    {
        // Establezca la conexión a la base de datos
        databaseConnection.Open();

        // Ejecute SQL Command
        dsCmd.SelectCommand = selectCommand;
        int numRows = dsCmd.Fill(resultDataSet, "Books");
    }
    catch (Exception e)
    {
```



```

        Console.WriteLine("***** Caught an exception:\n{0}",
e.Message);
    }
    finally
    {
        databaseConnection.Close();
    }

    return resultDataSet;
}

```

La propiedad `oleDbConnectionString` contiene la cadena de conexión a la base de datos SQL Server. En código de producción, puede usar una cuenta con derechos de seguridad apropiados (y una contraseña) en lugar de la todopoderosa cuenta "sa". El método `getList()` abre una conexión a la base de datos SQL Server y recupera un conjunto de datos que contiene la lista de libros invocando al comando `pc_getBooks`. Como puede ver, usar ADO.NET es muy sencillo.

8. ¡Eso es todo! También puede agregar a la clase una declaración de espacio de nombre, como muestra el siguiente ejemplo:

```

[WebService(Namespace="http://microsoft.com/webservices/")]
public class Books : System.Web.Services.WebService

```

Los espacios de nombre son un modo de evitar la duplicidad de nombres. Se usa un único prefijo para distinguir entre servicios Web con el mismo nombre. Por lo general, se usan direcciones URL como base para estos nombres únicos. Esto está en concordancia con el modo en que se usan las URL en los espacios de nombre XML, como se describe en el apéndice.

9. Guarde el proyecto y pulse **F5** para comprobar el servicio Web.

Ahora que ha creado un servicio Web, intente crear una aplicación cliente que use este servicio Web.

Cómo usar Visual Studio .NET para acceder a un servicio Web

El siguiente ejemplo muestra los pasos necesarios para crear una aplicación de servicio Web en C#:

1. Abra Visual Studio .NET y seleccione Archivo>Nuevo proyecto.
2. Seleccione Aplicación Web ASP.NET como el tipo de proyecto.
3. Déle al proyecto el nombre `BookRetailer`.

4. Seleccione **Proyecto>Agregar referencia Web**.
5. Haga clic en **Referencias Web en Servidor local** para que Visual Studio .NET detecte automáticamente los servicios Web disponibles en el servidor local.
6. Seleccione `http://localhost/BookSeller/BookSeller.vsdisco` y haga clic en **Agregar referencia**.

Ha importado la información necesaria para llamar a este servicio Web.

7. En el modo **Diseño**, agregue un control `Label` a la parte superior de la página y un control `DataGrid` bajo de la etiqueta y cambie a la vista de código.
8. Agregue una declaración `using` a la página ASP.NET: esto indica al compilador que va a usar el código del servicio Web.

```
using BookRetailer.localhost;
```

9. Agregue el siguiente código al método `Page_Init`. En este ejemplo, se establece el texto `Label` y luego se rellena el `DataGrid` de un modo rápido y sencillo (esta página no ganará ningún premio por su apariencia):

```
private void Page_Init(object sender, EventArgs e)
{
    //
    // CODEGEN: ASP.NET Windows Form Designer necesita esta
    // llamada.
    //
    InitializeComponent();

    // Agregado por PGB
    Label1.Text = "Available Books";

    Books books = new BookRetailer.localhost.Books();
    DataSet bookList = books.getList();
    DataGrid1.DataSource =
    bookList.Tables["Books"].DefaultView;
    DataGrid1.DataBind();
    // Fin de la adición PGB
}
```

10. Guarde y ejecute el proyecto (utilice **F5** como tecla de método abreviado). Aparecerá una pantalla como la que se muestra en la figura 27.4.

Reflexionemos sobre lo que hemos logrado. Hemos creado un servicio Web (que puede ejecutarse en un servidor conectado a Internet).

En esta sección, creó una página Web (que puede ejecutarse en un servidor diferente) que usa este servicio Web para recuperar una lista de libros del primer servidor.

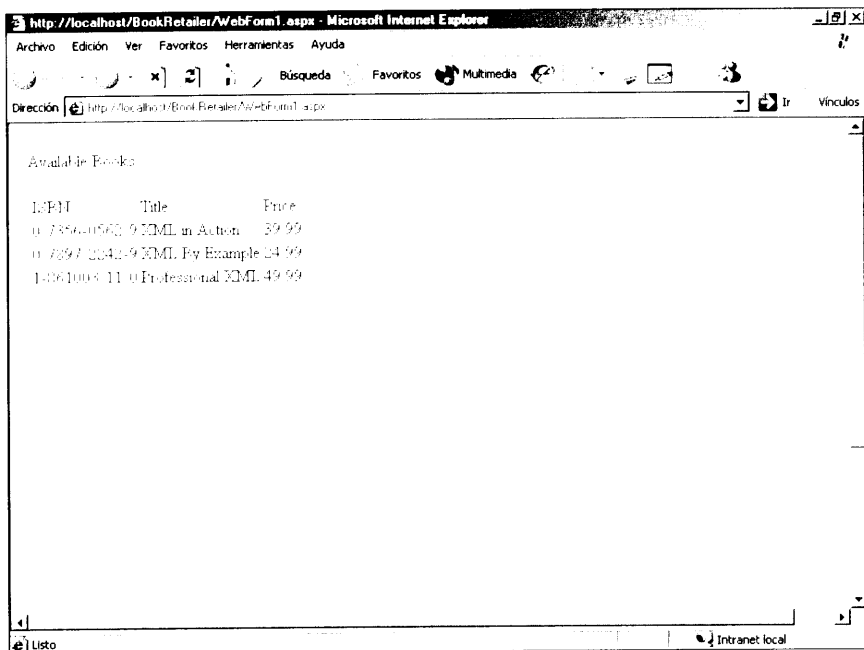
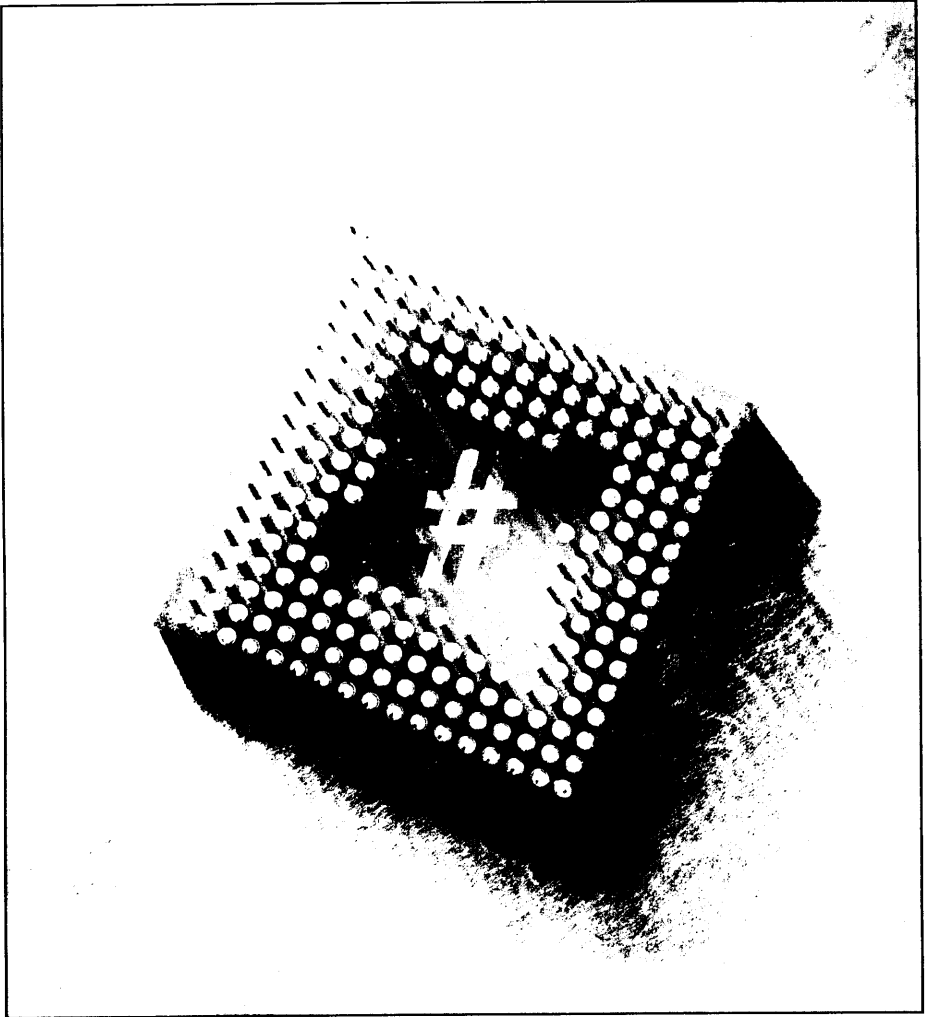


Figura 27.4. Un servicio Web en funcionamiento, tras haber recuperado una lista de libros del proveedor de servicio Web

Resumen

En este capítulo ha estudiado los estándares XML que hay tras los servicios Web. Vimos las dos tecnologías opcionales que toman parte en la definición de servicios Web: UDDI para el mecanismo de descubrimiento y WSDL para la descripción de servicio. También estudiamos el formato de mensaje que se usa durante la invocación real de un servicio Web: SOAP. Creamos un sencillo servicio Web usando Visual Studio. Por último, creamos un segundo proyecto que usaba el servicio Web que había construido anteriormente.



28 Cómo usar C# en ASP.NET

La llegada de Internet y las intranets corporativas han llevado al desarrollo de las aplicaciones distribuidas. Una aplicación distribuida puede acceder a la información de diferentes fuentes de datos que pueden estar dispersas en varias localizaciones geográficas.

Visual Studio .NET lleva las aplicaciones distribuidas a nuevos niveles al permitirle usar servicios Web y clientes de servicios Web.

Los servicios Web de ASP.NET son servicios basados en XML que están expuestos en Internet y a los que tienen acceso otros servicios Web y los clientes de servicios Web. Un servicio Web muestra métodos Web a los que tienen acceso los clientes del servicio Web, que implementan la funcionalidad del servicio Web.

Antes de Visual Studio .NET, la programación ASP se realizaba usando VBScript. Sin embargo, con la llegada de Visual Studio .NET, puede usar dos lenguajes para la programación ASP: Visual C# y Visual Basic .NET.

Visual C# permite escribir código ASP.NET para servicios y aplicaciones Web. Por último, aprenderá a implementar la aplicación Web mediante Visual Studio .NET.

En este capítulo, aprenderá a usar C# para crear aplicaciones ASP.NET. Empezaremos creando un servicio Web en Visual C#. Tras ello, crearemos un cliente de servicio Web en C#, que en realidad es una aplicación Web que usa el servicio Web.

Cómo crear un servicio Web

En Visual Studio .NET, los servicios Web se usan para integrar las aplicaciones remotas con sus actuales soluciones comerciales. En líneas generales, los servicios Web presentan dos ventajas:

- Puede usar un servicio Web de otra organización para crear una aplicación personalizada para su empresa. Por ejemplo, puede usar el servicio de autenticación de Microsoft Passport para incluir esa autenticación en su página Web. Puede beneficiarse de este proyecto porque no necesita crear la infraestructura necesaria para implementar autenticación personalizada en su página Web. Además, su página podrá atender a un mayor número de visitantes porque todos los usuarios registrados con el servicio Passport (lo que incluye a todos los usuarios de Hotmail y MSN) podrán conectarse a su página Web.
- También puede utilizar los servicios Web para comunicarse con sus compañeros de empresa. Por citar un ejemplo, imagine un vendedor de libros que tenga libros de varios editores. Si cada editor puede hospedar un servicio Web que proporciona información sobre los últimos libros que ha publicado, el vendedor puede desarrollar un cliente de servicio Web que se conecte a estos servicios Web y recupere datos de estos libros.

A continuación vamos a crear un servicio Web que usa una base de datos. Por tanto, en el primer paso aprenderá a crear una base de datos para el servicio Web. A continuación, aprenderá a usar la base de datos y a crear el servicio Web.

NOTA: Al crear el servicio Web de este ejemplo, nos concentraremos sólo en las tareas necesarias para crearlo. Para aprender más sobre los conceptos que intervienen en la implementación de servicios Web, consulte un capítulo anterior.

Cómo crear una base de datos para un servicio Web

Los servicios o aplicaciones Web suelen emplear una base de datos para almacenar sus datos pertenecientes a la aplicación. En un entorno empresarial, las bases de datos, como Microsoft SQL Server y Oracle, están muy preparadas para gestionar datos. En el servicio Web que creará en este capítulo se usa una base de datos SQL Server. Antes de crear la base de datos de SQL Server y las tablas para el servicio Web, revise los conceptos más importantes del Sistema de gestión de bases de datos relacionales (RDBMS).

Conceptos del sistema de gestión de bases de datos relacionales

Un sistema de gestión de bases de datos relacionales (RDBMS) es adecuado para las soluciones de negocios empresariales. Un RDBMS, como Microsoft SQL Server, Oracle o DB2, permite la creación, actualización y administración de bases de datos relacionales. Una *base de datos relacional* es una colección de datos organizados en forma de tablas. Las aplicaciones pueden acceder a los datos de las tablas usando instrucciones de lenguaje de consulta estructurado (SQL). En un RDBMS, puede acceder a datos y reorganizarlos sin reorganizar toda la base de datos. Esto mejora considerablemente el rendimiento de la base de datos. Además, puede aplicar fácilmente reglas de negocios, validaciones y restricciones a los datos de las tablas de un RDBMS. Las reglas empresariales y las validaciones garantizan la integridad de los datos. Por ejemplo, cuando se inscribe a un pasajero en un vuelo con el sistema de reservas de una compañía aérea, debe existir el número de vuelo especificado. Puede determinar el número del vuelo estableciendo una regla de negocio y usándola al reservar el billete.

Tipos de datos de SQL Server

Los datos de una base de datos se almacena en tablas, como filas y columnas. Las columnas de una tabla almacenan información clasificada, como el número de identificación del producto, su nombre y el número de unidades disponibles. Las filas de una tabla almacenan registros específicos. Cada columna de una tabla tiene un tipo de datos específico. La tabla 28.1 describe algunos de los tipos de datos SQL Server más comunes.

Tabla 28.1. SQL Data Types

Tipo de dato	Descripción
Integer	Se usa para almacenar números enteros.
Float	Se usa para almacenar números decimales.
char(n)	Se usa para almacenar datos de caracteres que pueden ser alfabéticos, numéricos, caracteres especiales, como #, % o \$, o una combinación de letras y caracteres. Un tipo de dato <code>char</code> almacena un solo carácter. Para almacenar más de un carácter se usa <code>char(n)</code> , donde <code>n</code> hace referencia al número de caracteres que se quieren almacenar.
varchar(n)	Se usa para almacenar datos de caracteres, donde <code>n</code> hace referencia al número de caracteres que se quieren almacenar. Un tipo de dato <code>varchar</code> es diferente de un tipo de datos <code>char</code> porque la memoria asignada

Tipo de dato	Descripción
	a un tipo de datos <code>varchar</code> depende del tamaño de los datos, a diferencia de los tipos de datos <code>char</code> , en los que la memoria asignada está predefinida.
Datetime	Se usa para almacenar datos de fecha y hora.
Money	Se usa para almacenar datos monetarios que requieren gran precisión.

TRUCO: Cada tabla debe tener al menos una columna que identifica unívocamente una fila (a la que llamaremos *registro*) de la tabla. Esta columna es la clave primaria de la tabla. Por ejemplo, la columna `ProductID` de una tabla `Products` identifica cada fila unívocamente y por tanto es la clave primaria. No puede haber dos valores iguales en una clave primaria.

Cómo crear bases de datos y tablas

En Microsoft SQL Server, se pueden crear bases de datos, tablas, procedimientos almacenados y consultas usando Transact-SQL (T-SQL).

TRUCO: También puede usar SQL Server Enterprise Manager para crear una base de datos y tablas. SQL Server Enterprise Manager proporciona una interfaz gráfica para realizar los mismos pasos que realizamos mediante las instrucciones T-SQL.

Para crear una base de datos o una tabla mediante T-SQL, se emplea la instrucción `Create`. Por ejemplo, para crear una base de datos `Sales`, escriba el siguiente código en la ventana del analizador de consultas:

```
Create Database Sales
```

Tras crear la base de datos, puede agregarle tablas. Agregue la tabla `Products` a la base de datos `Sales` mediante la siguiente sintaxis:

```
Create Table Products
(
ProductID VarChar (4) Primary Key,
ProductName VarChar (20),
UnitPrice Integer,
QtyAvailable Integer
)
```

Cómo recuperar datos

Puede recuperar información almacenada en tablas usando la instrucción `Select`. Por ejemplo, para recuperar todos los registros de la tabla `Products` de la base de datos `Sales`, use las siguientes instrucciones:

```
Use Sales
Select * From Products
```

Cómo insertar, actualizar y eliminar datos

Puede agregar, actualizar y eliminar datos de una base de datos de SQL Server siguiendo los pasos indicados a continuación:

- **Agregar un registro:** Para agregar una nueva fila a una tabla SQL Server, use la instrucción `Insert`. Por ejemplo, para agregar un nuevo registro a la tabla `Products`, se usa la siguiente instrucción:

```
Insert Into Products (ProductID, ProductName, UnitPrice,
QtyAvailable)
Values ('P001', 'Baby Food', 2.5, 12000)
```

ADVERTENCIA: Para que la inserción tenga éxito, los valores de la columna deben proporcionarse en el mismo orden que las columnas de la tabla. Además, si el tipo de datos de una columna es `char`, `varchar` o `datetime`, debe especificar los valores entre comillas.

- **Modificar un registro:** Para modificar un registro de una tabla SQL Server, use la instrucción `Update`:

```
Update Products
Set UnitPrice=75
Where ProductID="P010"
```

El anterior código actualiza el precio por unidad del registro cuya identificación de producto es `P010` hasta `75`.

- **Eliminar un registro:** Para eliminar un registro de una tabla, use la instrucción `Delete`. Por ejemplo, para eliminar un registro de la tabla `Products` con la identificación de producto `P011`, puede especificar la siguiente instrucción:

```
Delete From Products where ProductID="P011"
```

Cómo usar procedimientos almacenados

Un *procedimiento almacenado* es un conjunto de instrucciones SQL usadas para realizar tareas específicas. Un procedimiento almacenado se aloja en un Servidor SQL y puede ser ejecutado por cualquier usuario que tenga los permisos

adecuados. Puede crear un procedimiento almacenado usando la instrucción `Create Procedure`. Use el siguiente código para crear un procedimiento almacenado que acepte `ProductID` como parámetro y devuelva el precio por unidad del registro que concuerde con el `ProductID`:

```
Create Procedure ProductPrice (@id char (4))
As
Select UnitPrice
From Products Where ProductID=@id
Return
```

El procedimiento requiere un parámetro, `@id`, en el momento de la ejecución.

Los procedimientos almacenados son particularmente útiles cuando se necesita realizar varias tareas consecutivas en una base de datos. Por ejemplo, cuando quiere cancelar la reserva de un pasajero, querrá calcular la tarifa que debe devolverse al cliente y borrar su reserva de la tabla de reservas. Al mismo tiempo, también deberá actualizar el estado de los otros pasajeros que puedan estar en lista de espera para entrar en la lista de pasajeros. En lugar de especificar consultas de SQL cada vez que quiera cancelar una reserva, puede usar un procedimiento almacenado para cancelar la reserva de un pasajero.

ADVERTENCIA: Cada procedimiento almacenado debe terminar una instrucción `Return`.

Para ejecutar el procedimiento anterior para que muestre el precio del producto con la identificación ID P010, use el siguiente código:

```
Execute ProductPrice "P010"
```

Cómo crear la estructura de la base de datos

En este capítulo, necesitamos crear una base de datos `Sales` para nuestro servicio Web. Tras crear la base de datos `Sales`, agregue una tabla `Products` a la base de datos. Para crear una base de datos `Sales` y añadirla la tabla `Products`, siga los siguientes pasos:

1. Seleccione Inicio>Programas>Microsoft SQL Server>Query Analyzer. Se abrirá el cuadro de diálogo `Connect to SQL Server`.
2. En el cuadro de diálogo `Connect to SQL Server`, escriba el nombre del servidor SQL en el cuadro de texto `SQL Server`, especifique un nombre de contacto en el cuadro de texto `Login Name` y especifique la contraseña para el nombre de contacto en el cuadro de texto `Password`.
3. Haga clic en **OK** para conectarse al SQL Server y abra el editor de consultas.

4. En el editor de consultas, introduzca las siguientes instrucciones para crear la base de datos Sales y agregar la tabla Products a la base de datos:

```
Create database Sales
GO
Use Sales
Create Table Products
(
ProductID VarChar (4) Primary Key,
ProductName VarChar (20),
UnitPrice Integer,
QtyAvailable Integer
)
GO
```

5. Seleccione Query>Execute para ejecutar la consulta.

Tras ejecutar la consulta, la estructura de la base de datos está creada. Ahora estamos listos para crear el servicio Web (la primera de las aplicaciones ASP.NET que creará este capítulo). El servicio Web que se crea en este capítulo agrega registros a la tabla Products de la base de datos Sales que creamos anteriormente en esta misma sección.

Cómo usar la plantilla Servicio Web ASP.NET

Debe usar la plantilla de proyecto Servicio Web ASP.NET para crear un servicio Web. Este proyecto sirve como plantilla basada en la Web para crear los componentes del servicio Web. Para crear un servicio Web, siga estos pasos:

1. Seleccione Archivo>Nuevo>Proyecto para abrir el cuadro de diálogo Nuevo proyecto.

TRUCO: Puede pulsar simultáneamente las teclas **Control-Mayús-N** para abrir el cuadro de diálogo Nuevo Proyecto.

2. Seleccione Proyectos de Visual C# de la lista Tipos de proyecto.
3. Seleccione Servicio Web ASP.NET del cuadro de plantillas a la derecha del cuadro de diálogo
4. En el cuadro Nombre, escriba `OrdersWebService`. En el cuadro Ubicación, introduzca el nombre de su servidor Web como `http://<nombredeservidor>`. Haga clic en **Aceptar**.

TRUCO: También puede escribir `localhost` en el cuadro Ubicación si el servidor Web está instalado en el equipo en el que está creando el servicio Web.

NOTA: Puede que tenga que esperar bastante tiempo mientras Visual Studio .NET crea el servicio Web.

Una vez que Visual Studio .NET ha creado el servicio Web, puede configurarlo para que gestione datos en el servidor. Esto lo haremos en la siguiente sección.

Cómo agregar controles de datos al servicio Web

Debe agregar controles de datos al servicio Web para permitir la comunicación con la base de datos Sales que creó en el anterior apartado. Para comunicarse con la base de datos, debe agregar los siguientes controles a su servicio Web:

- `SqlDataAdapter`: El control `SqlDataAdapter` se usa para transferir datos entre fuentes de datos.
- `SqlConnection` y `SqlDataAdapter`: Los controles `SqlDataAdapter` y `SqlConnection` se usan para conectar con la origen de datos.
- `SqlCommand`: Tras establecer una conexión con el origen de datos, use el control `OleDbCommand` para acceder a los datos.
- `DataSet`: Los datos se almacenan en un control `DataSet`.

Los pasos para agregar el control `SqlDataAdapter` son los siguientes:

1. Seleccione **Ver>Cuadro de herramientas** para abrir el cuadro de herramientas.
2. En el cuadro de herramientas, haga clic en **Datos** para activar la ficha **Datos**.
3. Arrastre el control `SqlDataAdapter` desde el cuadro de herramientas hasta el Diseñador de componentes.
4. Al arrastrar el control `SqlDataAdapter` desde el cuadro de herramientas, se inicia el asistente para la configuración del adaptador de datos. En la pantalla de bienvenida del asistente, haga clic en **Siguiente**.
5. Aparecerá el cuadro de diálogo **Elegir la conexión de datos** del asistente, como se ilustra en la figura 28.1. Haga clic en **Nueva conexión** para crear una nueva conexión usando el controlador `OleDbDataAdapter`.
6. Se abrirá el cuadro de diálogo **Propiedades de vínculo de datos**. Por defecto, este cuadro de diálogo tiene seleccionada la ficha **Conexión**. Especifique el nombre del servidor SQL en el cuadro de diálogo **Seleccione o escriba un nombre de servidor**.

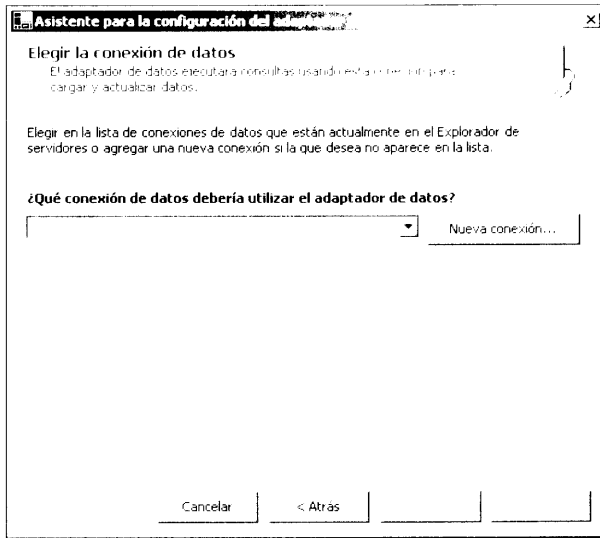


Figura 28.1. El cuadro de diálogo Propiedades de vínculos de datos

7. Seleccione el nombre de usuario y la contraseña para conectarse al servidor SQL Server y seleccione la base de datos **Sales** en la lista desplegable **Seleccione la base de datos en el servidor**. La figura 28.2 muestra la pantalla Propiedades de vínculos de datos completa. Haga clic en **Aceptar**.

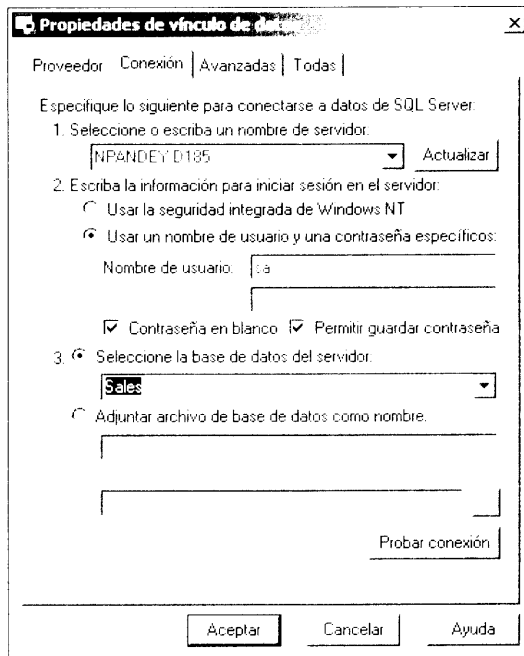


Figura 28.2. Conecte con el origen de datos usando el cuadro de diálogo Propiedades de vínculo de datos

8. El adaptador de datos que ha configurado aparecerá en el cuadro de diálogo **Elegir la conexión de datos**. Haga clic en **Siguiente** para continuar.
9. Se abrirá el cuadro de diálogo **Elegir la conexión de datos**. Para usar una consulta SQL para recuperar datos de la base de datos, mantenga la opción por defecto. Usar instrucciones de SQL y haga clic en **Siguiente**.
10. Se abrirá el cuadro de diálogo **Generar las instrucciones SQL**. En este cuadro de diálogo, escriba la consulta `Select * from Products` y haga clic en **Siguiente**.
11. Se abrirá el cuadro de diálogo **Ver resultados del asistente**. Este cuadro de diálogo resume las opciones que ha seleccionado en los anteriores cuadros de diálogo del asistente. Haga clic en **Finalizar** para dar por finalizado el asistente de configuración del adaptador de datos.

Tras completar el asistente de configuración del adaptador de datos, el control `SqlDataAdapter` está configurado para su aplicación. Como se puede apreciar en la figura 28.3, los controles `sqlDataAdapter1` y `sqlConnection1` se han agregado a su aplicación.

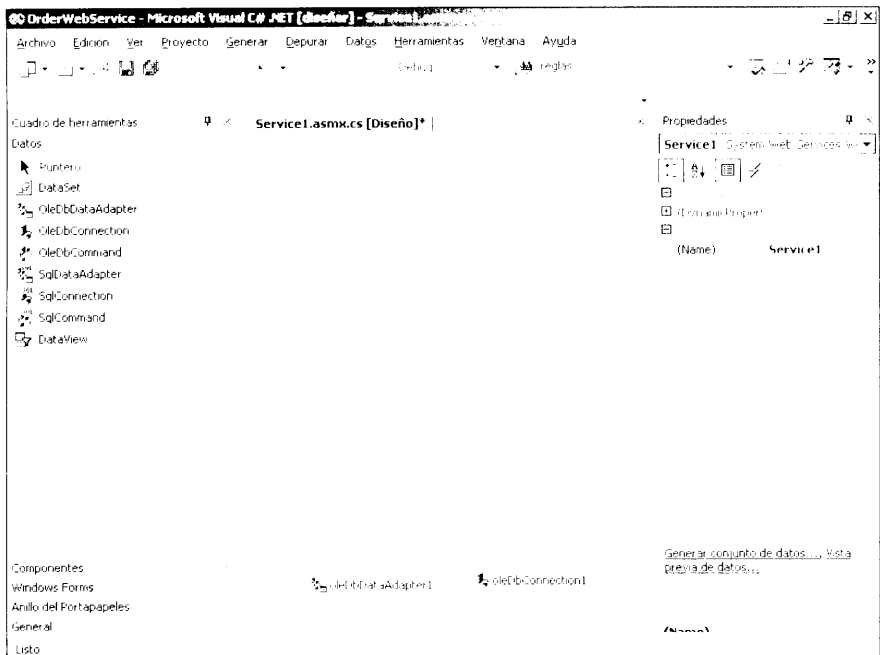


Figura 28.3. Los controles `sqlDataAdapter1` y `sqlConnection1` se han agregado a su aplicación

A continuación, agregue el control `SqlCommand` al servicio Web. El control `SqlCommand` se usa para especificar comandos que necesitan ser ejecutados en

el origen de datos. Siga estos pasos para agregar un control `SqlCommand` a su aplicación:

1. Arrastre el control `SqlCommand` desde el cuadro de herramientas al diseñador de componentes.
2. Abra la ventana **Propiedades** y seleccione `sqlConnection1` para la propiedad de conexión del control `SqlCommand`.

A continuación, debe generar un conjunto de datos para almacenar los datos que recuperen los controles de datos:

1. Para generar el conjunto de datos, seleccione el control `SqlCommand1` que agregó en los pasos anteriores y seleccione la opción de menú **Datos > Generar conjunto de datos**.
2. Se abrirá el cuadro de diálogo **Generar conjunto de datos**. En este cuadro de diálogo, ya está seleccionada la tabla `Products` de la base de datos. Seleccione la casilla **Agregar este conjunto de datos al diseñador** y haga clic en **Aceptar** para generar el conjunto de datos y agregarlo al diseñador de componentes.

Los cuatro controles que ha agregado al servicio Web ahora son visibles para el diseñador de componentes. Ahora necesita codificar los métodos del servicio Web, como verá en la siguiente sección.

Cómo codificar el servicio Web

Tras agregar los controles de datos al servicio Web, debe codificar los métodos del servicio Web. En este capítulo le enseñamos a codificar un método para agregar productos a la base de datos `Products` mediante el servicio Web.

Antes de proceder a codificar los métodos del servicio Web, añádales una descripción y cambie el espacio de nombres por defecto del servicio Web. La descripción y el espacio de nombres del servicio Web permiten al desarrollador del cliente del servicio Web entender el modo de empleo del servicio Web. Para agregar una descripción y cambiar el espacio de nombres por defecto asociado al servicio Web, siga estos pasos:

1. Haga doble clic en el diseñador de componentes para abrir el editor de código.
2. En el editor de código, localice la instrucción `public class Servicio1`.
3. Agregue el siguiente código antes de la instrucción que localizó en el segundo paso:

```
[WebService(Namespace="http://ServiceUPL.com/products/",  
Description="Use the servicio Web to add products to the  
Sales database.")]
```


Tras introducir la línea de código anterior, el espacio de nombres del servicio Web es `http://ServiceURL.com/products/` y se ha agregado una descripción al servicio Web.

A continuación, escriba el código para agregar productos a la tabla `Products`. Este código necesita escribirse justo debajo de la declaración del servicio Web. El código del método `AddProduct`, que agrega detalles del producto a la tabla `Products`, es el siguiente:

```
[WebMethod(Description="Specify the product ID, product name,
unit price, and quantity to add it to the Sales catalog")]
public string AddProduct(string PID, string ProductName, int
Price, int Qty)
{
    try
    {
        ProductName=ProductName.Trim();
        if (Price<0)
            return "Please specify a valid value for price";
        if (Qty<0)
            return "Please specify a valid value for quantity";
        sqlConnection1.Open();
        sqlCommand1.CommandText="INSERT INTO Products(ProductID,
            ProductName, UnitPrice, QtyAvailable) VALUES ('" + PID +
            "\", '" + ProductName + "\", '" + Price + "\", '" + Qty
+
            "'");
        sqlCommand1.ExecuteNonQuery();
        sqlConnection1.Close();
        return "Record updated successfully";
    }
    catch(Exception e)
    {
        return e.Message;
    }
}
```

El código anterior usa el control `sqlConnection1` para agregar un registro a la tabla `Products`. Al agregar registros a un servicio Web, se usan los valores que la función `AddProduct` proporciona como parámetros.

Mientras codifica el método `AddProduct` del servicio Web, puede codificar otros métodos para recuperar detalles de producto de la base de datos `Products` o realizar otras acciones personalizadas. Para probar el servicio Web tras agregarle los métodos, siga estos pasos:

1. Seleccione **Generar>Generar solución** para generar el servicio Web.
2. Para empezar a depurar el servicio Web, seleccione la opción de menú **Depurar>Iniciar**.

El servicio Web se abre en Internet Explorer. El primer cuadro de diálogo muestra los métodos que codificó para su servicio Web.

Para comprobar el método `AddProduct`, siga estos pasos:

1. Haga clic en **AddProduct** en el cuadro de diálogo Servicio Web Service1.
2. Se abrirá el cuadro de diálogo `AddProduct`, como muestra la figura 28.4. En este cuadro de diálogo, puede invocar la función `AddProduct` tras proporcionar los parámetros requeridos para probar su resultado.

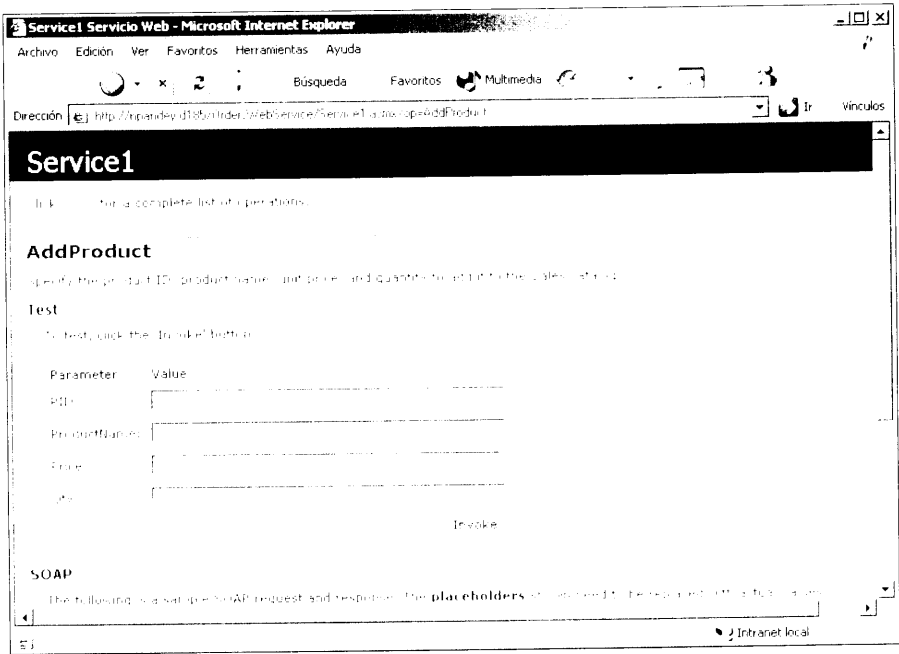


Figura 28.4. Especifique los parámetros para la función `AddProduct` para comprobarla

3. Escriba `P002`, `PDA`, `100` y `200` como los parámetros para el `PID`, `ProductName`, `Price` y `Qty` respectivamente y haga clic en **Invoke** para probar su resultado.
4. Cuando se logra agregar el registro al servicio Web, se consigue un resultado como el que se muestra en la figura 28.5.

Tras probar su servicio Web, puede crear un cliente de servicio Web para acceder al servicio Web.

Cómo crear un cliente de servicio Web

Un cliente de servicio Web de ASP.NET es una aplicación Web compuesta por uno o más WebForms. En esta sección, vamos a crear una aplicación Web ASP.NET que accede al servicio Web creado en la anterior sección.

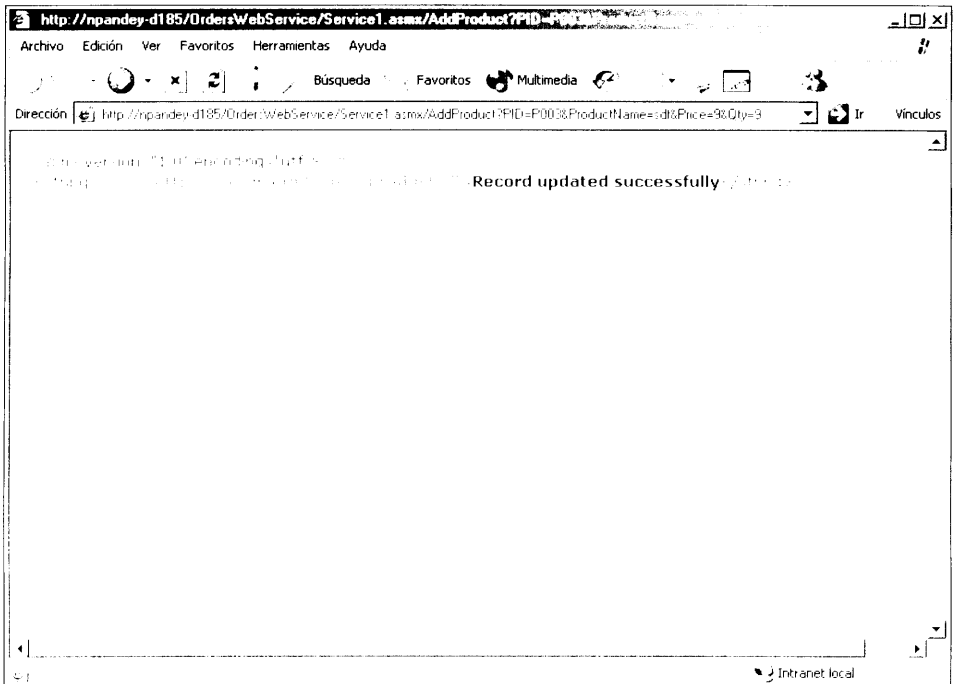


Figura 28.5. Este resultado se consigue cuando se logra agregar un registro a un servicio Web

Para crear un cliente de servicio Web sólo tiene que seguir estos pasos:

1. Cree un nuevo proyecto de Aplicación Web ASP.NET.
2. Agregue una referencia Web a la aplicación Web.
3. Implemente los métodos del servicio Web en la aplicación Web.

La siguiente sección estudia detenidamente cada uno de estos pasos.

Cómo crear un nuevo proyecto de aplicación Web ASP.NET

Para crear un proyecto de aplicación Web, siga esto pasos:

1. Seleccione la opción de menú Archivo>Nuevo>Proyecto. Se abrirá el cuadro de diálogo Nuevo proyecto.
2. En el cuadro de diálogo Nuevo proyecto, seleccione la plantilla de proyecto Aplicación ASP.NET de la lista Proyectos de Visual C#.
3. Escriba `OrdersWebApplication` como nombre del proyecto y haga clic en **Aceptar** para crear la aplicación Web.

Cómo agregar una referencia Web

Tras crear el servicio Web, necesita agregar una referencia Web al servicio Web que creó en la anterior sección. Al agregar una referencia Web, el cliente de servicio Web descarga la descripción del servicio Web y crea una clase de proxy para el servicio Web. Una clase de proxy incluye funciones para los métodos del servicio Web. Para agregar una referencia Web al servicio Web, siga estos pasos:

1. Seleccione el formulario Web `WebForm1.aspx` y a continuación seleccione la opción de menú **Proyecto>Agregar referencia Web**. Se abrirá la opción de menú **Agregar referencia Web**.
2. En el cuadro de diálogo **Agregar referencia Web**, haga clic en **Referencias Web** en el vínculo **Servidor Web local**. Los servicios Web disponibles en el servidor Web local aparecerán en la lista **Referencias disponibles**, como muestra la figura 28.6.
3. En el cuadro de diálogo **Agregar referencia Web**, seleccione el vínculo al servicio Web `OrdersWebService` y haga clic en **Agregar referencia** para agregar una referencia al servicio Web.

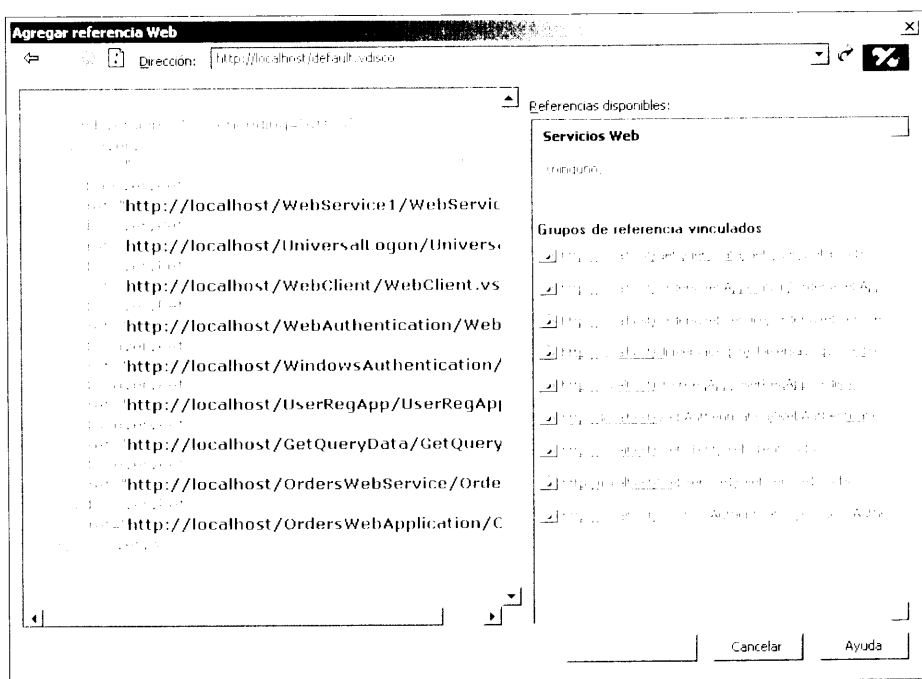


Figura 28.6. Los servicios Web disponibles se muestran en el cuadro de diálogo **Agregar referencia Web**

La referencia al servicio Web que se agrega aparece en el explorador de soluciones. Por defecto, recibe el nombre `localhost`. Puede cambiar el nombre del

servicio Web por un nombre de su elección. En la figura 28.7, recibe el nombre OSI.

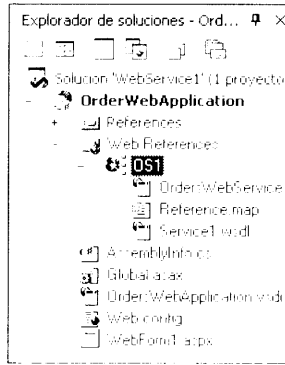


Figura 28.7. Las referencias a servicios Web aparecen en el Explorador de soluciones

Cómo implementar los métodos del servicio Web

Para implementar los métodos del servicio Web, necesita diseñar un WebForm. El WebForm acepta información que debe añadirse a la base de datos Sales. El WebForm diseñado para la aplicación Web aparece en la figura 28.8.

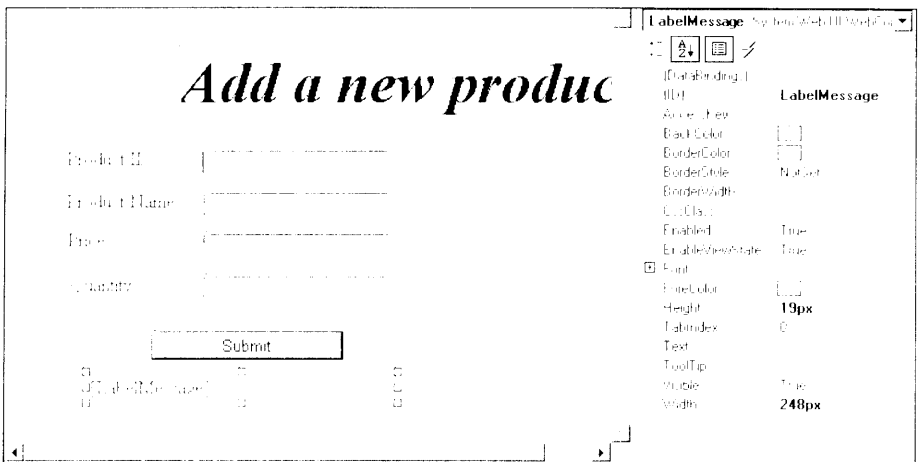


Figura 28.8. Diseño de un WebForm que acepta información de los usuarios

Para diseñar el formulario, puede agregar los controles que aparecen en la tabla 28.2.

Tras diseñar el WebForm, escriba el código para el evento `Click` del botón **Submit**. Cuando el usuario haga clic en el botón **Submit**, el código del evento `Click` hará lo siguiente:

1. Creará una instancia del servicio Web.
2. Llamará al método `AddProduct` del servicio Web, pasando como parámetros al WebForm los valores introducidos por el usuario.

Tabla 28.2. Controles de formularios Web

Tipo de control	Propiedades cambiadas
Label	Se ha agregado seis etiquetas: el encabezamiento principal de la página, Product ID, Product Name, Price, Quantity y Message, respectivamente. Se ha cambiado el Id. de la última etiqueta a LabelMessage y se ha limitado su texto.
TextBox	ID=PID
TextBox	ID=ProductName
TextBox	ID=Price
TextBox	ID=Quantity
Button	ID=Submit Text=Submit

3. Mostrará el valor devuelto desde el método `AddProduct` en la etiqueta `LabelMessage`.

A continuación se muestra el código del evento `Click` del botón **Submit** que cumple las tareas indicadas anteriormente:

TRUCO: Para introducir el código del evento `Click` del botón **Submit**, haga doble clic en el botón **Submit** en la vista diseño.

```
private void Submit_Click(object sender, System.EventArgs e)
{
    OS1.Service1 Web1=new OS1.Service1();
    LabelMessage.Text=Web1.AddProduct(PID.Text,
    ProductName.Text,
    Convert.ToInt32(Price.Text), Convert.ToInt32
    (Quantity.Text));
}
```

4. Seleccione **Depurar>Inicio** para ejecutar el servicio Web. El resultado de la aplicación Web aparece en la figura 28.9.

Especifique valores en los cuadros de texto **Product ID**, **Product Name**, **Price** y **Quantity** y haga clic en **Submit**. La aplicación Web agregará la información

requerida a la base de datos Sales y aparecerá un mensaje en la etiqueta LabelMessage, como muestra la figura 28.10.

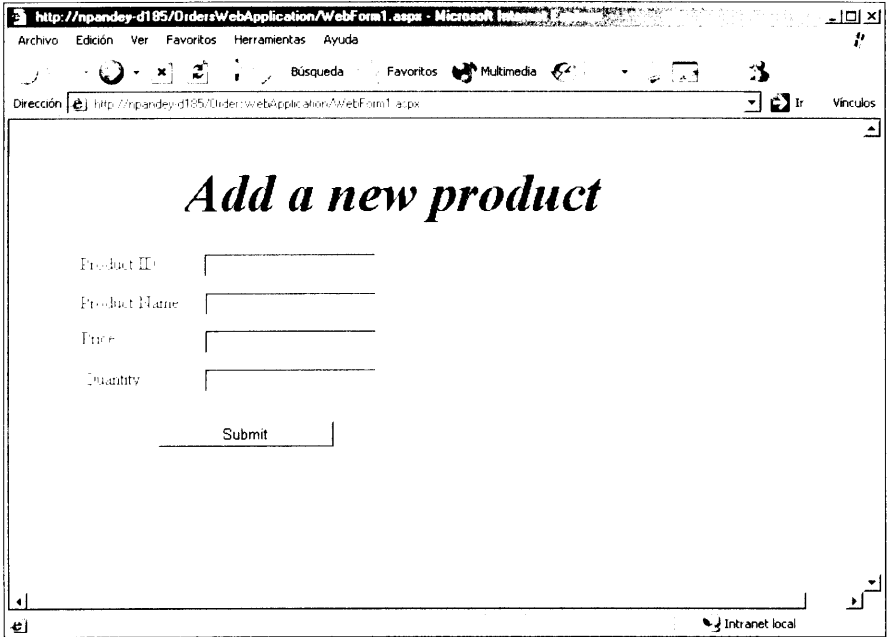


Figura 28.9. Este WebForm muestra el resultado de la aplicación Web

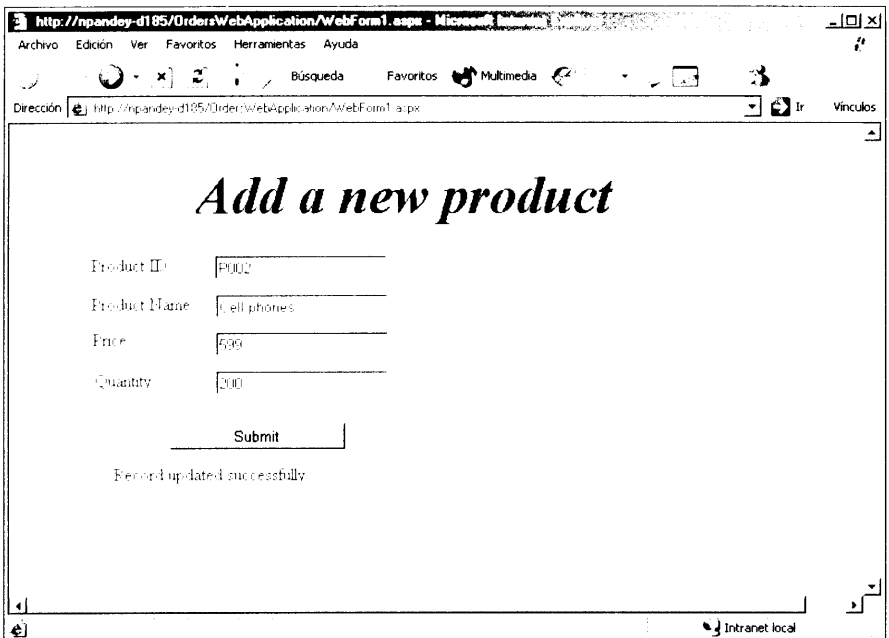


Figura 28.10. Puede agregar un registro a la base de datos Sales mediante la aplicación Web

Cómo implementar la aplicación

El último paso para integrar varias aplicaciones de Visual Studio .NET es implementar la aplicación que ha construido. Para implementar la aplicación, puede usar uno o más proyectos proporcionados por Visual Studio .NET. Las siguientes secciones estudian la implementación de proyectos proporcionada por Visual Studio .NET y describen los pasos necesarios para implementar una aplicación ASP.NET.

Implementación de proyectos en Visual Studio .NET

Visual Studio .NET proporciona cuatro tipos de proyectos de instalación: Proyectos Cab, Proyectos de módulos de combinación, Proyectos de instalación y Proyectos de programas de instalación Web. La siguiente lista describe estos tipos de proyecto:

- **Proyectos Cab:** Puede crear un archivo contenedor (CAB) para empaquetar controles ActiveX. Cuando empaqueta un control ActiveX en un archivo CAB, el control puede descargarse desde Internet.
- **Proyectos de módulos de combinación:** Puede crear un módulo de combinación para los componentes de aplicación que quiera incluir en varias aplicaciones. Por ejemplo, un control de servidor que haya creado puede ser empaquetado como un proyecto de módulo de combinación que puede incluir en proyectos de instalación para sus aplicaciones de Windows.
- **Proyectos de instalación:** Puede usar un proyecto de instalación para empaquetar aplicaciones de Windows. Un proyecto de aplicación crea un archivo Microsoft Installer (MSI) que puede usarse para instalar una aplicación en el equipo de destino.
- **Proyectos de programas de instalación Web:** Puede usar la plantilla Proyecto de programa de instalación Web para una aplicación Web ASP.NET o para un servicio.

La siguiente sección describe cómo crear un proyecto de implementación para una aplicación Web ASP.NET.

Cómo usar un proyecto de implementación para implementar una aplicación

Los pasos necesarios para implementar una aplicación ASP.NET son los siguientes:

1. En el Explorador de soluciones, haga clic en la solución `OrderWebApplication`. Desde el menú emergente que aparece, seleccione `Agregar>Nuevo proyecto`.
2. En el cuadro de diálogo `Nuevo proyecto` que se abre, seleccione `Proyectos de instalación e implementación` en el cuadro `Tipos de proyecto`.
3. En el cuadro `Plantillas`, seleccione `Proyecto de programa de instalación Web`. Escriba como nombre del proyecto `WebSetupDeploy` y haga clic en `Aceptar`. El nuevo proyecto se agregará a la ventana del explorador de soluciones. Por defecto, se abre el Editor del sistema de archivos para el proyecto de implementación. El Editor del sistema de archivos le ayuda a agregar archivos al proyecto de implementación.
4. En el Editor del sistema de archivos (el cuadro a la izquierda), seleccione `Carpeta de aplicación Web`.
5. Para agregar el resultado de `OrderWebApplication` al proyecto de implementación, seleccione `Acción>Agregar>Resultados de proyectos`. Se abrirá el cuadro de diálogo `Agregar grupo de resultados del proyecto`.
6. En el cuadro de diálogo `Agregar grupo de resultados del proyecto`, seleccione `OrderWebApplication` en la lista desplegable `Proyectos`, si es necesario.
7. Mantenga presionada la tecla **Control** y seleccione `Resultado principal` y `Archivos de contenido` de la lista de archivos. Las opciones seleccionadas aparecen en la figura 28.11.

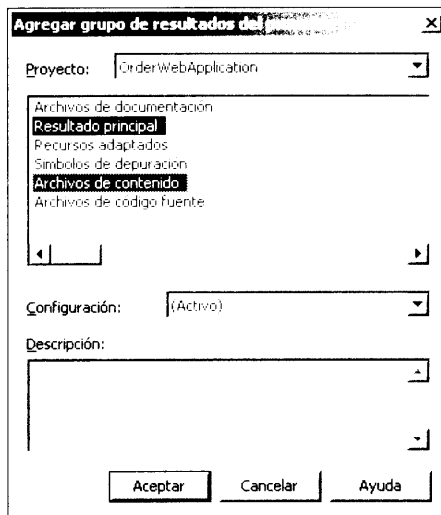


Figura 28.11. El cuadro de diálogo `Agregar grupo de resultados del proyecto` muestra las opciones que ha seleccionado

8. Haga clic en **Aceptar** para cerrar el cuadro de diálogo **Agregar grupo de resultados del proyecto**.
9. En la ventana del editor del sistema de archivos, haga clic con el botón derecho del ratón en la **Carpeta de aplicación Web**. En el menú desplegable, seleccione la ventana **Propiedades** para abrir la ventana **Propiedades**.
10. Escriba **OrderWebApplication** como nombre del directorio virtual en la propiedad **VirtualDirectory** y cierre la ventana **Propiedades**.
11. Seleccione la opción de menú **Generar>Generar solución**.

Se ha creado un archivo Microsoft Installer (MSI) para su proyecto. Puede hacer doble clic en el archivo para instalar su aplicación en un equipo.

Cómo implementar un proyecto usando la opción Copiar proyecto

Si no quiere tomarse la molestia de crear un proyecto de implementación para su aplicación y usar el proyecto para implementarla, también puede usar la opción **Copiar proyecto** de Visual Studio .NET para copiar todos los archivos de su aplicación en la carpeta de destino.

La opción **Copiar proyecto** es útil cuando necesita implementar un proyecto en una red de comunicaciones. Sin embargo, esta opción no funciona cuando quiere implementar su aplicación en un equipo remoto que no tiene acceso a la red de comunicaciones.

Para usar la opción **Copiar proyecto**, siga estos pasos:

1. Abra el proyecto que quiere copiar.
2. Seleccione la opción de menú **Proyecto>Copiar proyecto**. Se abrirá el cuadro de diálogo **Copiar proyecto**.
3. Especifique en el cuadro de diálogo la **Carpeta de proyecto de destino**, la ruta de destino en la que quiere copiar el proyecto.
4. Seleccione los archivos del proyecto que quiere copiar y haga clic en **Aceptar**. Su proyecto se copiará en el destino que especificó en el Paso 3. Tras copiar el proyecto, puede ejecutarlo desde la nueva ubicación.

Resumen

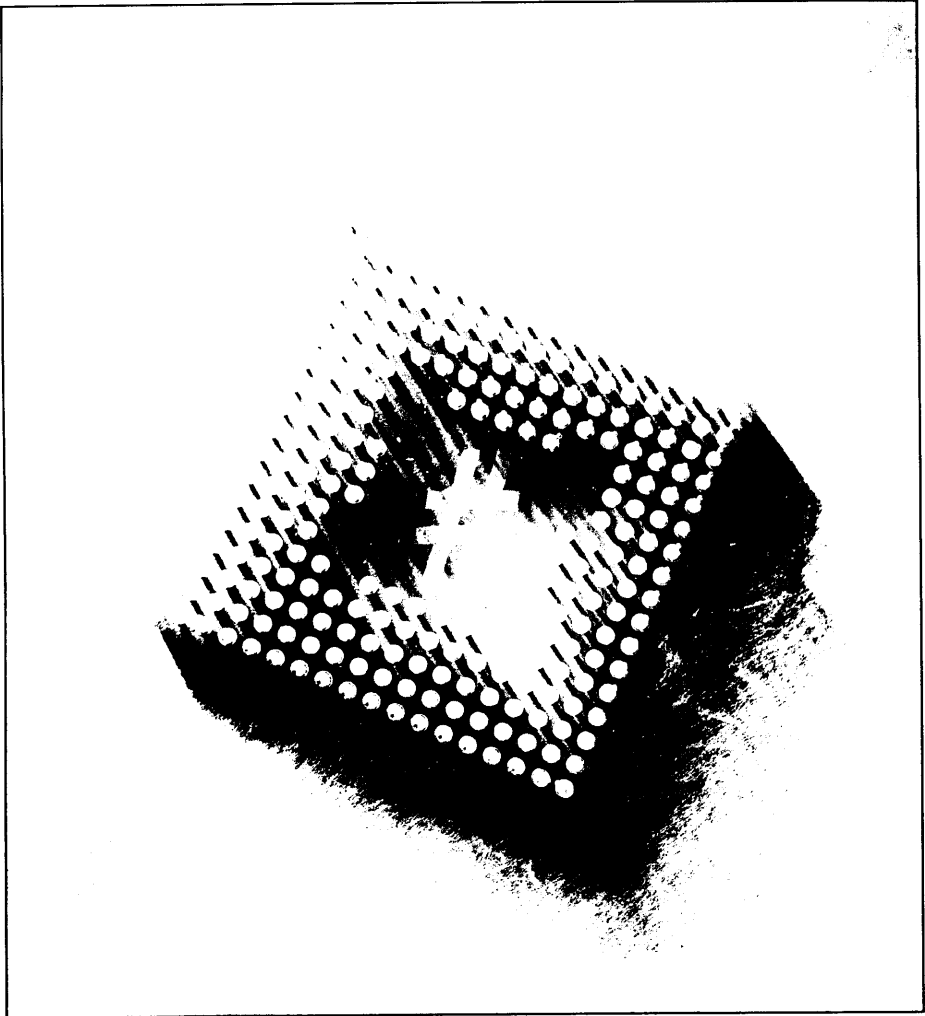
El papel de C# en ASP.NET es muy útil para los servicios y aplicaciones Web. Puede crear un servicio Web mediante la sintaxis del lenguaje C# en ASP.NET.

También puede crear una aplicación Web que acceda al servicio Web y aproveche su funcionalidad.

Cuando crea una aplicación en ASP.NET, primero diseña la base de datos que debe usar la aplicación. A continuación, crea un nuevo proyecto en ASP.NET mediante la Aplicación Web de ASP.NET o las plantillas de proyecto servicio Web ASP.NET.

Tras crear un proyecto en ASP.NET, use los controles de datos proporcionados por Visual Studio .NET para conectar con un origen de datos y utilizarlo.

Si ha creado un servicio Web, también puede crear una aplicación Web para utilizar los métodos Web proporcionados por el servicio Web.



29 **Cómo construir controles personalizados**

En este capítulo aprenderá a crear controles personalizados. Estos controles pueden tomar la forma de un componente visible que puede agregarse a un formulario o simplemente ser un objeto de clase encapsulado en una DLL. En cada instancia, estos controles aprovechan su conocimiento de la reutilización de código cuando empieza a empaquetar y distribuir la funcionalidad como un control que puede ser usado una y otra vez.

Biblioteca de control de Windows

Una biblioteca de control de Windows es un proyecto que permite la creación de componentes iguales que los mostrados en el cuadro de herramientas de Visual Studio. Hace años, estos controles eran siempre objetos COM o controles ActiveX. Ahora puede agregar componentes COM al cuadro de herramientas o puede crear un control que tenga la forma de una biblioteca de vínculos dinámicos (DLL).

Las anteriores versiones de Visual Studio tenían asistentes que ayudaban en la creación de controles y Visual Studio .NET no es diferente. Las tareas asociadas con la creación de propiedades, métodos y campos se han simplificado gracias a la inclusión de los asistentes.

Propiedades

Las propiedades se usan principalmente para establecer y recuperar parámetros de control. Cuando a las propiedades se le conceden los accesos de lectura y de escritura, se componen de dos funciones llamadas `get` y `set`. Dado que una propiedad debe llamar a un método cuando se le asignan o se recuperan de ella valores, puede realizar cálculos u otras funciones útiles inmediatamente. En este aspecto, son muy diferentes de los campos. Un ejemplo de propiedad puede ser la propiedad `BackColor`. Al usar esta propiedad, puede especificar el color que se va a usar como color de fondo de un control o comprobar cuál es el actual color de fondo.

Las propiedades se implementan fácilmente en un proyecto usando el sencillo asistente de Visual Studio .NET. En la ventana Vista de clases, haga clic con el botón derecho del ratón en la clase del control y escoja `Agregar > Agregar clase`. El asistente, que aparece en la figura 29.1, le permite establecer el tipo de acceso, el tipo de propiedad (`int`, `bool` y similares), el nombre de la propiedad, los descriptores de acceso y los modificadores.

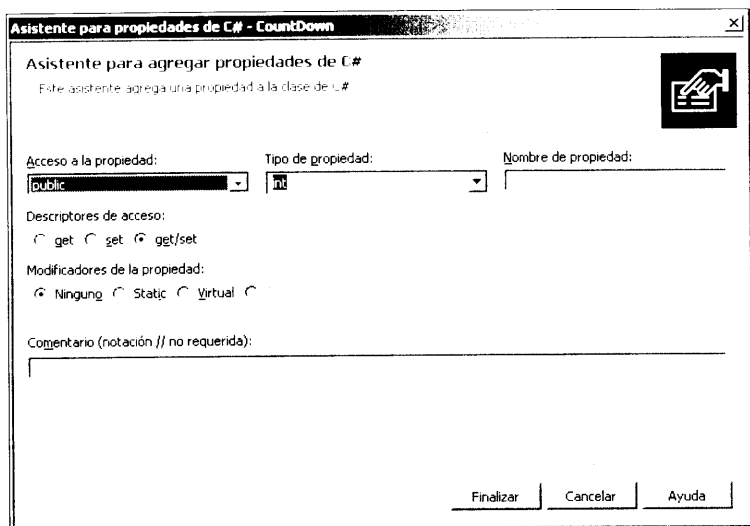


Figura 29.1. El asistente para agregar propiedades permite agregar una propiedad a su proyecto

Tras terminar con el asistente, verá el almacén de código para la propiedad:

```
public int TestProperty
{
    get
    {
        return 0;
    }
    set
```

```
    {  
    }  
}
```

En este código faltan algunos elementos básicos. Cuando se establece el valor de una propiedad, la información no se almacena. Debe crear antes una variable de clase privada que pueda ser usada por esta propiedad para almacenar y recuperar el valor. En este caso, simplemente debe incluir la siguiente declaración en la parte superior de su declaración de clase:

```
private int m_TestProperty;
```

Desde el método `set` de su propiedad, puede asignar los datos de su variable privada usando la palabra clave `value`:

```
set  
{  
    m_TestProperty = value;  
}
```

Una vez que hemos almacenado el valor de la propiedad en la variable de clase privada, debe asegurarse de que el método `Get` devuelve el valor correcto al ser usado. El siguiente código logra esto: simplemente devuelve el valor de la variable privada del bloque `get`, como muestra el siguiente fragmento:

```
get  
{  
    return m_TestProperty  
}
```

También es útil describir las propiedades de los componentes existentes en su control. Por ejemplo, imagine que ha creado un control que contiene una etiqueta. Su control debe contener una propiedad `Font` que obtenga y establezca la fuente del componente de la etiqueta de su control. Para permitir que el contenedor manipule la fuente de su etiqueta, simplemente cree una propiedad como la siguiente:

```
public Font CurrentFont  
{  
    get  
    {  
        return labell.Font;  
    }  
    set  
    {  
        labell.Font = value;  
    }  
}
```

Este código crea una propiedad llamada `CurrentFont`, que devuelve la fuente actual del control `label` cuando se le solicita. Cuando el contenedor

desear cambiar la fuente, simplemente tiene que asignar ese valor a la fuente actual del control de la etiqueta.

Métodos

Un método se define como una función que forma parte de una clase. Los métodos se implementan del mismo modo que en muchos otros lenguajes, como C++, C, VB y similares. Los métodos suelen usarse para iniciar una acción dentro de un control. Un ejemplo podría ser el método `Start` del control `Timer`.

Puede agregar un método a un control de una de estas dos formas. Puede usar el asistente para métodos de C#, al que se puede acceder mediante el Explorador de soluciones mientras se está en la Vista de clases o simplemente agregando una función pública al proyecto.

El asistente para métodos de C#, que se muestra en la figura 29.2, permite especificar el nombre del método, el tipo devuelto, el método de acceso y la información de parámetro de una forma muy sencilla.

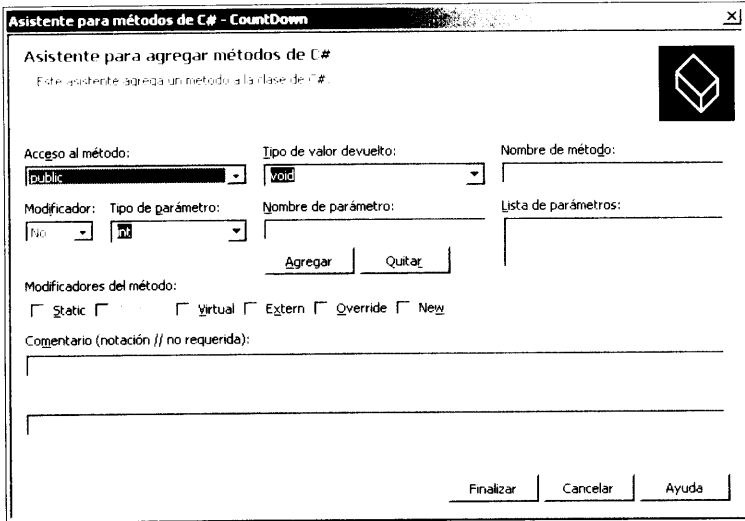


Figura 29.2. El asistente para métodos de C# crea la estructura básica del código necesario para el método

El inconveniente del asistente para métodos es que sólo proporciona tipos simples de devolución. Por ejemplo, imagine que quiere un método que devuelva un valor de tipo `Font`. El cuadro combinado Tipo de valor devuelto no ofrece el tipo `Font` como opción.

Cuando haga pruebas con el asistente aprenderá que puede introducir sus propios tipos en estos cuadros: sus opciones no se limitan a las que le ofrecen los cuadros. Las opciones que presentan sólo son los tipos más usados para que el programador inexperto no se confunda. Después de todo, si el asistente mostrara

todas las opciones posibles, tardaríamos mucho tiempo en desplazarnos por toda la lista en busca del tipo deseado.

.NET Framework también aporta a los métodos algunas funcionalidades muy necesarias. Ahora los métodos pueden sobrecargarse. Es decir, pueden tener el mismo nombre mientras su lista de parámetros sea diferente.

Campos

Los campos son simples variables públicas definidas en un control. La aplicación contenedora puede asignar al campo un valor determinado y recuperarlo. Como sólo son variables, no pueden tener funciones asociadas a ellos. Los campos no pueden realizar cálculos ni llamar a métodos cuando se les asigna un valor o se recupera de ellos un valor.

Hay dos tipos de campos: de instancias y estáticos. Un campo es, por defecto, un campo de instancia. Un campo de instancia puede contener sus propios datos en cada objeto que lo contenga. Cuando un campo es declarado como estático, contiene los mismos datos en todas las instancias del objeto. Un campo estático no suele ser aconsejable porque otras instancias del objeto pueden cambiar el valor y las instancias restantes no se percatarán de ello.

Los campos proporcionan elementos de código de bajo nivel pero también están limitadas en términos de funcionalidad. Puede asignar y recuperar los valores de un campo, pero al hacerlo no puede desencadenar otras acciones dentro del control. Los campos son útiles cuando se necesita asignar parámetros de operación a un control.

Eventos

Para definir un evento, antes debe hacer una declaración de clase pública, como la siguiente:

```
Public event EventHandler Expired;
```

Puede usar eventos con controles de Windows de uno de estos dos modos: puede hacer clic en el icono `Event` en la ventana `Propiedades` y hacer doble clic en el evento para agregar el controlador automáticamente o puede crear un controlador de eventos *usted* mismo.

Un evento básico se declara como un tipo `EventHandler`, pero tiene muchos más tipos de evento entre los que elegir y cada uno tiene una firma única. El parámetro de cada evento contiene una estructura `System.EventArgs`. Esta estructura permite al control pasar información al evento, con lo que define muchas aspectos, como qué desencadena el evento, qué control hace que se desencadene el evento y aspectos similares. Antes de escoger un tipo de evento para un control, asegúrese de que conoce los diferentes tipos, además de los parámetros que pasa cada uno.

Aprender con un ejemplo

Empecemos creando un control temporizador de cuenta atrás. Este ejercicio le permitirá estudiar los entresijos de la creación de controles.

Cómo crear un temporizador de cuenta atrás

Antes de empezar a crear el control temporizador de cuenta atrás, debe decidir lo que hará y lo que no hará este control. Esto simplificará el proceso de crear el control. Siempre es bueno tenerlo todo bien planificado antes de empezar un proyecto de control, incluso más que en cualquier proyecto de programación típico. Este control particular deberá tener las siguientes características:

- El control deberá ser un componente visible que pueda colocarse en un formulario. Este control deberá tener las propiedades habituales, como `BackColor`, `Anchor`, etc.
- El control deberá recibir un valor que indique el número de segundos desde los que empezar la cuenta atrás.
- El control deberá tener un método `Start()` y `Stop()` para empezar y terminar la cuenta atrás.
- Finalmente, El control deberá desencadenar un evento cuando la cuenta atrás llegue a cero.

Una vez definidos los requisitos, puede empezar a crear su control.

Abra Visual Studio .NET y cree un nuevo proyecto de biblioteca de control de Windows. Llame a este proyecto `CountDown`, como muestra la figura 29.3.

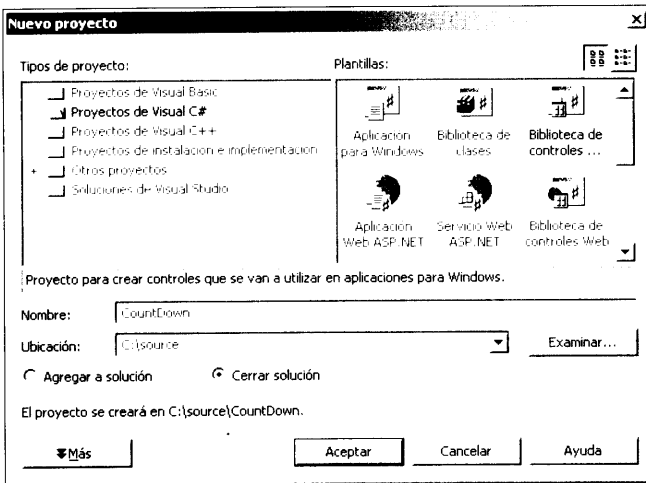


Figura 29.3. Cree un nuevo proyecto de biblioteca de control de Windows

1. Cuando Visual Studio haya creado la estructura básica de su control, se le mostrará un `UserControl`. Éste será la interfaz visual para su control `CountDown`. Aquí es donde se coloca cualquier elemento visual que quiera que el usuario vea o con el que quiera que interactúe. Lo primero que debe hacer es colocar un control `Label` en `UserControl`, como muestra la figura 29.4.
3. Cambie la propiedad `Anchor` de este control `Label` a `Top.Left.Bottom.Right`. Esto asegura que cuando el control cambie de tamaño, la etiqueta también lo haga. Debe tener mucho cuidado cuando agregue componentes existentes a un control porque muchos elementos pueden cambiar de aspecto cuando el usuario coloca un control en el formulario.
4. Ahora debe agregar un control `Timer` al proyecto (desde el cuadro de herramientas). El control `Timer` permite contar un segundo cada vez hasta que crea que el tiempo se ha agotado. El aspecto externo del control ya está completo. Ahora debe agregar el código que apoye a este control.

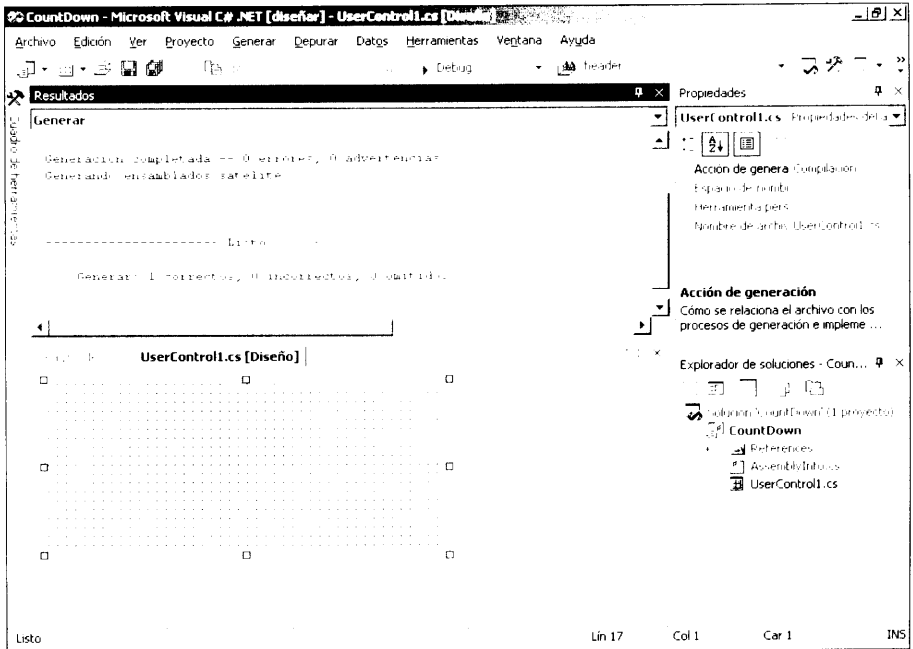


Figura 29.4. Coloque una etiqueta en `UserControl`

5. En el listado 29.1 declaró una variable de miembro para que contuviese el número de segundos desde los que el temporizador de cuenta atrás empezaría a contar. También debe declarar un `EventHandler` para que se dispare cuando el control llegue al final de la cuenta atrás. Coloque el siguiente código inmediatamente debajo de la declaración de clase de `UserControl`:

Listado 29.1. Miembros públicos

```
private int m_Seconds;  
public event EventHandler Expired;
```

6. La variable `m_Seconds` no necesita estar expuesta al contenedor de controles, sino sólo al propio control, por lo vamos a declarar la variable como privada. Al declarar un `EventHandler` público, `Expired` aparece en la ventana de eventos. De modo que puede programar este evento con sólo hacer doble clic en él.
7. En el evento `Tick` del objeto `Timer`, debe restar un segundo del número total de segundos. Luego puede comprobar si el número total de segundos ha llegado a cero. En caso afirmativo, debe detener el temporizador y desencadenar el evento `Expired`. Después, puede actualizar el control `Label` para que muestre el valor del control `CountDown Timer` en ese momento. En la ventana **Propiedades**, haga clic en el cuadro combinado **Selector de objeto** y seleccione el control `Timer`. A continuación haga clic en el botón **Eventos** de la barra de herramientas en la parte superior de la ventana **Propiedades**. Verá todos los eventos que muestra el control `Timer`. Haga doble clic en el evento `Tick`. Coloque el código del listado 29.2 en el controlador de eventos `Tick` del control `Timer`.

Listado 29.2. El evento `Tick` actualiza el control

```
private void timer1_Tick(object sender, System.EventArgs e)  
{  
    m_Seconds -= 1;  
    if (m_Seconds <= 0)  
    {  
        timer1.Stop();  
        Expired(this,e);  
    }  
    label1.Text = m_Seconds.ToString();  
}
```

8. Su control también necesita un método `public()` que permite que el contenedor inicie el control para que empiece a contar segundos. El listado 29.3 define el intervalo del control `Timer` (en milisegundos) y luego inicia el control `Timer`. Una vez que el temporizador ha sido iniciado, debe actualizar el control `Label` porque no se actualizará hasta un segundo después de que el temporizador desencadene el evento `Tick` por primera vez.
9. Para agregar un método al control, cambie la vista del explorador de soluciones a vista de clase haciendo clic en la ficha **Vista de clase**. Expanda la vista de clase hasta que encuentre las clases de controles. Haga clic con el botón derecho del ratón, seleccione **Agregar** y luego **Agregar méto-**

dos. El Asistente para métodos de C# se abrirá, como muestra la figura 29.5.

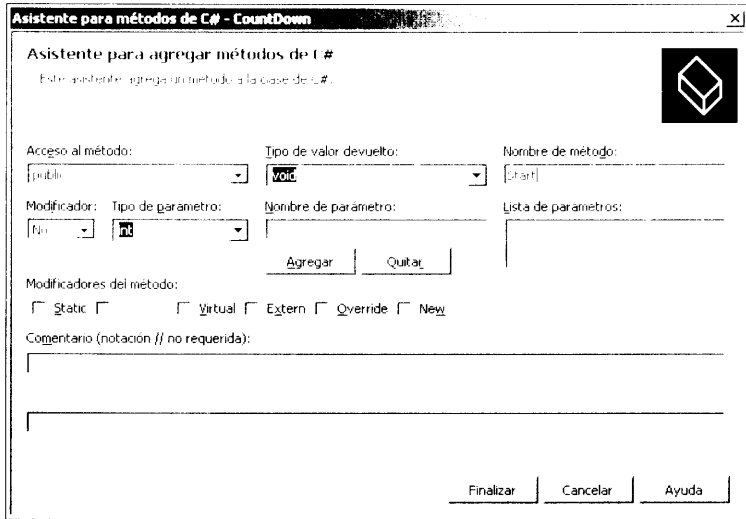


Figura 29.5. Asistente para métodos de C#

10. Llame a este método `Start` y asegúrese de que es un método público. Puede dejar todos los otros valores que se ofrecen por defecto. Tras crear el nuevo método, agregue el código del listado 29.3 al listado del código del método.

Listado 29.3. Método `Start`

```
public void Start()
{
    timer1.Interval = 1000;
    timer1.Start();
    label1.Text=m_Seconds.ToString();
}
```

11. Un método `Stop()` complementa al método `Start()`. El método `Stop()` debe detener en primer lugar al control `Timer` para que el evento `Tick` no vuelva a desencadenarse. También debe asignar el valor de la propiedad `Seconds` a la variable de miembro `m_Seconds`. Esto garantiza que si el control se inicia de nuevo, lo hará desde el principio y no desde donde se encontraba cuando fue detenido. Cree un nuevo método llamado `Stop()`, del mismo modo que creó el método `Start()`. Agregue el código del listado 29.4 a este nuevo método.

Listado 29.4. Método `Stop`

```
public void Stop()
{
```

```

timer1.Stop();
m_Seconds = this.Seconds;
}

```

Aún no ha creado la propiedad `Seconds`. La propiedad `Seconds` permite conseguir y establecer el número total de segundos desde los que el control empezará la cuenta atrás. Una propiedad se crea de una manera muy parecida a un método.

1. Desde la vista de clase del Explorador de soluciones, haga clic con el botón derecho y escoja **Agregar** y luego **Agregar propiedad**. El asistente para propiedades de C#, mostrado en la figura 29.6, se abrirá.

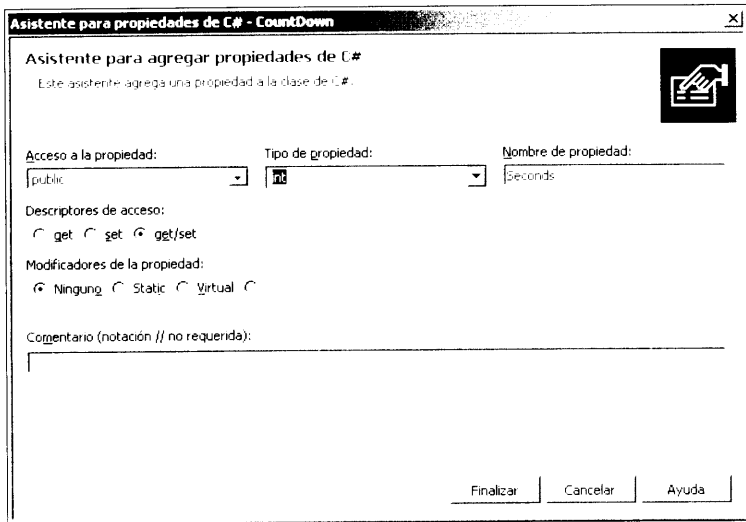


Figura 29.6. En el asistente para propiedades de C#, llame a esta propiedad `Seconds`. Asegúrese de que esta propiedad tenga acceso público y de que el tipo de la propiedad sea `int`

2. Queremos que la aplicación contenedora de los controles pueda asignar un valor a esta propiedad o leer un valor de ella. Por tanto, queremos que el descriptor de acceso sea `get/set`. Agregue el código del listado 29.5 al código creado por el asistente para propiedades de C#.

Listado 29.5. Propiedad `Seconds`

```

public int Seconds
{
    get
    {
        return m_Seconds;
    }
    set
    {

```

```
        m_Seconds = value;
    }
}
```

Preste especial atención a la palabra clave `value`. Cuando se pasa un valor a una propiedad (por ejemplo, `set`), puede ser recuperado mediante la palabra clave `value`.

Hasta ahora, su control no es muy sofisticado, ni presenta buen aspecto, pero funciona. En este momento, puede crear el nuevo control seleccionando **Generar solución** en el menú **Generar**.

Ahora genere una prueba de carga que use este control y permita comprobar todos los métodos, propiedades y eventos del control.

Cómo crear una prueba de carga Countdown

Uno de los puntos fuertes de Visual Studio es que permite agregar un proyecto a un proyecto existente, lo que es muy importante para probar controles. Esta posibilidad permite crear una aplicación de prueba de carga que usa uno de los controles que acaba de crear. Cuando encuentra algún error, no sólo se interrumpe la aplicación, sino que se interrumpe en el punto del código donde el control funcionó mal. Esto le ahorrará tener que pasarse incontables horas depurando código cuando empiece a crear controles complejos. Siga los siguientes pasos para crear una prueba de carga:

1. Desde el proyecto de control `CountDown`, haga clic en el menú **Archivo** y seleccione **Nuevo>Proyecto**. Se abrirá la ventana **Nuevo proyecto**.
2. Seleccione **Aplicación para Windows** y llame a esta aplicación `CDHarness`. En la parte inferior de esta ventana, asegúrese de que el botón de opción **Agregar a solución** esté seleccionado. Haga clic en **Aceptar**.
3. El **Explorador de soluciones** muestra los dos proyectos. Haga clic con el botón derecho del ratón en la aplicación `CDHarness` en el **Explorador de soluciones** y seleccione **Establecer como proyecto de inicio**. A partir de ahora, cuando ejecute la aplicación, la prueba de carga será la aplicación principal que ejecuta y puede usar cualquier otro proyecto que esté abierto en ese momento en el "grupo".
4. Asegúrese de que está en el proyecto `CDHarness` haciendo clic en el **Explorador de soluciones**. Haga clic con el botón derecho del ratón en el cuadro de herramientas y escoja **Personalizar cuadro de herramientas**.
5. Cuando se abra la ventana **Personalizar cuadro de herramientas**, haga clic en la ficha **Componentes de .NET Framework**. Ahí es donde se agrega el nuevo control al proyecto.

6. Haga clic en el botón **Examinar** y busque el control temporizador `CountDown`. Cuando lo encuentre, selecciónelo y márkelo. Haga clic en **Aceptar**.
7. En la lista de controles, verá este control; marque el cuadro junto a él. Ahora puede hacer clic en **Aceptar** en este formulario. En el **Cuadro de herramientas**, verá el control temporizador `CountDown` en la lista.
8. Haga doble clic en el control para agregarlo a su proyecto. Mientras está en el cuadro de herramientas, agregue también dos botones a su formulario. El primer botón se usa para iniciar el temporizador de cuenta atrás y el segundo botón se usa para detener el control `Countdown`. Alinee y establezca las propiedades de texto como se muestra en la figura 29.7.

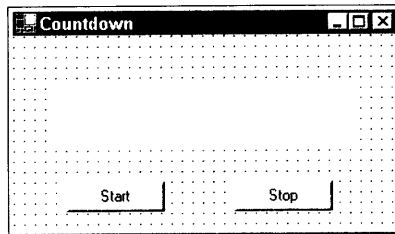


Figura 29.7. La interfaz principal de su aplicación de carga `CountDown`

9. Haga doble clic en el botón **Start** de su aplicación para que pueda empezar a codificar. Agregue el código del listado 29.6 al evento `Click` del botón **Start**.

Listado 29.6. Inicia el control

```
private void button1_Click(object sender, System.EventArgs e)
{
    cdTimer1.Seconds=60;
    cdTimer1.Start();
}
```

10. El botón **Start** sirve sólo para una cosa: para preparar e iniciar el control `CountDown`. Para empezar, establezca en 60 el número de segundos y luego llame al método `Start` del control.
11. También necesita un modo de detener el control. Haga doble clic en el botón **Stop** para insertar código en el evento `Click` de este botón. Inserte el siguiente código del listado 29.7 en el evento `Click` del botón `Stop`.

Listado 29.7. Detiene el control

```
private void button2_Click(object sender, System.EventArgs e)
{
    cdTimer1.Stop();
}
```

12. El evento `Expired` del control `CountDown` debe realizar alguna acción que le informe de que la cuenta atrás ha llegado a cero. En la ventana **Propiedades**, haga clic en el botón **Eventos** en la barra de herramientas. A continuación, haga doble clic en el evento `Expired`. Es en este código en el que debe codificar esa acción especial. No intente nada extravagante: simplemente vamos a cambiar el título de su formulario a `Done`. Inserte el código del listado 29.8 en el evento `Expired` del control `CountDown`.

Listado 29.8. Código del evento `Expired`

```
private void cdTimer1_Expired(object sender, System.EventArgs e)
{
    this.Text="Done!";
}
```

Su control y su aplicación de prueba ya están listos para ser ejecutados. Pulse **F5** para ejecutar la aplicación. Cuando se abra la aplicación, haga clic en el botón **Start** para iniciar la cuenta atrás. Su aplicación de prueba deberá empezar a contar hacia atrás. Cuando la cuenta atrás llegue a cero, el título del formulario deberá cambiar, gracias al evento `Expired`.

Cómo usar una biblioteca de clases

Una biblioteca de clases es un modo eficiente de reutilizar y distribuir el código reutilizable. Las bibliotecas de clases se almacenan en las DLL, pero no es necesario envolver el código con instrucciones y declaraciones como en otros lenguajes, como C++.

Como con cualquier componente reutilizable, las bibliotecas de clase proporcionan constructores y destructores de clase para la inicialización y la limpieza, pero estos elementos no son realmente imprescindibles.

Para empezar su viaje por las bibliotecas de clase, cree un sencillo DLL con algunos métodos públicos de prueba. A continuación, cree el control con algunos atributos adicionales.

Cómo crear una clase para calcular el efecto de viento

La biblioteca de clases de cálculo del efecto de viento no sólo proporciona una función para calcular la temperatura del aire, si que también proporciona funciones para diferenciar los grados Celsius de los grados Fahrenheit y viceversa.

Como las operaciones que va a realizar en esta biblioteca de clases pertenecen a dos categorías distintas, conversión y cálculo, debe incluir dos clases separadas. Cree un nuevo proyecto de biblioteca de clases y llámelo `Temperature`. Vi-

sual Studio creará el código que será la estructura básica mostrado en el listado 29.9.

Listado 29.9. Código creado para una biblioteca de clases por Visual Studio

```
using System;

namespace Temperature
{
    /// <summary>
    /// Descripción breve de Class1.
    /// </summary>
    public class Class1
    {
        public Class1()
        {
            //
            // TODO: Agregar aquí la lógica del constructor
            //
        }
    }
}
```

Visual Studio ha creado un espacio de nombres, una clase y un constructor de clase. La clase y el constructor de clase son públicos. Esto no es una coincidencia. Cualquier aplicación que use esta biblioteca necesita acceder tanto a la clase como a los miembros específicos dentro de la clase.

1. Para empezar nuestro ejemplo, elimine la clase y el constructor de clase y reemplácelos por el código del listado 29.10.

Listado 29.10. La clase y el constructor Calc

```
public class Calc
{
    public Calc()
    {
    }
}
```

2. La clase `Calc` es donde colocará el método de cálculo del efecto de viento. De esta forma, estamos seguros de que nuestro cálculo del efecto de viento queda separado de los métodos de conversión que pronto agregará. Inserte el código del listado 29.11 en la clase `Calc`.

Listado 29.11. Cálculo del efecto de viento actual

```
public double WindChill(double DegreesF, double WindSpeedMPH)
{
    double WindRaised;
    WindRaised = System.Math.Pow(WindSpeedMPH, 1.6);
}
```

```

    return 35.74 + (.6215 * DegreesF) - (35.75 * WindRaised) +
(.4275 * DegreesF * WindRaised);
}
}

```

3. También agregaremos algunos métodos que no son cálculos, sino conversiones. Por tanto, deberá añadir una nueva clase. Esta nueva clase se llamará `Conversion`, como se puede ver en el listado 29.12. Agregue la nueva clase y constructor de clase al proyecto.

Listado 29.12. Agregar una segunda clase a la biblioteca de clases

```

public class Conversion
{
    public Conversion()
    {
    }
}

```

4. Inicie la nueva clase con una función que calcule la temperatura en grados Fahrenheit a partir de la temperatura en grados Celsius. Es poco probable que el valor devuelto o el parámetro sean valores enteros, de modo que es importante que ambos sean valores `double`. El listado 29.13 contiene el listado completo de la función. Agregue este código a la clase `Conversion`.

Listado 29.13. Conversión de Celsius a Fahrenheit

```

public double CelcToFahr(double Celsius)
{
    return (9/5) * (Celsius + 32);
}

```

5. Como ha incluido una función de conversión de Celsius a Fahrenheit, es lógico incluir la conversión inversa. Agregue el método del listado 29.14 a la clase `Conversion`.

Listado 29.14. Conversión de Fahrenheit a Celsius

```

public double FahrToCelc(double Fahrenheit)
{
    return (5/9) * (Fahrenheit - 32);
}

```

Con esto finaliza la funcionalidad que queremos incluir en el objeto biblioteca de clases, de modo que ya puede crear la DLL seleccionando **Generar solución** en el menú **Generar**. Una vez que todo está en su sitio, observe cómo se utilizan los métodos públicos que hay en su interior.

1. Cree un nuevo proyecto de aplicación de consola y llámelo `DLLTest`. En el menú **Proyecto**, seleccione **Agregar referencia** y a continuación haga

clic en el botón **Examinar**. Busque la DLL que acaba de crear y haga doble clic sobre ella. Ya puede hacer clic en **Aceptar** para salir de la ventana **Agregar referencia**.

2. Tras agregar una referencia, puede simplemente crear una nueva variable del tipo adecuado y hacer referencia al método que prefiera. El listado 29.15 contiene el listado completo del código de la aplicación DLLTest.

Listado 29.15. Código fuente de la aplicación DLLTest

```
using System;

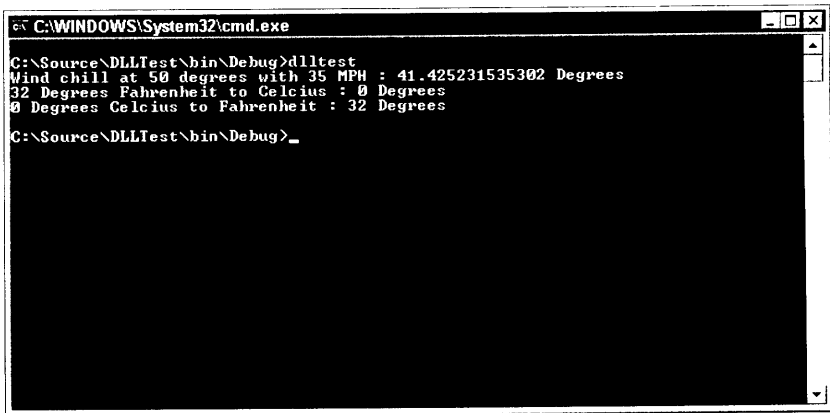
namespace DLLTest
{
    /// <summary>
    /// Descripción breve de Class1.
    /// </summary>
    class Class1
    {
        static void Main(string[] args)
        {
            Temperature.Calc WMethod = new Temperature.Calc();
            Temperature.Conversion ConvMethod = new
            Temperature.Conversion();

            Console.WriteLine("Wind chill at 50 degrees with 35 MPH
: {0} Degrees",WMethod.WindChill(50,35));
            Console.WriteLine("32 Degrees Fahrenheit to Celsius :
{0} Degrees",ConvMethod.FahrToCelc(32));
            Console.WriteLine("0 Degrees Celsius to Fahrenheit :
{0} Degrees",ConvMethod.CelcToFahr(0));
        }
    }
}
```

El acceso a los métodos es una tarea muy sencilla: cree nuevas variables del tipo `Temperature.Conversion` y `Temperature.Calc` y luego acceda a sus métodos públicos. Esta aplicación de ejemplo calcula el efecto del viento con una temperatura de 50 grados y una velocidad del viento de 35 millas por hora; a continuación calcula la temperatura en grados Celsius y Fahrenheit. El resultado de esta aplicación se muestra en la figura 29.8.

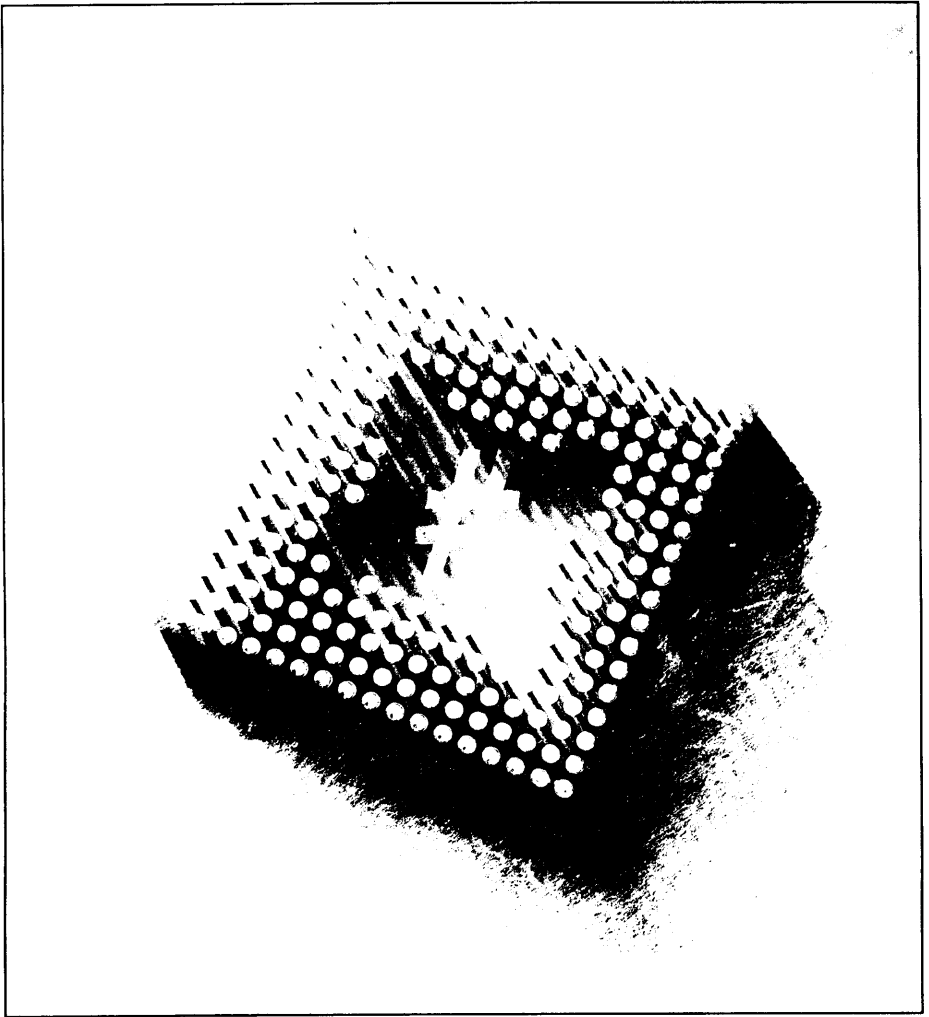
Resumen

.NET presenta algunas innovaciones en la forma de construir controles extremadamente importantes. A medida que cree controles para sus propios proyectos o para implementar en la Web, irá descubriendo algunos avances en la programación, como el objeto `State`, que le ahorrarán grandes cantidades de tiempo.



```
C:\WINDOWS\System32\cmd.exe
C:\Source\DLLTest\bin\Debug>dlltest
Wind chill at 50 degrees with 35 MPH : 41.425231535302 Degrees
32 Degrees Fahrenheit to Celcius : 0 Degrees
0 Degrees Celcius to Fahrenheit : 32 Degrees
C:\Source\DLLTest\bin\Debug>_
```

Figura 29.8. DLLTest le enseña a usar un control de biblioteca de clases



30 **Cómo construir aplicaciones móviles**

Las aplicaciones Web móviles son un grupo de tecnologías emergentes que permiten que el contenido Web sea accesible a un número mayor de público que el que puede ofrecer Internet actualmente. Puede crear aplicaciones de Intranet corporativa para los empleados que se mueven entre los edificios de la compañía o para los que se desplazan entre continentes durante sus viajes de negocios. Este capítulo describe varios apartados de las aplicaciones Web móviles.

La red inalámbrica

El concepto de acceso a Internet mediante dispositivos móviles lleva mucho tiempo entre nosotros pero ha tardado en ser aceptado. Las herramientas adecuadas para crear contenido Web han sido bastante escasas. .NET Framework, junto con el Mobile Internet Toolkit de Microsoft, permite crear emocionantes aplicaciones Web que se pueden usar en distintos tipos de dispositivos móviles.

Estos dispositivos móviles incluyen dispositivos Windows CE, dispositivos Pocket PC y muchos teléfonos móviles. Evidentemente, la mayoría de los dispositivos móviles están muy limitados en comparación con los navegadores Web a los que estamos acostumbrados. No es sólo que los dispositivos móviles dispongan

de menos espacio para el contenido, muchos de estos dispositivos carecen de color o, en alguna otra forma, de la capacidad para mostrar gráficos.

Este capítulo empieza estudiando el software que necesita, junto con los emuladores adecuados que puede usar para probar su código, en caso de que no tenga acceso a un dispositivo móvil con acceso a Internet.

Introducción al Mobile Internet Toolkit

El Mobile Internet Toolkit de Microsoft permite a Visual Studio crear aplicaciones Web seleccionándolo como proyecto en el menú Nuevo proyecto. Este conjunto de herramientas no se incluye en el mismo paquete que Visual Studio .NET, por lo que debe ser descargado de forma separada desde la página Web de Microsoft. El Mobile Internet Toolkit actualmente se encuentra en la versión 1.0 y se puede descargar desde <http://msdn.microsoft.com/download>. Cuando llegue a esta página, debe seleccionar Software Development Kits en el marco a la izquierda de su navegador Web y luego seleccionar Microsoft Mobile Internet Toolkit.

El Software Development Kit (SDK) actual ocupa algo más de 4MB y contiene varios controles ASP.NET para generar Lenguaje de marcado inalámbrico (WML) y varios detalles de HTML, además del módulo que se añade a Visual Studio .NET, documentación y ejemplos de aplicaciones. Asegúrese de que al instalar el SDK, todos los procesos de Visual Studio están cerrados.

Emuladores

Los emuladores permiten escribir aplicaciones para dispositivos y probarlas sin tener que comprar realmente uno de los dispositivos. Hay emuladores para teléfonos móviles, PDA, equipos de escritorio y dispositivos intermedios. Los siguientes apartados tratan sobre algunos de los emuladores más populares, que puede descargar y empezar a escribir aplicaciones para ellos.

Nokia

En la página Web de Nokia (<http://www.nokia.com>), puede descargarse un emulador de teléfono Nokia para probar sus aplicaciones Web para teléfonos móviles. Esta descarga es de casi 22MB y requiere el entorno de ejecución Java, lo que añade otros 5MB a su descarga. El emulador Nokia actualmente le permite elegir entre uno de los tres diferentes modelos Nokia que ofrece para que realice sus pruebas. Esta aplicación es un valioso recurso para probar sus aplicaciones.

Pocket PC

El emulador de Pocket PC se incluye en el Microsoft Embedded Devices Toolkit. Es un excelente emulador por si no posee un Pocket PC ya que Pocket Internet

Explorer admite muchas más funcionalidades que la mayoría de los dispositivos móviles del mercado actual. Tenga en cuenta que el tamaño real de este conjunto de herramientas es superior a los 300MB. Si no posee una conexión a Internet de alta velocidad, es mejor que use Microsoft Mobile Explorer, del que le hablamos en la siguiente sección.

Microsoft Mobile Explorer

Microsoft Mobile Explorer es una descarga de sólo 3MB y se usa principalmente para integrarse con Visual Studio. Cuando pruebe sus aplicaciones, establezca MME como su navegador por defecto.

Cómo crear un calculador de edades

Tras instalar el Mobile Internet Toolkit de Microsoft, cree un nuevo proyecto de C#. Aparecerá una nueva opción llamada Mobile Web Application, como muestra la figura 30.1.

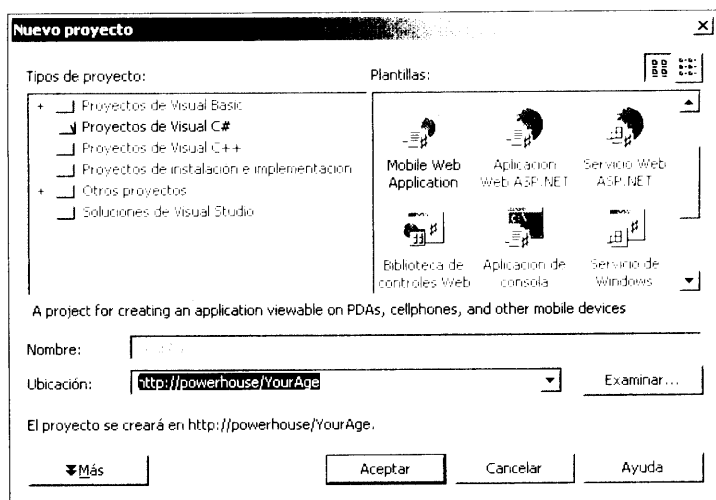


Figura 30.1. La nueva opción Mobile Web Application se agregó cuando instaló Mobile Internet Toolkit.

Esta aplicación de ejemplo toma su año de nacimiento para calcular cuál es su edad actual (o la que debería ser).

1. Llame al proyecto `YourAge` y cuando Visual Studio cree la estructura de su aplicación, verá un formulario Web Mobile en blanco llamado `Form1`.
2. Haga clic con el botón derecho del ratón en este formulario y seleccione **Copiar**. Ahora haga clic con el botón derecho del ratón debajo del formulario y seleccione **Pegar**. Esta aplicación necesita dos formularios:

uno que adquiere la información del usuario y otro que muestra los resultados.

- Una vez que tenga los dos formularios, coloque un control `Button`, un `Label` y un `TextBox` en `Form1` y coloque un control `Label` y un `Link Label` en `Form2`. Coloque estos controles como se muestra en la figura 30.2.

Una vez situados los controles, debe asignar todas las propiedades adecuadas. La tabla 30.1 contiene las propiedades de cada control. Colóquelas adecuadamente antes de seguir adelante.

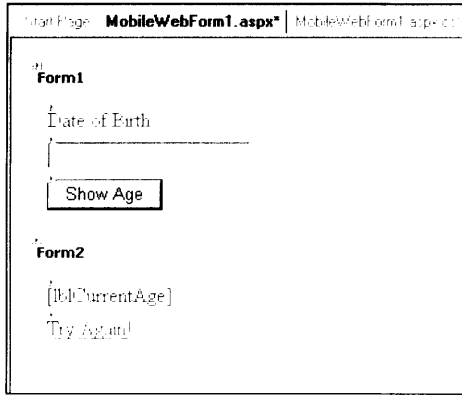


Figura 30.2. Estos formularios son necesarios para la aplicación `YourAge`.

Tabla 30.1. Propiedades del control `YourAge`

Formulario	Control	Propiedad Name	Propiedad Text
Form1	Label	Label1	Year of Birth
Form1	Text Box	txtDOB	<Empty>
Form1	Button	Command1	Show Age
Form2	Label	lblCurrentAge	<Empty>
Form2	Link Label	Link1	Try Again

Debe establecer una propiedad antes de empezar a codificar. Seleccione el control `Link Label` y asigne la propiedad `NavigateUrl` a `Form1`. Esto asegura que, cuando se hace clic en el vínculo, el usuario es enviado de vuelta al `Form1` del proyecto.

Antes de empezar con el código necesario para su programa, examine el código oculto de estos formularios Web Mobile. En la parte inferior del diseñador de formularios hay dos botones con la etiqueta **Diseño** y **HTML**. Haga clic en el

botón **HTML** para mostrar el código HTML oculto de estos formularios, que debe ser igual que el código del listado 30.1.

Listado 30.1. Código HTML oculto de los formularios Web YourAge

```
< @ Register TagPrefix="mobile"
Namespace="System.Web.UI.MobileControls"
Assembly="System.Web.Mobile, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" >
< @ Page language="c#" Codebehind="MobileWebForm1.aspx.cs"
Inherits="YourAge.MobileWebForm1" AutoEventWireup="false" >
<meta name="GENERATOR" content="Microsoft Visual Studio 7.0">
<meta name="CODE_LANGUAGE" content="C#">
<meta name="vs_targetSchema" content="http://
schemas.microsoft.com/Mobile/Page">
<body Xmlns:mobile="http://schemas.microsoft.com/Mobile/
WebForm">
  <mobile:Form id="Form1" runat="server">
    <mobile:Label id="Label1" runat="server">Year of Birth:</
mobile:Label>
    <mobile:TextBox id="txtDOB" runat="server" />
mobile:TextBox>
    <mobile:Command id="Command1" runat="server">Show Age</
mobile:Command>
  </mobile:Form>
  <mobile:Form id="Form2" runat="server">
    <mobile:Label id="lblCurrentAge" runat="server" />
mobile:Label>
    <mobile:Link id="Link1" runat="server"
NavigateUrl="#"Form1">Try Again!</mobile:Link>
  </mobile:Form>
</body>
```

Este código es el estándar de HTML combinado con algunos elementos que están definidos por el espacio de nombres `MobileControls`. Esta unión de la página HTML y la clase .NET `MobileControls` puede apreciarse en la línea que sigue a `Namespace`. Cuando empiece a codificar, observe que varios espacios de nombres ya han sido agregados al proyecto por este motivo.

Ahora está listo para empezar a codificar su aplicación Web móvil. Este proyecto necesita que haya código en un lugar. Haga doble clic en el botón que anteriormente colocamos en `Form1`. El listado 30.2 contiene el código que debe insertar tras el evento clic de este botón.

Listado 30.2. Cómo calcular la edad en el evento Click

```
private void Command1_Click(object sender, System.EventArgs e)
{
    int iDOB = System.Convert.ToInt16(txtDOB.Text);
    int YearsOld;

    if (iDOB > 1000)
```

```

        YearsOld = 2002 - iDOB;
    else
        YearsOld = 199 - iDOB + 2;

    lblCurrentAge.Text = YearsOld.ToString() + " years old.";

    ActiveForm = Form2;
}

```

En esta parte del código, deberá realizar algunas tareas. En primer lugar, debe convertir en un valor entero el año que se introduce mediante la clase `Convert`. A continuación, dependiendo del tamaño de ese año (dos o cuatro dígitos), calcule la edad en consecuencia, usando como año actual 2002.

Una vez que se ha calculado la edad, asigne la edad y una cadena a su etiqueta `currentAge`. Su aplicación ya es funcional, pero aunque ha asignado el resultado a la etiqueta adecuada, todavía no se muestra la pantalla que contiene la información. En la última línea del listado 30.2, asigne `ActiveForm` a `Form2`. Esto carga `Form2` en el navegador del usuario para mostrar los resultados.

Puede ejecutar esta aplicación pulsando **F5** y si todo el código se ha realizado correctamente, se abrirá el navegador Web que tenga por defecto. Para ver esta aplicación en un dispositivo móvil, ejecute su emulador preferido e introduzca la URL adecuada. La figura 30.3 muestra la aplicación ejecutándose en un emulador de Pocket PC.

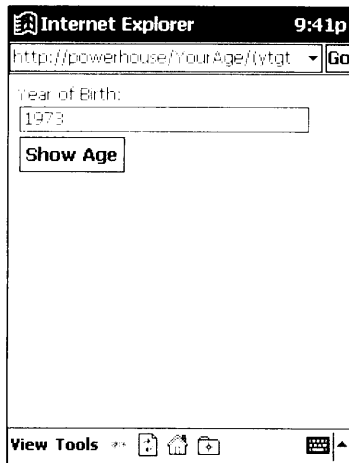


Figura 30.3. Ejecución de `YourAge` en un emulador de Pocket PC

Tras introducir su año de nacimiento, haga clic en el botón **Show Age** y se le mostrará el `Form2`, como muestra la figura 30.4.

Felicidades por su primera aplicación Web móvil. Aquellos que usaron Internet Explorer o el emulador de Pocket PC para hacer su prueba, se estarán preguntando cómo se mostrará la aplicación Web en un teléfono móvil. La figura 30.5 muestra la misma aplicación ejecutándose en el emulador de teléfono móvil Nokia.

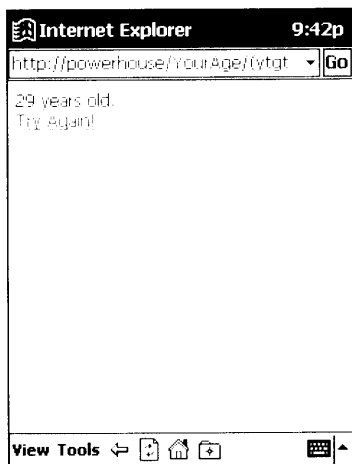


Figura 30.4. Resultado de YourAge en un Pocket PC

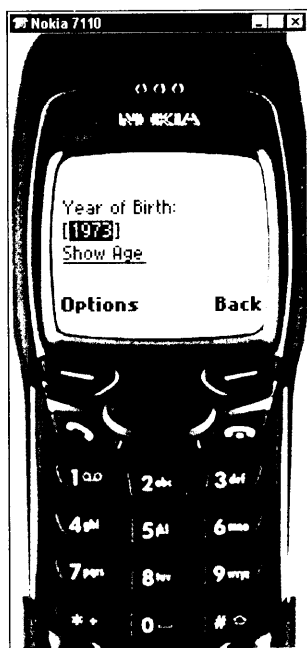


Figura 30.5. Nokia 7110 ejecutando la aplicación YourAge

Cómo puede ver, la página presenta un aspecto muy parecido al que mostraba en los anteriores emuladores, excepto en algunos detalles de desplazamiento entre opciones. Como los Pocket PC son dispositivos en los que se señala y se hace clic, el desplazamiento se realiza mucho más suavemente. El teléfono móvil, por otra parte, realiza todos sus desplazamientos entre opciones usando sus, aproximadamente, 20 botones. Es decir, no puede simplemente hacer clic en un botón o un vínculo para avanzar; debe usar los botones para desplazarse hacia arriba o abajo y seleccionar los vínculos antes de poder seguirlos.

Funciones de los dispositivos móviles

La clase `System.Web.Mobile.MobileCapabilities` contiene varias docenas de propiedades que se usan para detectar las funciones de un dispositivo móvil. Por ejemplo, puede consultar la resolución de pantalla de un dispositivo, comprobar si es de color o de blanco y negro y consultar si el dispositivo es capaz de mandar correo, por nombrar sólo algunas.

Al crear aplicaciones Web móviles, es importante tener en cuenta los distintos tipos de dispositivo que pueden acceder a la aplicación. A diferencia de los navegadores convencionales, cuya funcionalidad varía muy ligeramente, puede haber enormes diferencias entre ver una página en un teléfono móvil o, por ejemplo, en un PDA que utiliza Windows CE. Debe asegurarse de que sus páginas serán procesadas adecuadamente por cada dispositivo.

Empiece usando la clase `Device` para comprobar el número de caracteres máximos, horizontal y verticalmente, que admite la pantalla de su dispositivo móvil. Abra la aplicación `YourAge` que creó en el listado 30.2 y agregue dos controles `Label` a `Form2`. Cambie las propiedades `Name` de estas etiquetas a `lblHeight` y `lblWidth`.

Ahora debe modificar el código fuente original para llenar estas etiquetas después de que se muestre `Form2`. El listado 30.3 contiene el código que debe añadir (en negrita) para que la nueva funcionalidad surta efecto.

Listado 30.3. Cómo mostrar las funciones de un dispositivo

```
private void Command1_Click(object sender, System.EventArgs e)
{
    int iDOB = System.Convert.ToInt16(txtDOB.Text);
    int YearsOld;

    if (iDOB > 1000)
        YearsOld = 2002 - iDOB;
    else
        YearsOld = 100 - iDOB + 2;

    lblCurrentAge.Text = YearsOld.ToString() + " years old." ;
    lblHeight.Text = "Height: " +
Device.ScreenCharactersHeight;
    lblWidth.Text = "Width: " + Device.ScreenCharactersWidth;
    ActiveForm = Form2;
}
```

En un Pocket PC, puede conseguir 17 x 34 caracteres, mientras que en un teléfono móvil Nokia 7110 sólo puede conseguir 4 x 22, como muestra la figura 30.6.

No todas las propiedades de esta clase funcionan en todos los dispositivos. El archivo de ayuda de Mobile Internet Toolkit de Microsoft contiene una tabla de

funciones de los dispositivos que define qué propiedades funcionarán normalmente con HTML, cHTML y con WML.

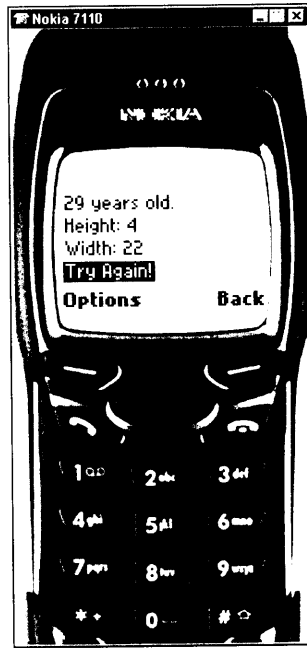


Figura 30.6. Altura y anchura de pantalla en un emulador de Nokia

El cuadro de herramientas de Visual Studio .NET también tiene un control llamado `DeviceSpecific` que puede ser situado en un formulario para que realice ciertas tareas (dependiendo del dispositivo con el que se esté comunicando). Esto se consigue mediante filtros y reduce considerablemente el esfuerzo que sería necesario para codificar todos los distintos contextos posibles.

Funcionamiento de los controles móviles

Los controles Web móviles deben ser muy versátiles a la hora de mostrar interfaces visuales. Algunos controles necesitan más espacio del disponible en la mayoría de los dispositivos móviles. Cuando esto sucede, el control debe determinar cómo manejar la situación. Las siguientes secciones estudian dos de estos controles, `Calendar` e `Image`, y cómo modifican sus interfaces visuales cuando es necesario.

Cómo usar el control `Calendar`

El control `Calendar` permite mostrar un completo calendario en su página Web móvil. Para comprobar lo versátil que es este control, cree una nueva aplica-

ción Web móvil y coloque un control `Calendar` en el `Form1` del proyecto. Cuando ejecute esta nueva aplicación mediante su navegador por defecto o mediante el emulador de `Pocket PC`, verá un calendario mensual completo que permite hacer clic en cualquier día de la semana. En la figura 30.7 se muestra esta aplicación ejecutándose en un `Pocket PC`.

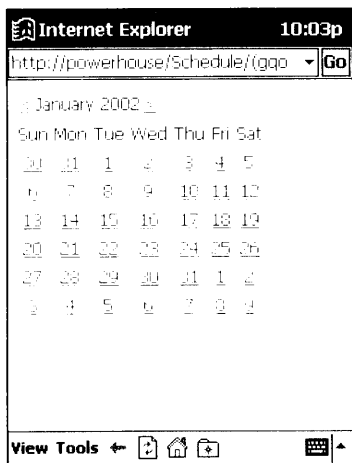


Figura 30.7. Aplicación de prueba de `Calendar` ejecutándose en un emulador de `Pocket PC`

¿Cómo puede mostrarse este calendario en un dispositivo mucho más pequeño, como por ejemplo un teléfono móvil? El control `Calendar` sabe en qué tipo de dispositivo va a ser mostrado y cambia su interfaz de usuario en consecuencia. La figura 30.8 muestra esta misma aplicación Web ejecutándose en un emulador de teléfono móvil. Ya no ve el calendario, sino que ve la fecha actual seguida de dos opciones. Estas opciones le permiten introducir directamente una fecha o buscarla semana a semana y luego por cada día de la semana.

Este tipo de comportamiento no exige ningún tipo de programación, lo que libera al programador de mucho trabajo.

Cómo usar el control `Image`

El control `Image` es otro control único muy parecido al control `Calendar`. Le permite mostrar una imagen en varios tipos de dispositivos móviles con poca o ninguna programación. Cree una nueva aplicación Web móvil de `C#` y coloque un control `Image` en `Form1`. Proporcione a la `ImageUrl` de este control `Image` la ruta de la imagen que desea mostrar. Esta imagen puede ser un mapa de bits (bmp), una imagen JPEG (jpg) o alguno de los otros tipos que acepta. Para el ejemplo, escoja una imagen en color jpg.

Al ejecutar esta imagen en un `Pocket PC` se muestra la imagen correctamente, como muestra la figura 30.9.

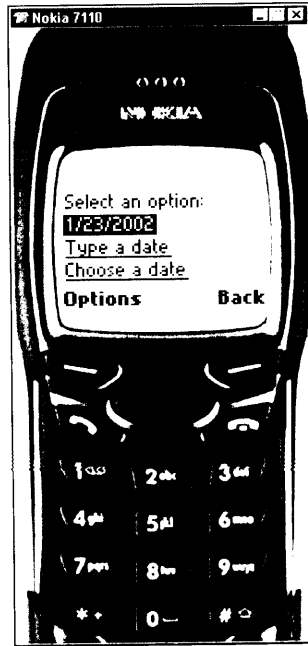


Figura 30.8. Aplicación de prueba de Calendar ejecutándose en un emulador de Nokia



Figura 30.9. Pocket PC muestra la imagen sin problemas aparentes

Si intenta ver esta misma página con el emulador de teléfono móvil Nokia, recibirá un mensaje de error indicando que no se pudo cargar la página. Esto obviamente es debido a que la página no es de un tipo que se pueda ver en una pantalla en blanco y negro de un tamaño tan pequeño.

Sin embargo, se puede solventar este problema. Puede convertir su imagen JPG en una imagen de mapa de bits de dos colores (blanco y negro), también conocido como de *mapa de bits inalámbrico* (WBMP), que el dispositivo móvil

puede procesar. En el evento `Page_Load` del `Form1`, puede comprobar la propiedad `Device.PreferredRenderingType` para ver qué imagen debe ser cargada. Si esta propiedad devuelve `wml11`, debe ajustar la propiedad `ImageUrl` del control `Image` a `WBMP_picture`; en caso contrario, puede mostrar el original.

Paginación en dispositivos móviles

La *paginación* es la capacidad de un dispositivo móvil de dividir grandes cantidades de contenido en varias páginas de información. Esto sucede, por ejemplo, cuando se muestra una larga lista de contactos en un formulario Web móvil y el dispositivo concreto no puede mostrarlo entero en su pequeña pantalla.

Le alegrará saber que puede programar este tipo de comportamiento para que su formulario Web móvil lo controle automáticamente con sólo cambiar dos propiedades. La propiedad `Paginate`, cuando es `True`, divide automáticamente todo el contenido en varias páginas de información, dependiendo de las posibilidades de su dispositivo remoto. También debe asignar la propiedad `ItemsPerPage` a un control determinado (por ejemplo, el control `List`) para forzar la paginación de un determinado número de elementos. Sin embargo, por lo general esto no es necesario ya que el número por defecto, siete, suele ser dar buen resultado.

Puede comprobarlo creando una nueva aplicación Web móvil llamada `Contacts`. Asigne a la propiedad `Paginate` de `Form1` el valor `True` y agregue un control `List` a la página. Déle a la propiedad `ItemsPerPage` del control `List` el valor `7`.

Ahora debe agregar algunos elementos a este control `List`, como muestra el siguiente código:

```
List1.Items.Add("Kim Pack");
List1.Items.Add("Timothy Hyde");
List1.Items.Add("Donna Malone");
List1.Items.Add("Joshua Trueblood");
List1.Items.Add("Staci Springer");
List1.Items.Add("Chris Stephens");
List1.Items.Add("Amy Sherman");
List1.Items.Add("Steve Million");
List1.Items.Add("Jim Mattingly");
List1.Items.Add("Ryan Boyles");
List1.Items.Add("Scott Leathers");
```

Estos 11 elementos se agregan a su lista. Por tanto, debería de ver siete elementos en la primera página, junto a un vínculo `Next` a una segunda página que contiene cuatro elementos. Ejecute la aplicación en el dispositivo móvil preferido. Como se esperaba, sus primeros siete elementos aparecen en la primera página, junto a un vínculo hacia la siguiente página, como muestra la figura 30.10.



Figura 30.10. Los siete primeros elementos de la lista

El usuario puede hacer clic en el botón `Next` para ver la segunda página. En la segunda página, no tiene cuatro elementos, como se esperaba, sino siete elementos, debido a que el control `List` envolvió los contenidos de la primera página en la segunda, como muestra la figura 30.11.



Figura 30.11. Los elementos de la lista restantes se muestran en la segunda página.

Como acabamos de demostrar, cuando la paginación está activada, se pueden aprovechar varias propiedades, como `Page`, que le permite especificar un número de índice de la página que va a ver.

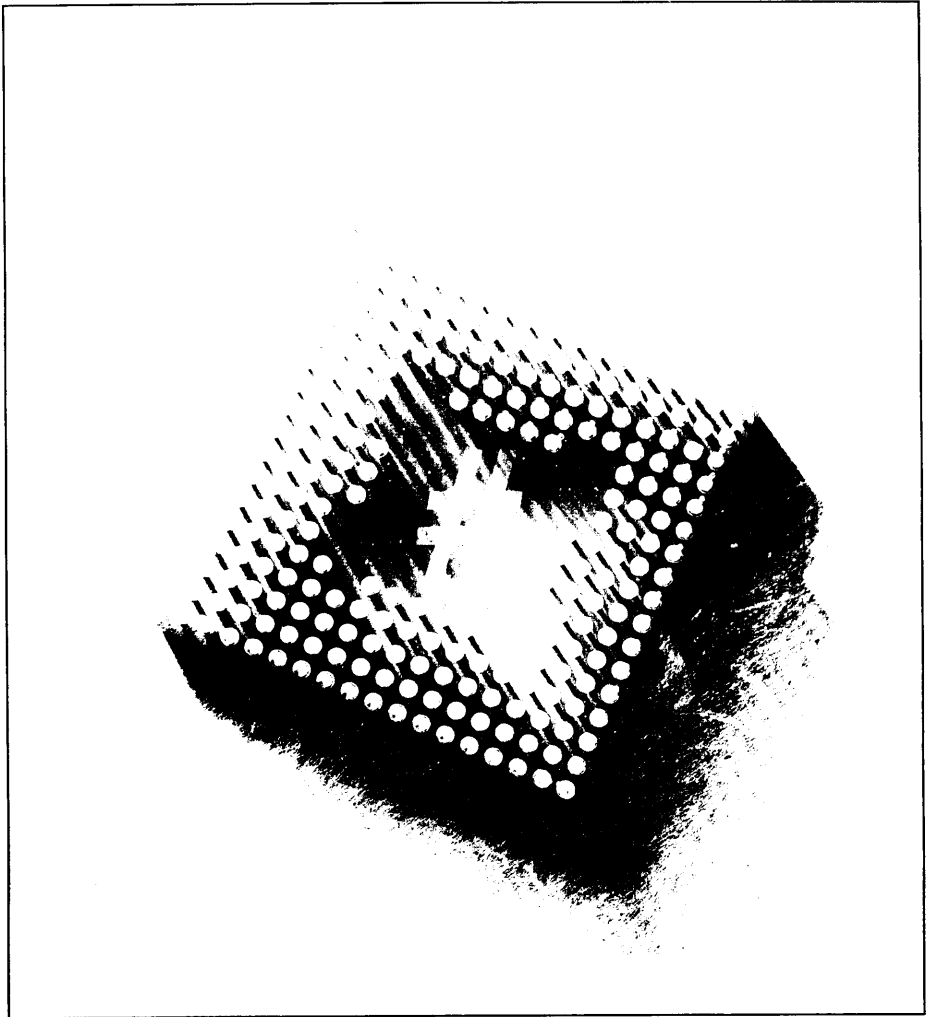
Resumen

En este capítulo, hemos estudiado los diferentes rasgos de Mobile Internet Toolkit, que le permite implementar contenido Web en dispositivos móviles. Hemos construido varias aplicaciones que demuestran cómo los controles Web móviles cambian dinámicamente su modo de presentación en tiempo de ejecución, dependiendo del dispositivo móvil en el que se estén ejecutando. También hemos repasado los modos de detectar las funciones del dispositivo para aprovechar las características de varios dispositivos móviles.

Aunque el capítulo sólo ha explicado brevemente algunas de estas características, ya dispone de un excelente punto de partida para crear contenido Web muy dinámico para ser implementado en dispositivos móviles.

Parte V

C# y .NET Framework



31 Cómo trabajar con ensamblados

El código creado para aprovechar las ventajas de .NET Framework se compila en una unidad de empaquetado llamada *ensamblado*. Los ensamblados son el corazón de la implementación del código y una estrategia de seguridad para .NET Framework, de modo que es importante entender lo que son y su modo de comportamiento.

En este capítulo, estudiaremos los ensamblados y cómo escribir código C# para trabajar con información en ellos. .NET Framework contiene una clase llamada `Assembly` que hace que trabajar con ensamblados sea algo muy sencillo y este capítulo le presenta los entresijos de la clase `Assembly`.

Ensamblados

Los ensamblados pueden contener código, recursos o una combinación de ambos. El código de un ensamblado debe contener las instrucciones reales del Lenguaje intermedio de Microsoft (MSIL) que pueden ser ejecutadas por el Entorno común de ejecución (CLR) y un *manifiesto* que describa el contenido del código. Los manifiestos contienen tipos y más información descriptiva que describen el código al CLR. Los ensamblados también forman los límites del código encerrado entre ellos. Los ensamblados forman los límites de tipos, en los que cualquier tipo

que pueda usarse en cualquier código .NET procede de un sólo ensamblado y los tipos con el mismo nombre procedentes de diferentes ensamblados son, de hecho, tipos diferentes. Los ensamblados también forman un límite de seguridad, por medio del cual todo el código del ensamblado tiene el mismo conjunto de información de seguridad, restricciones y permisos.

Los ensamblados se empaquetan mediante el archivo ejecutable portable de Win32 y pueden ser empaquetados como DLL o EXE. Cualquier código generado por un compilador compatible con el CLR y compilado en un ejecutable de consola, un ejecutable Windows o una biblioteca es empaquetado en un ensamblado. Este paquete forma una unidad de implementación para un conjunto de tipos de un ensamblado. No piense que sólo se considera ensamblado al código .NET basado en DLL. Cualquier paquete de código .NET, recurso o metadato que tenga como objetivo un ejecutable o una biblioteca es un ensamblado, incluso si el paquete tiene la forma de un ejecutable. Las aplicaciones WinForms, por ejemplo, son ensamblados .NET válidos, igual que las bibliotecas de clase basadas en DLL. Tenga en cuenta que el compilador de C# también puede generar módulos, pero esos módulos no son ensamblados. Los módulos son fragmentos de código (y probablemente recursos) compilados que se convertirán en un ensamblado más tarde. Los módulos pueden contener MSIL y metadatos que describen los tipos que se encontraron en el módulo, pero no contienen un manifiesto. El CLR no puede abrir ni ejecutar módulos y por tanto, éstos no pueden ser considerados ensamblados.

Cómo encontrar ensamblados cargados

Empezaremos a estudiar el concepto de ensamblado escribiendo una pequeña aplicación de consola que escribe un poco de información sobre los ensamblados cargados en un proceso. Si la información de tipo procede de los ensamblados, el CLR debe cargar la información del ensamblado en el espacio de proceso de un fragmento de código .NET que se esté ejecutando. Por cada tipo al que se hace referencia en una aplicación, el CLR debe recuperar información del ensamblado que contiene el tipo, de modo que pueda usar el tipo adecuadamente. Estos ensamblados reciben el nombre de *ensamblados referenciados*, porque otro ensamblado .NET les hace referencia.

La operación de descubrir la lista de ensamblados a los que se hace referencia es un proceso sencillo. Observe la sencilla aplicación de consola del listado 31.1.

Listado 31.1. Cómo recuperar una lista de ensamblados referenciados

```
using System;
using System.Reflection;

public class MainClass
{
```

```

static void Main()
{
    Assembly EntryAssembly;

    EntryAssembly = Assembly.GetEntryAssembly();
    foreach(AssemblyName Name in
EntryAssembly.GetReferencedAssemblies())
        Console.WriteLine("Name: {0}", Name.ToString());
}
}

```

El listado 31.1 presenta muchos conceptos importantes. En primer lugar, presenta una clase .NET llamada *Assembly*, que se incluye en el espacio de nombres *System.Reflection* de .NET. La clase *Assembly* es la clase mediante la cual cualquier código .NET puede examinar y trabajar con los contenidos de un ensamblado .NET. Si necesita trabajar con un ensamblado .NET, debe usar la clase *Assembly* para examinar los contenidos del ensamblado.

El segundo concepto importante del listado 31.1 es el de *ensamblado de entrada*. El ensamblado de entrada es el ensamblado que se ejecuta en primer lugar en el proceso. Para ejecutables, como el ejecutable de consola creado por el listado 31.1, el ensamblado de entrada es el ensamblado que contiene el punto de entrada de la función. Normalmente, el punto de entrada recibe el nombre *Main()* en los ensamblados basados en ejecutables, aunque en C# se puede cambiar mediante el argumento */main* y especificando otro punto de entrada para el ensamblado. El acceso al ensamblado de entrada se realiza mediante un método estático de la clase *Assembly* llamado *GetEntryAssembly()*. Este método devuelve una instancia de un objeto *Assembly* que hace referencia al ensamblado de entrada.

El tercer concepto importante del listado 31.1 es el de que un ensamblado puede contener ensamblados a los que se hace referencia. La información de los ensamblados a los que se hace referencia se obtiene mediante una llamada a un método de ensamblado llamado *GetReferencedAssemblies()*. Este método devuelve una matriz de objetos de una clase llamada *AssemblyName*. Los objetos de *AssemblyName* describen completamente el nombre de un ensamblado o se pueden convertir en una sencilla cadena mediante el conocido método *ToString()*. Obtener una representación de cadena de un nombre de ensamblado facilita el que las aplicaciones muestren información del nombre del ensamblado en las interfaces de usuario.

Con esta información, resulta sencillo adivinar lo que hace el listado 31.1. El código obtiene una referencia al ensamblado de entrada y envía los nombres de los ensamblados a los que se hace referencia a la consola. Si se compila y ejecuta el código del listado 31.1, se envía la siguiente información a la consola:

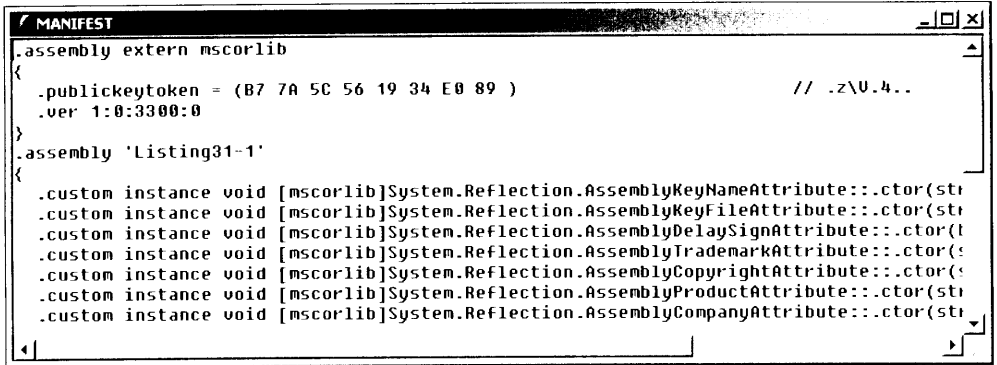
```

Name: mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089

```

¿Cómo sabe .NET Framework que el listado 31.1 hace referencia al ensamblado *mscorlib*? Esa información se almacena en el manifiesto del ensamblado

del Listado 31.1. Para ver esta información, inicie la herramienta ILDASM de .NET Framework y abra el ensamblado del listado 31.1 en ella. Haga doble clic en la entrada del manifiesto en el árbol que aparecerá y se mostrará el manifiesto para el ensamblado en una ventana distinta, como muestra la figura 31.1.



```
MANIFEST
{
  .assembly extern mscorlib
  {
    .publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\0.4..
    .ver 1:0:3300:0
  }
  .assembly 'Listing31-1'
  {
    .custom instance void [mscorlib]System.Reflection.AssemblyKeyNameAttribute::.ctor(st
    .custom instance void [mscorlib]System.Reflection.AssemblyKeyFileAttribute::.ctor(st
    .custom instance void [mscorlib]System.Reflection.AssemblyDelaySignAttribute::.ctor(t
    .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(s
    .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(s
    .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(st
    .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(st
```

Figura 31.1. Referencias a un ensamblado externo de un manifiesto

El manifiesto contiene una entrada con la etiqueta `.assembly extern`. Esa entrada del manifiesto describe un ensamblado externo del que depende el ensamblado que contiene el manifiesto. Esta entrada registra que el ensamblado que contiene este manifiesto depende de la versión 1.0.3300.0 de un ensamblado externo llamado `mscorlib`. .NET Framework debe leer este manifiesto y cargar los ensamblados dependientes en el espacio de proceso que se esté ejecutando en ese momento.

NOTA: El ensamblado `mscorlib` contiene información vital para clases como `System.Object` y siempre se la hace referencia sin ningún argumento especial de compilados. El resto de ensamblados pueden referenciarse usando la opción `/r` para el compilador de líneas de comando de C# o con el elemento de menú Agregar referencias de Visual Studio .NET.

Nombres seguros de ensamblado

El resultado del listado 31.1 puede parecer un poco confuso al principio, ya que muestra algo más que el simple nombre del ensamblado, que en este caso se llama `mscorlib`.

Este resultado en realidad define cuatro fragmentos de información para el nombre del ensamblado:

- El nombre en sí (`mscorlib`)
- Un número de versión (1.0.3300.0)

- Información de referencia cultural (*neutral*)
- Un símbolo de clave pública (*b77a5c561934e089*)

Como mínimo, todos los ensamblados contienen un nombre, una versión y una referencia cultural. Sin embargo, los ensamblados pueden contener una clave pública. Un ensamblado que contiene los cuatro fragmentos de información tiene un nombre seguro.

Sólo los que contienen nombres seguros pueden almacenarse en la caché de ensamblados local. La caché de ensamblados global (GAC) es una colección basada en disco de ensamblados .NET a los que puede acceder cualquier fragmento de código .NET del equipo que contiene el GAC. .NET Framework busca un ensamblado en el directorio del ensamblado de entrada cuando debe cargar un ensamblado. Este esquema de implementación es sencillo; sin embargo, puede crear varias copias de un ensamblado en un volumen de disco para los ensamblados más utilizados, ya que cada ensamblado necesita ser copiado en cada ensamblado de entrada que necesite el ensamblado al que se hace referencia. .NET Framework incluye la GAC para simplificar las cosas, de modo que los ensamblados más utilizados, como los ensamblados incluidos en .NET Framework, pueden ser colocados en un equipo una vez y ser referenciados varias veces. .NET Framework comprueba la GAC cuando busca un ensamblado.

Cuando .NET Framework se instala en un equipo, el proceso de configuración instala una extensión de intérprete de comandos del Explorador de Windows que hace que la GAC aparezca como una carpeta de Windows estándar. El directorio base de Windows, que es `C:\WINDOWS` en casi todos los equipos, contiene un ensamblado de llamada a carpetas en los equipos que tienen instalado .NET Framework. Esta carpeta muestra los contenidos de la GAC, con información de nombre seguro en las columnas, como muestra la figura 31.2.

Puede colocar ensamblados con diferentes nombres seguros uno junto al otro en la caché de ensamblados global, aunque los nombres de segmento coincidan. Por ejemplo, la versión 1.0.0.0 de un ensamblado llamado `assembly.dll` puede estar instalada en la caché de ensamblados global junto a la versión 2.0.0.0 de un ensamblado también llamado `assembly.dll`. El código que hace referencia a un ensamblado con nombre seguro tiene el nombre seguro del ensamblado escrito en el manifiesto y siempre se enlaza al ensamblado con ese nombre seguro, aunque otros ensamblados con algunos componentes del nombre seguro sean iguales. Si un ensamblado de entrada hace referencia a la versión 1.0.0.0 de `assembly.dll`, por ejemplo, .NET Framework siempre carga la versión 1.0.0.0 de `assembly.dll` en el espacio de proceso del ensamblado de entrada, aunque existan otras versiones de `assembly.dll` en la GAC. La operación de asignar la información que compone un nombre seguro es tan sencilla como agregar algunos atributos a uno de los archivos de código fuente para un proyecto. Estos atributos pueden agregarse a un archivo fuente que contenga código C# o pueden agregarse a un archivo distinto que sólo contenga los atributos en sí.

Esta sintaxis indica al compilador de C# que debe asignar el número de revisión que prefiera al ensamblado. El compilador de C# calcula el número de segundos entre la medianoche y la hora en que compiló su código, divide el número entre dos y usa el resto de la división como base para generar un número de revisión único. (A esto se le llama operación *módulo 2*, ya que una operación de módulo calcula el resto de la división entre los dos operandos.) Esto permite generar un número de versión único para cada compilación.

Para facilitar aún más las cosas, el compilador de C# genera automáticamente un número de compilación y un número de revisión si se usa un asterisco como número de compilación:

```
[assembly: AssemblyVersion("1.0.*")]
```

Esta sintaxis indica al compilador de C# que debe asignar el número de compilación y el número de revisión que prefiera al ensamblado. Además del cálculo automático del número de revisión descrito, el compilador de C# también calcula un número de compilación usando el número de días entre el 1 de enero del 2000 y el día en que se produjo la compilación.

Cómo asignar la información de referencia cultural

La información de referencia cultural especifica la referencia cultural para la que está diseñado el ensamblado. La información de referencia cultural se especifica mediante un atributo llamado `AssemblyCulture`. El atributo de referencia cultural recibe una cadena que describe la referencia cultural para la que se diseñó el ensamblado. La cadena puede especificarse mediante una cadena vacía que informa a .NET Framework de que la referencia cultural del ensamblado es neutral y no contiene código o recursos de una cultura específica:

```
[assembly: AssemblyCulture("")]
```

La cadena también puede especificar el idioma y el país para los que se ha diseñado el ensamblado. El formato de la cadena que especifica la información del idioma y el país debe estar registrado en Internet RFC 1766, que tiene la forma idioma-país:

```
[assembly: AssemblyCulture("en-US")]
```

TRUCO: El estándar RFC 1766 recibe el nombre de "Etiquetas para la identificación de idiomas" y está disponible en la dirección de Internet en <http://www.ietf.org/rfc/rfc1766.txt>.

Sólo los ensamblados basados en DLL deben especificar la información de referencia cultural. Los ensamblados basados en EXE deben usar una cadena vacía para la información de referencia cultural. Si se especifica información de

referencia cultural para un ensamblado basado en EXE se produce un error del compilador de C#.

Cómo asignar la información de clave

.NET Framework incluye una utilidad de línea de comando llamada la utilidad nombre seguro o, para abreviar, `sn`, que ayuda a crear nombres seguros para los ensamblados .NET. Una de sus características más usadas permite la creación de un nuevo conjunto de claves de firma digital que pueden instalarse en un ensamblado y ser usadas como parte de un nombre seguro en un ensamblado. Las claves están escritas en un archivo binario cuyo nombre se especifica a la utilidad `sn` mediante el argumento `-k`, como se puede apreciar en la siguiente línea de comando:

```
sn -k KeyPair.snk
```

Esta línea de comando indica a la utilidad `sn` que debe generar un nuevo par de claves de firma digital y enviar las claves a un archivo binario llamado `KeyPair.snk`. Este archivo es nombrado como un argumento para un atributo llamado `AssemblyKeyFile`:

```
[assembly: AssemblyKeyFile("KeyPair.snk")]
```

La extensión `.snk` no es imprescindible. Se puede usar cualquier extensión para el archivo clave generado por la utilidad nombre seguro.

Cómo trabajar con la clase `Assembly`

Una vez que conoce la clase `Assembly`, puede estudiarla más atentamente. Las siguientes secciones describen cómo puede usar sus propiedades y métodos.

Cómo encontrar la información de ubicación del ensamblado

La clase `Assembly` contiene propiedades que describen la ubicación de un ensamblado. Una propiedad `Location` especifica la ubicación del archivo que contiene el manifiesto para el ensamblado. Una propiedad `CodeBase` especifica la ubicación de un ensamblado como un Identificador de recursos uniforme (URI). La propiedad relacionada `EscapedCodeBase` especifica el URI del ensamblado, con caracteres especiales reemplazados por los códigos de escape equivalentes (por ejemplo, los espacios en los valores `CodeBase` se sustituyen por la secuencia de escape `%20` en la propiedad `EscapedCodeBase`). La propiedad `GlobalAssemblyCache` es un booleano que devuelve `True` si el ensamblado se ha cargado desde la GAC y `False` en caso contrario.

El listado 31.2 muestra una sencilla aplicación de consola que obtiene una referencia para el ensamblado de entrada de la aplicación y envía su información de ubicación a la consola.

Listado 31.2. Cómo examinar la información de ubicación

```
using System;
using System.Reflection;

public class MainClass
{
    static void Main()
    {
        Assembly EntryAssembly;

        EntryAssembly = Assembly.GetEntryAssembly();
        Console.WriteLine("Location: {0}",
EntryAssembly.Location);
        Console.WriteLine("Code Base: {0}",
EntryAssembly.CodeBase);
        Console.WriteLine("Escaped Code Base: {0}",
EntryAssembly.EscapedCodeBase);
        Console.WriteLine("Loaded from GAC: {0}",
EntryAssembly.GlobalAssemblyCache);
    }
}
```

Si se compila y ejecuta el código del listado 31.2 se envía la siguiente información a la consola:

- **Location:** C:\Documents and Settings\User\My Documents\Listing31-2\bin\Debug\Listing31-2.exe
- **Code Base:** file:///C:/Documents and Settings/User/My Documents/Listing31-2/bin/Debug/Listing31-2.exe
- **Escaped Code Base:** file:///C:/Documents and Settings/User/My Documents/Listing31-2/bin/Debug/Listing31-2.exe
- **Loaded from GAC:** False

La información de la ruta varía según la ubicación real del código cuando se ejecuta, pero los resultados son básicamente los mismos: La propiedad `Location` hace referencia a una posición en el disco y las propiedades `CodeBase` y `EscapedCodeBase` hacen referencia a la ubicación del ensamblado como URI.

Cómo encontrar puntos de entrada del ensamblado

Algunos ensamblados tienen *puntos de entrada*. Imagine un punto de entrada como el "método inicial" de un ensamblado. El ejemplo más obvio de punto de entrada de un ensamblado es el método `Main()` de los ejecutables basados en C#. El CLR carga un ejecutable, busca un punto de entrada al ensamblado y empieza a ejecutarse con ese método de punto de entrada.

Los ensamblados basados en DLL, en cambio, no suelen tener puntos de entrada. Estos ensamblados generalmente contienen recursos o tipos usados por otros fragmentos de código y son pasivos, en el sentido de que esperan a ser llamados antes de que se ejecute algún código del ensamblado.

La clase `Assembly` contiene una propiedad llamada `EntryPoint`. La propiedad `EntryPoint` es un valor de un tipo llamado `MethodInfo`, que se incluye en el espacio de nombres `System.Reflection`.NET. La clase `MethodInfo` describe los detalles específicos de un método y al llamar a `ToString()` sobre un objeto de tipo `MethodInfo` se devuelve una cadena que describe el tipo devuelto, el nombre y los parámetros del método. La propiedad `EntryPoint` es nula si la referencia del ensamblado no tiene un punto de entrada o un objeto `MethodInfo` válido si la referencia del ensamblado tiene un punto de entrada, como se aprecia en el listado 31.3.

Listado 31.3. Cómo trabajar con un punto de entrada de un ensamblado

```
using System;
using System.Reflection;

public class MainClass
{
    static void Main(string[] args)
    {
        Assembly EntryAssembly;

        EntryAssembly = Assembly.GetEntryAssembly();
        if(EntryAssembly.EntryPoint == null)
            Console.WriteLine("The assembly has no entry point.");
        else
            Console.WriteLine(EntryAssembly.EntryPoint.ToString());
    }
}
```

Si se compila y ejecuta el listado 31.3 se envía la siguiente información a la consola:

```
Void Main(System.String[])
```

En el sencillo ejemplo del listado 31.3, la propiedad `EntryPoint` nunca es nula, pero siempre conviene comprobar la posibilidad de que se produzca un valor nulo, especialmente con fragmentos de código más complicados.

Cómo cargar ensamblados

En muchas aplicaciones, a los ensamblados que contienen los tipos necesarios para una aplicación se les hace referencia cuando se crea la aplicación. Sin embargo, también es posible cargar los ensamblados mediante programación. Hay varios modos de cargar un ensamblado dinámicamente y cada una de estas técni-

cas de carga devuelve un objeto `Assembly` que hace referencia a un ensamblado cargado.

La primera técnica de carga de ensamblado usa un método de ensamblado estático llamado `Load()`. El método `Load()` recibe una cadena que proporciona el nombre del ensamblado que se va a cargar. Si no se encuentra el ensamblado nombrado, el método `Load()` inicia una excepción. En cambio, el método `LoadWithPartialName()` busca el directorio de la aplicación y la GAC del ensamblado especificado, usando toda la información de nombre disponible para el invocador. El listado 31.4 muestra la diferencia entre estos dos métodos.

Listado 31.4. Cómo cargar ensamblados dinámicamente con `Load()` y `LoadWithPartialName()`

```
using System;
using System.Reflection;
using System.IO;

public class AssemblyLoader
{
    private Assembly LoadedAssembly;

    public AssemblyLoader(string LoadedAssemblyName, bool
PartialName)
    {
        try
        {
            Console.WriteLine("+-----");
            Console.WriteLine("| Loading Assembly {0}",
LoadedAssemblyName);
            Console.WriteLine("+-----");
            if(PartialName == true)
                LoadedAssembly =
Assembly.LoadWithPartialName(LoadedAssemblyName);
            else
                LoadedAssembly = Assembly.Load(LoadedAssemblyName);
            WritePropertiesToConsole();
        }
        catch(FileNotFoundException)
        {
            Console.WriteLine("EXCEPTION: Cannot load assembly.");
        }
    }

    private void WritePropertiesToConsole()
    {
        Console.WriteLine("Full Name: {0}",
LoadedAssembly.FullName);
        Console.WriteLine("Location: {0}",
LoadedAssembly.Location);
        Console.WriteLine("Code Base: {0}",
LoadedAssembly.CodeBase);
    }
}
```

```

        Console.WriteLine("Escaped Code Base: {0}",
LoadedAssembly.EscapedCodeBase);
        Console.WriteLine("Loaded from GAC: {0}",
LoadedAssembly.GlobalAssemblyCache);
    }
}

public class MainClass
{
    static void Main(string[] args)
    {
        AssemblyLoader Loader;

        Loader = new AssemblyLoader("System.Xml,
Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089", false);
        Loader = new AssemblyLoader("System.Xml", false);
        Loader = new AssemblyLoader("System.Xml", true);
    }
}

```

El listado 31.4 muestra una clase llamada `AssemblyLoader`. cuyo constructor recibe un nombre de ensamblado y un indicador booleano que especifica si el ensamblado nombrado debe cargarse usando un nombre parcial. El constructor carga el ensamblado y a continuación llama a un método privado para escribir algunas de las propiedades de designación y de ubicación del ensamblado cargado en la consola.

El método `Main()` del listado 31.4 crea nuevos objetos de la clase `AssemblyLoader` e intenta cargar el ensamblado de .NET Framework `System.XML`. que se encuentra en la GAC. de varios modos.

Si se ejecuta el listado 31.4 se escribe la siguiente información en la consola:

```

+-----+
| Loading Assembly System.Xml, Version=1.0.3300.0,
Culture=neutral, PublicKeyTok
en=b77a5c561934e089
+-----+
Full Name: System.Xml, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5
c561934e089
Location: c:\windows\assembly\gac\system.xml\1.0.3300.0__
b77a5c561934e089\system
.xml.dll
Code Base: file:///c:/windows/assembly/gac/system.xml/
1.0.3300.0__b77a5c561934e0
89/system.xml.dll
Escaped Code Base: file:///c:/windows/assembly/gac/system.xml/
1.0.3300.0__b77a5c
561934e089/system.xml.dll
Loaded from GAC: True
+-----+

```

```

| Loading Assembly System.Xml
+-----+
EXCEPTION: Cannot load assembly.
+-----+
| Loading Assembly System.Xml
+-----+
Full Name: System.Xml, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c5e1934e089
Location: c:\windows\assembly\gac\system.xml\1.0.3300.0_...
b77a5c5e1934e089\system
.xml.dll
Code Base: file:///c:/windows/assembly/gac/system.xml/
1.0.3300.0_b77a5c5e1934e0
89/system.xml.dll
Escaped Code Base: file:///c:/windows/assembly/gac/system.xml/
1.0.3300.0_b77a5c
5e1934e089/system.xml.dll
Loaded from GAC: True

```

Observe con atención lo que se está produciendo. En el segundo caso, el método `Main()` especifica el nombre seguro para el ensamblado `System.Xml`, incluyendo su nombre, clave pública, información de versión y detalles específicos de su referencia cultural. Como el ensamblado `System.Xml` está en la GAC, no se almacena en el directorio de la aplicación y el método `Load()` no puede encontrar el ensamblado en el directorio que contiene el ejecutable del listado 31.4. Sin embargo, como se especificó el nombre seguro para el ensamblado, el método `Load()` tiene suficiente información para buscar el ensamblado en la GAC. El método `Load()` puede encontrar el ensamblado en la GAC y la operación de carga se completa con éxito. En el segundo caso, el método `Main()` sólo especifica el nombre base del ensamblado `System.Xml`. Como el ensamblado `System.Xml` está en la GAC, no se almacena en el directorio de la aplicación y el método `Load()` no puede encontrar el ensamblado en el directorio que contiene el ejecutable del listado 31.4. Además, el método `Load()` no tiene suficiente información para ubicar el ensamblado en la GAC, ya que pueden existir muchas instancias de `System.Xml` en la GAC con diferentes números de versión o claves públicas, por lo que la carga falla.

En el tercer y último caso, el método `Main()` sólo especifica el nombre base del ensamblado `System.Xml` e indica al cargador que encuentre un ensamblado usando sólo un nombre parcial. De nuevo, como el ensamblado `System.Xml` está en la GAC, no se almacena en el directorio de la aplicación y el método `LoadWithPartialName()` no puede encontrar el ensamblado en el directorio que contiene el ejecutable del listado 31.4. Sin embargo, el método `LoadWithPartialName()` recibe el nombre parcialmente proporcionado e intenta hacer coincidir el nombre con un ensamblado de la GAC. Como se ha proporcionado un nombre parcial de `System.Xml` y hay un ensamblado con el nombre `System.Xml` en la GAC, la operación de carga se completa con éxito.

ADVERTENCIA: No se recomienda utilizar `LoadWithPartialName()`. Si el ensamblado parcialmente nombrado tiene varias copias en la GAC (quizás con diferentes números de versión, referencias culturales o claves públicas) la instancia realmente cargada puede no ser la versión esperada. Además, la instancia cargada puede ser una versión diferente de la que pretendía cargar después de que se hayan cargado versiones más modernas en la GAC. Use `Load()` en lugar de `LoadWithPartialName()` siempre que pueda.

Cómo trabajar con información de tipo de ensamblado

Los ensamblados pueden contener tipos, recursos o una combinación de ambos. Tras cargar un ensamblado, se puede obtener información sobre los tipos que se encuentran en el ensamblado. Además, se pueden crear instancias de los tipos mediante programación. El listado 31.5 muestra estos conceptos.

Listado 31.5. Cómo encontrar y crear tipos de ensamblado

```
using System;
using System.Reflection;

public class MainClass
{
    static void Main(string[] args)
    {
        Assembly XMLAssembly;
        Type[] XMLTypes;

        XMLAssembly = Assembly.Load("System.Xml,
Version-1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c5a1934e089");
        XMLTypes = XMLAssembly.GetExportedTypes();
        foreach(Type XMLType in XMLTypes)
        {
            object NewObject;
            try
            {
                Console.WriteLine(XMLType.ToString());
                NewObject =
XMLAssembly.CreateInstance(XMLType.ToString());
                if(NewObject != null)
                    Console.WriteLine("- Creation successful");
                else
                    Console.WriteLine("- CREATION ERROR");
            }
            catch(Exception e)
            {
                Console.WriteLine("- EXCEPTION: {0}", e.Message);
            }
        }
    }
}
```

```
}  
}  
}
```

El código del listado 31.5 carga el ensamblado `System.Xml` desde la GAC y llama a un método en la clase `Assembly` llamado `GetExportedTypes()` para conseguir una matriz de objetos de tipo que representan los tipos que se encuentran en el ensamblado y pueden ser utilizados o exportados al exterior del ensamblado. El código recorre cada tipo de la matriz devuelta e invoca a otro método de ensamblado llamado `CreateInstance()` para crear una instancia de objeto del tipo nombrado. Si la creación se realiza con éxito, el método `CreateInstance()` devuelve una referencia de objeto válida. Si la creación no tiene éxito, `CreateInstance()` devuelve una referencia de objeto nula o inicia una excepción, dependiendo de la naturaleza del error.

A continuación tiene varias líneas de resultado del listado 31.5:

```
System.Xml.XPath.XPathNavigator - EXCEPTION: Constructor on  
type System.Xml.XPath.XPathNavigator not found.  
System.Xml.IHasXmlNode - EXCEPTION: Constructor on type  
System.Xml.IHasXmlNode not found.  
System.Xml.XPath.XPathNodeIterator - EXCEPTION: Constructor on  
type System.Xml.XPath.XPathNodeIterator not found.  
System.Xml.EntityHandling - Creation successful  
System.Xml.IXmlLineInfo - EXCEPTION: Constructor on type  
System.Xml.IXmlLineInfo not found.  
System.Xml.XmlNameTable - EXCEPTION: Constructor on type  
System.Xml.XmlNameTable not found.  
System.Xml.NameTable - Creation successful  
System.Xml.ReadState - Creation successful  
System.Xml.ValidationType - Creation successful  
System.Xml.WhitespaceHandling - Creation successful
```

Las excepciones se producen porque no todos los tipos exportados tienen constructores y con `CreateInstance()` sólo se pueden crear tipos de referencia con constructores adecuados.

TRUCO: Tras obtener una referencia a un objeto `Type`, se puede encontrar una referencia al ensamblado que contiene el tipo en la propiedad `Assembly` del objeto `Type`. Esto permite al código descubrir el ensamblado que hace referencia a un tipo.

Cómo generar código nativo para ensamblados

Cuando el CLR necesita ejecutar código en un ensamblado, pasa el código a través de un compilador justo a tiempo (JIT) y devuelve el MSIL a un código

nativo que puede ser ejecutado por la CPU del equipo. La ventaja del diseño JIT es que es posible enviar código MSIL sin tener que preocuparse por optimizar el código para el procesador al que está destinado. .NET Framework probablemente será exportado a una gran variedad de arquitecturas de CPU de una gran variedad de dispositivos, desde equipos de sobremesa hasta sistemas portátiles e intentar escribir código para cada una de estos procesadores podría ser una tarea impresionante.

Este trabajo no es necesario porque cada implementación de .NET Framework tiene un compilador JIT que convierte las instrucciones MSIL en instrucciones óptimas para la CPU de destino.

Si la principal preocupación de su aplicación es el rendimiento, puede convertir el código MSIL en código específico para la CPU de un equipo mediante un proceso conocido como *generación de imagen nativa*. Durante este proceso, las instrucciones MSIL de un ensamblado se convierten en instrucciones específicas de una CPU, que pueden ser escritas en disco. Después de completar esta generación de imágenes nativas, el CLR puede usar este código y puede omitir el paso JIT que normalmente se emplea en los ensamblados.

.NET Framework incluye una herramienta llamada **Generador de imágenes nativas**, que genera una imagen nativa para un ensamblado. Esta herramienta de línea de comando se encuentra en un ejecutable llamado `ngen.exe` y recibe como entrada un nombre de ensamblado:

```
ngen assembly.dll
```

La imagen nativa se coloca en una caché de imágenes nativas para ensamblados. Tenga en cuenta que `ngen` debe ejecutarse en el dispositivo que ejecuta el código generado. Por ejemplo, no puede construir ensamblados como parte de su proceso de compilación, ejecutar `ngen` sobre esos ensamblados y enviar esas imágenes nativas a sus clientes.

El equipo en el que se compila puede perfectamente tener una CPU diferente de la de los equipos de sus clientes y `ngen` genera código para la CPU en la que `ngen` se está ejecutando. Si para sus clientes es importante tener imágenes nativas para sus ensamblados, debe ejecutar `ngen` en los equipos de sus clientes como parte del proceso de instalación.

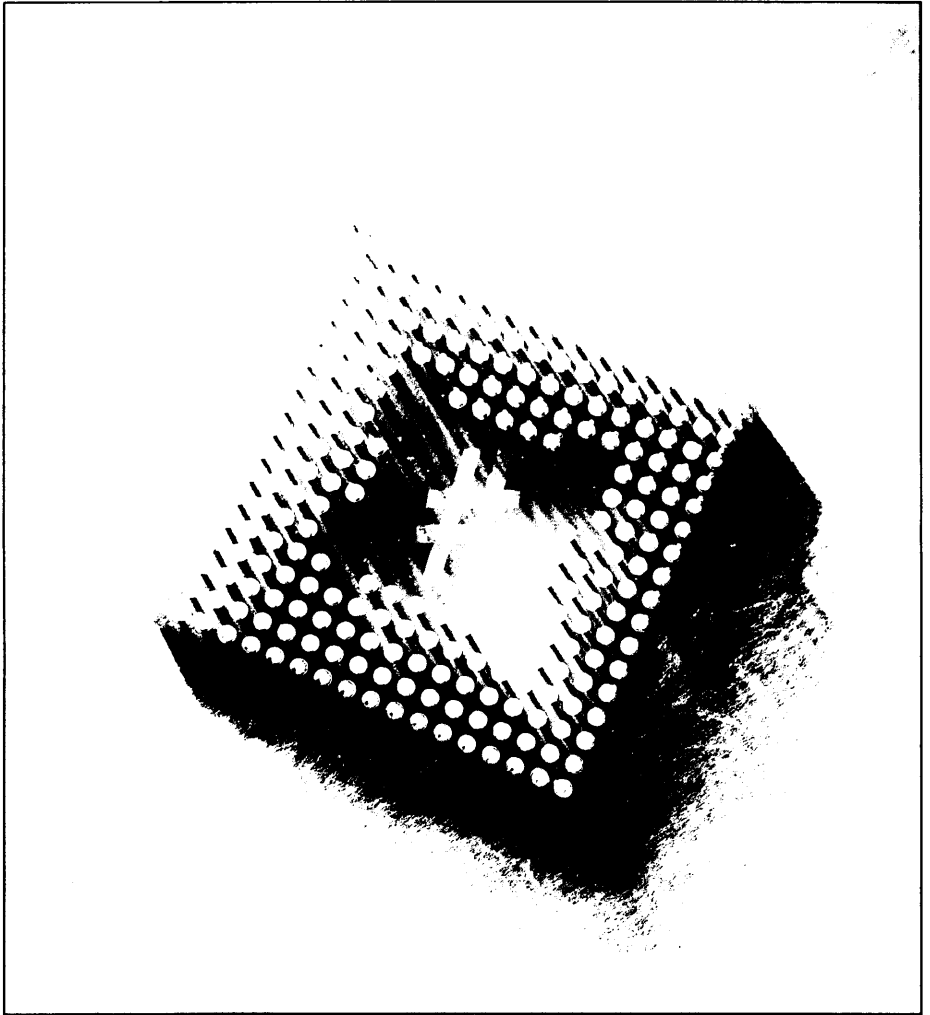
También es importante tener en cuenta que los ensamblados .NET originales deben estar disponibles en todo momento, aunque el código nativo esté disponible en la caché de imagen nativa.

Las imágenes nativas son archivos Ejecutable portables (PE) Win32 estándar y carecen de los metadatos existentes en un ensamblado .NET. Si el código carga su ensamblado de imagen nativa y ejecuta código que obliga a .NET Framework a examinar metadatos (por ejemplo, usando reflexión para obtener información para el ensamblado), entonces el ensamblado .NET original debe estar disponible para que el CLR pueda consultar sus metadatos. Los metadatos no pueden ser transportados junto a la imagen nativa.

Resumen

En este capítulo, se ha examinado el concepto de ensamblado .NET desde la perspectiva de las aplicaciones de C# que pueden acceder a la información de los ensamblados. El acceso a la información de los ensamblados se realiza mediante la clase `Assembly`. La clase `Assembly` muestra la información de nombre para el ensamblado y permite que los ensamblados se carguen dinámicamente. Los tipos gestionados por el ensamblado pueden ser creados al instante.

Puede aplicar los conceptos mostrados en este capítulo en la construcción de potentes aplicaciones .NET. Algunas de las herramientas que incorpora .NET Framework, como la herramienta ILDASM, usan una combinación de métodos de la clase `Assembly` y otras clases en el espacio de nombres `System.Reflection` para proporcionar una vista con todos los detalles de los ensamblados ya compilados. Se puede obtener una gran cantidad de información de los ensamblados, usando los métodos de la clase `Assembly` y otras clases de reflexión, aunque el código fuente usado para construir el ensamblado no esté disponible.



32 Reflexión

Una importante característica de .NET Framework es su capacidad para descubrir información de tipo en tiempo de ejecución. En concreto, puede usar el espacio de nombres `reflection` para ver la información de tipo que contienen los ensamblados que, más tarde, podrá enlazar a objetos e incluso puede usar este espacio de nombres para generar código en tiempo de ejecución. Esta tecnología se extiende a la tecnología de automatización COM, ya conocida por muchos de los lectores.

Como programador, seguramente necesite usar a menudo un objeto sin comprender del todo lo que hace ese objeto. La reflexión permite tomar un objeto y examinar sus propiedades, métodos, eventos, campos y constructores. Como la reflexión gira en torno a `System.Type`, puede examinar un ensamblado y usar métodos, como `GetMethods()` y `GetProperties()`, para devolver información de miembro desde el ensamblado. Con esta información, puede empezar usando el método `MethodInfo()` para devolver listas de parámetros e incluso llamar a métodos en el ensamblado con un método llamado `Invoke`.

En este capítulo, aprenderá a usar el espacio de nombres `Reflection` para examinar objetos en tiempo de ejecución. También aprenderá a enlazar tardíamente los objetos y a usar métodos y propiedades en estos objetos enlazados tardíamente.

La clase Type

La clase `Type` actúa como una ventana al API de reflexión, lo que permite el acceso a metadatos. La clase abstracta `System.Type` representa un tipo del Sistema completo de tipos (CTS). Este sistema completo de tipos es lo que permite examinar objetos en todos los lenguajes de la familia .NET. Como cada objeto usa el mismo entorno, tiempo de ejecución y sistema de tipos, la información de objeto y de tipo se consigue fácilmente.

Una de las mayores ventajas de las clases `Type` es su capacidad para crear objetos dinámicamente y usarlos en tiempo de ejecución.

Cómo recuperar información de tipo

La información de tipo puede ser recuperada de los objetos mediante varios métodos. Las siguientes secciones describen cómo hacer esto de tres maneras diferentes: usando un nombre de tipo, usando un nombre de proceso o especificando un nombre de ensamblado para recuperar la información. Aunque todas estas implementaciones realizan prácticamente la misma tarea, cada una es útil a su manera.

Dependiendo de los requisitos de su aplicación, sólo necesitará usar una de las formas de las funciones recolectoras de tipos.

Cómo recuperar tipos mediante el nombre

Simplemente especificando el nombre de un tipo, puede consultar casi todos los aspectos del objeto. Puede determinar si el objeto es una clase, de qué tipo es su sistema base y muchas otras propiedades.

Para comprobarlo, puede crear una sencilla aplicación para ver algunas propiedades de la clase `System.String`, como muestra el listado 32.1.

Listado 32.1. Cómo consultar información de tipo mediante el nombre

```
using System;
using System.Reflection;

class NameType
{
    public static void Main()
    {
        Type t = Type.GetType("System.String");
        Console.WriteLine("Name : {0}",t.Name);
        Console.WriteLine("Underlying System Type :
{0}",t.UnderlyingSystemType);
        Console.WriteLine("Is Class : {0}",t.IsClass);
    }
}
```

La figura 32.1 indica que `System.String` es el tipo de sistema base y que el objeto es, sin lugar a dudas, una clase.

Se estará preguntando qué utilidad tiene esta información. Imagine que está creando una aplicación que necesita generar instrucciones `insert` para introducir información en SQL Server. Escribir una gran cantidad de información requiere una gran cantidad de tiempo. Usando la reflexión y la clase `Type`, puede examinar el tipo subyacente de cada fragmento de información que quiera insertar en SQL Server y describir estos tipos a un tipo de datos SQL Server válido. Esto simplifica mucho el proceso de generar mediante programación las instrucciones `insert` que necesita.

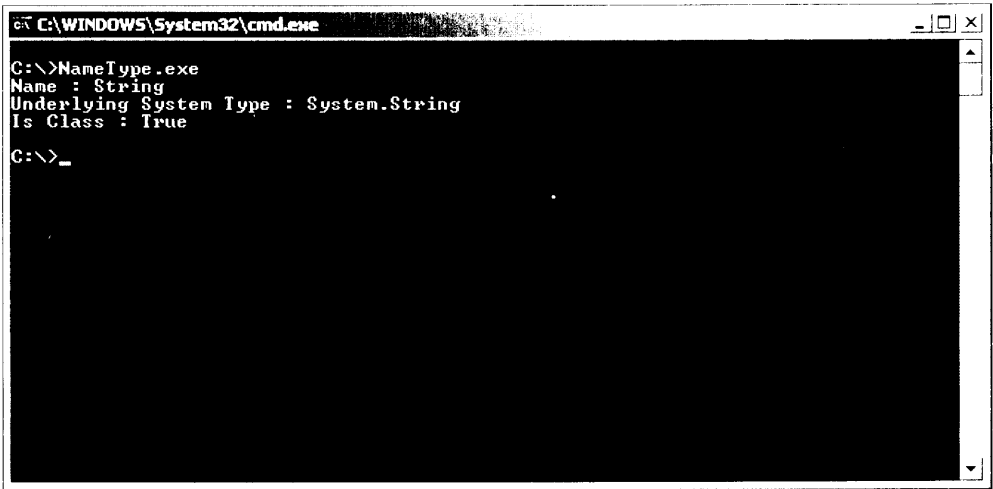


Figura 32.1. Información de tipo de consulta mediante un nombre de objeto

Cómo recuperar tipos mediante instancias

En lugar de usar el nombre de un tipo, puede simplemente usar una instancia de un objeto que quiera examinar. El listado 32.2 representa un ejemplo igual que el anterior.

Listado 32.2. Información de tipo de consulta usando una instancia de un objeto

```
using System;
using System.Reflection;

class InstanceType
{
    public static void Main()
    {
        String myVar = "Brian Patterson";
        Type t = myVar.GetType();
        Console.WriteLine("Name : {0}",t.Name);
    }
}
```

```

        Console.WriteLine("Underlying System Type :
{0}", t.UnderlyingSystemType);
        Console.WriteLine("Is Class : {0}", t.IsClass);
    }
}

```

En este ejemplo, en lugar de especificar que quiere ver la información de tipo de `System.String`, se crea una instancia de una variable de cadena sobre la que, a continuación, llama al método `GetType()`. La información obtenida aquí es la misma que la que se obtuvo en el ejemplo anterior, la diferencia está en que no tiene que saber el tipo antes de tiempo. Simplemente llame al método `GetType()` y asígnelo a un objeto `Type`, al que puede consultar el nombre, el tipo de sistema subyacente y similares.

Cómo recuperar tipos en un ensamblado

A menudo, querrá examinar los tipos que contiene un ensamblado. Este ensamblado puede ser un ejecutable o incluso una biblioteca de vínculos dinámicos contenida en el sistema. El listado 32.3 contiene el código necesario para examinar la información de tipo del propio ejecutable.

Listado 32.3. Cómo examinar un proceso en ejecución para obtener información de tipo

```

using System;
using System.Reflection;
using System.Diagnostics;

class AssemType
{
    public static void Main(string[] args)
    {
        Process p = Process.GetCurrentProcess();
        string assemblyName = p.ProcessName + ".exe";
        Console.WriteLine("Examining : {0}", assemblyName);
        Assembly a = Assembly.LoadFrom(assemblyName);

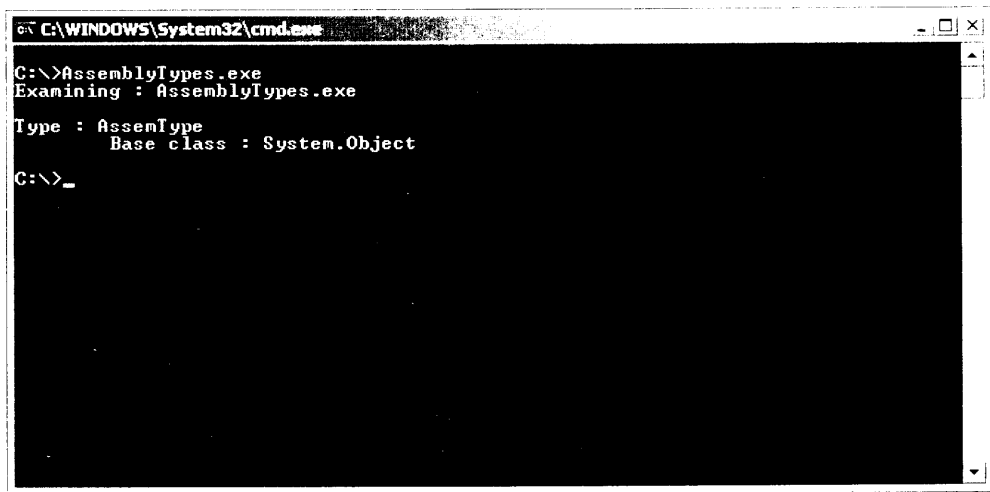
        Type[] types = a.GetTypes();
        foreach (Type t in types)
        {
            Console.WriteLine("\nType : {0}", t.FullName);
            Console.WriteLine("\tBase class :
{0}", t.BaseType.FullName);
        }
    }
}

```

El código anterior presenta algunos elementos nuevos. El tipo `process` se usa para examinar procesos que se están ejecutando. En este contexto, se emplea

para conseguir el nombre de su programa y luego agrega `.exe` al final del nombre para que pueda examinar el ensamblado. Puede igualmente escribir en el código el nombre de su aplicación, pero este método garantiza que funcionará, sin que importe cómo llame a su aplicación.

La figura 32.2 muestra el resultado de esta aplicación. No es un resultado muy espectacular, ya que su programa sólo contiene una clase.



```
C:\WINDOWS\System32\cmd.exe
C:\>AssemblyTypes.exe
Examining : AssemblyTypes.exe
Type : AssemType
Base class : System.Object
C:\>_
```

Figura 32.2. Información de proceso obtenida mediante el API Reflection

Como puede ver en el figura 32.2, hemos examinado el proceso en curso y la misma aplicación que está ejecutando y hemos mostrado todas sus clases internas y sus tipos. Para experimentar, intente agregar algunas clases nulas a este proyecto y luego vuelva a ejecutar la aplicación. Debería ver una lista de todas las clases que allí se contienen y sus tipos.

Cómo interrogar a objetos

El listado 32.4 contiene el código fuente de la aplicación `ReflectionTest`, que examina una clase y proporciona detalles sobre ella. Esta aplicación es un conglomerado de todo lo que ha aprendido sobre la reflexión hasta el momento.

Listado 32.4. Los objetos de clase proporcionan fácilmente la información de miembro

```
namespace ReflectionTest
{
    using System ;
    using System.Reflection ;

    public class Class1
    {
```

```

public static int Main ( )
{
    Type t = typeof( aUsefulClass ) ;
    Console.WriteLine ( "Type of class: " + t ) ;
    Console.WriteLine ( "Namespace: " + t.Namespace ) ;
    ConstructorInfo[] ci = t.GetConstructors( ) ;
        Console.WriteLine("-----
        ");
    Console.WriteLine( "Constructors are:" ) ;

    foreach( ConstructorInfo i in ci )
    {
        Console.WriteLine( i ) ;
    }

    PropertyInfo[] pi = t.GetProperties( ) ;
    Console.WriteLine("-----
    ");
    Console.WriteLine( "Properties are:" ) ;

    foreach( PropertyInfo i in pi )
    {
        Console.WriteLine( i ) ;
    }
    MethodInfo[] mi = t.GetMethods( ) ;
    Console.WriteLine("-----
    ");
    Console.WriteLine( "Methods are:" ) ;

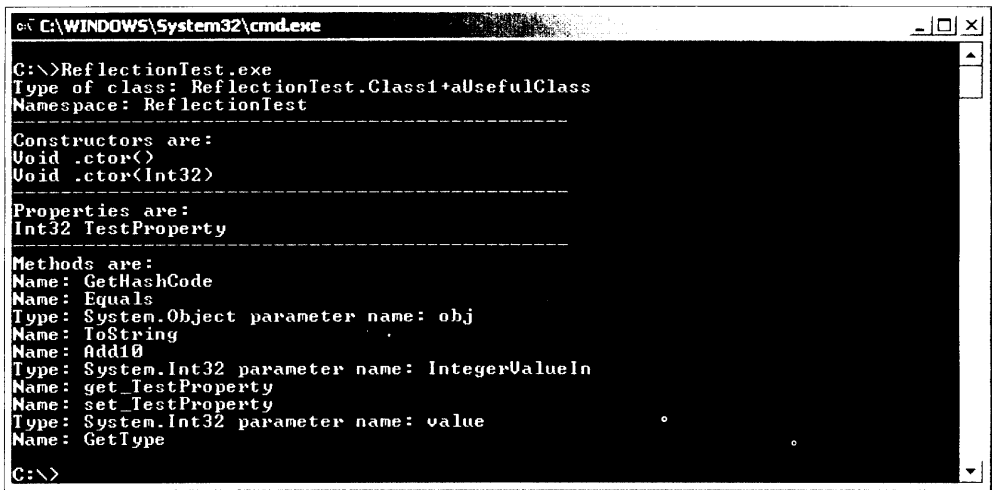
    foreach( MethodInfo i in mi )
    {
        Console.WriteLine( "Name: " + i.Name ) ;
        ParameterInfo[] pif = i.GetParameters ( ) ;
        foreach ( ParameterInfo p in pif )
        {
            Console.WriteLine("Type: " + p.ParameterType + "
parameter name: " + p.Name ) ;
        }
    }
    return 0 ;
}

public class aUsefulClass
{
    public int pubInteger ;
    private int privValue;
    public aUsefulClass()
    {
    }

    public aUsefulClass ( int IntegerValueIn )
    {
        pubInteger = IntegerValueIn ;
    }
}

```


importante aquí es que la reflexión funciona de la misma manera, aunque estemos tratando con un ensamblado para el que no tengamos el código fuente.



```
C:\WINDOWS\System32\cmd.exe
C:\>ReflectionTest.exe
Type of class: ReflectionTest.Class1+UsefulClass
Namespace: ReflectionTest
-----
Constructors are:
Void .ctor()
Void .ctor(Int32)
-----
Properties are:
Int32 TestProperty
-----
Methods are:
Name: GetHashCode
Name: Equals
Type: System.Object parameter name: obj
Name: ToString
Name: Add10
Type: System.Int32 parameter name: IntegerValueIn
Name: get_TestProperty
Name: set_TestProperty
Type: System.Int32 parameter name: value
Name: GetType
C:\>
```

Figura 32.3. Las clases `Reflection` y `Type` revelan una gran cantidad de información relativa al objeto de la clase

Cómo generar código dinámico mediante la reflexión

Puede crear código en tiempo de ejecución usando el espacio de nombres `System.Reflection.Emit`. Al usar las clases de este espacio de nombres, se puede definir un ensamblado en la memoria, crear un módulo, definir nuevos tipos para un módulo (incluyendo sus miembros) y emitir códigos de operación MSIL para lógica de la aplicación.

NOTA: "Opcodes" es la abreviatura de códigos de operación. Éste es el código real que genera el compilador de .NET.

El listado 32.5 contiene el código que puede usarse para generar código en tiempo de ejecución.

Listado 32.5. Cómo generar código dinámicamente en tiempo de ejecución

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DynamicCode
{
```

```

class CodeGenerator
{
    Type t;
    AppDomain currentDomain;
    AssemblyName assemName;
    AssemblyBuilder assemBuilder;
    ModuleBuilder moduleBuilder;
    TypeBuilder typeBuilder;
    MethodBuilder methodBuilder;
    ILGenerator msilG;

    public static void Main()
    {
        CodeGenerator codeGen = new CodeGenerator();
        Type t = codeGen.T;

        if (t != null)
        {
            object o = Activator.CreateInstance(t);
            MethodInfo helloWorld = t.GetMethod("HelloWorld");
            if (helloWorld != null)
            {
                // Ejecuta el metodo HelloWorld
                helloWorld.Invoke(o, null);
            }
        }
        else
        {
            Console.WriteLine("Could not retrieve
MethodInfo");
        }
        else
        {
            Console.WriteLine("Could not access the Type");
        }
    }

    public CodeGenerator()
    {
        // Obtiene el dominio de aplicacion actual.
        // Esto es necesario cuando se construye codigo.
        currentDomain = AppDomain.CurrentDomain;

        // Crea un nuevo ensamblado para nuestros metodos
        assemName = new AssemblyName();
        assemName.Name = "BibleAssembly";

        assemBuilder =
currentDomain.DefineDynamicAssembly(assemName,
AssemblyBuilderAccess.Run);
    }
}

```

```

// Crea un nuevo modulo en este ensamblado
moduleBuilder =
    assembler.DefineDynamicModule("BibleModule");

// Crea un nuevo tipo en el módulo
typeBuilder =
moduleBuilder.DefineType("BibleClass",TypeAttributes.Public);

// Ahora podemos agregar el
// método HelloWorld a la clase recién creada.
methodBuilder =
typeBuilder.DefineMethod("HelloWorld",
MethodAttributes.Public,null,null);

// Ahora podemos generar algo de código de lenguaje
// intermedio de Microsoft que simplemente escriba
// una línea de texto en la consola.
msilG = methodBuilder.GetILGenerator();
msilG.EmitWriteLine("Hello from C# Bible");
msilG.Emit(OpCodes.Ret);
// Crea un tipo.
t = typeBuilder.CreateType();
}

public Type T
{
    get
    {
        return this.t;
    }
}
}
}

```

Como se esperaba, esta aplicación sólo escribe un mensaje en la consola, como muestra la figura 32.4.

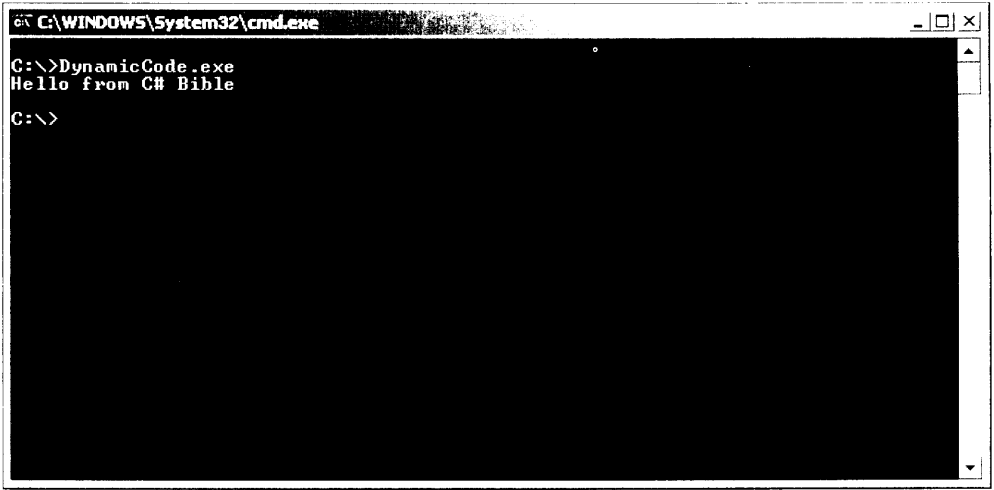
La función dereflexión para generar objetos y código en tiempo de ejecución es realmente impresionante y constituye la columna vertebral para la generación de aplicaciones de lógica difusa que se adaptan y aprenden como resulte apropiado.

NOTA: La lógica difusa es un tipo de álgebra que usa los valores verdadero y falso para tomar decisiones basándose en datos imprecisos. La lógica difusa suele relacionarse con los sistemas de inteligencia artificial.

Resumen

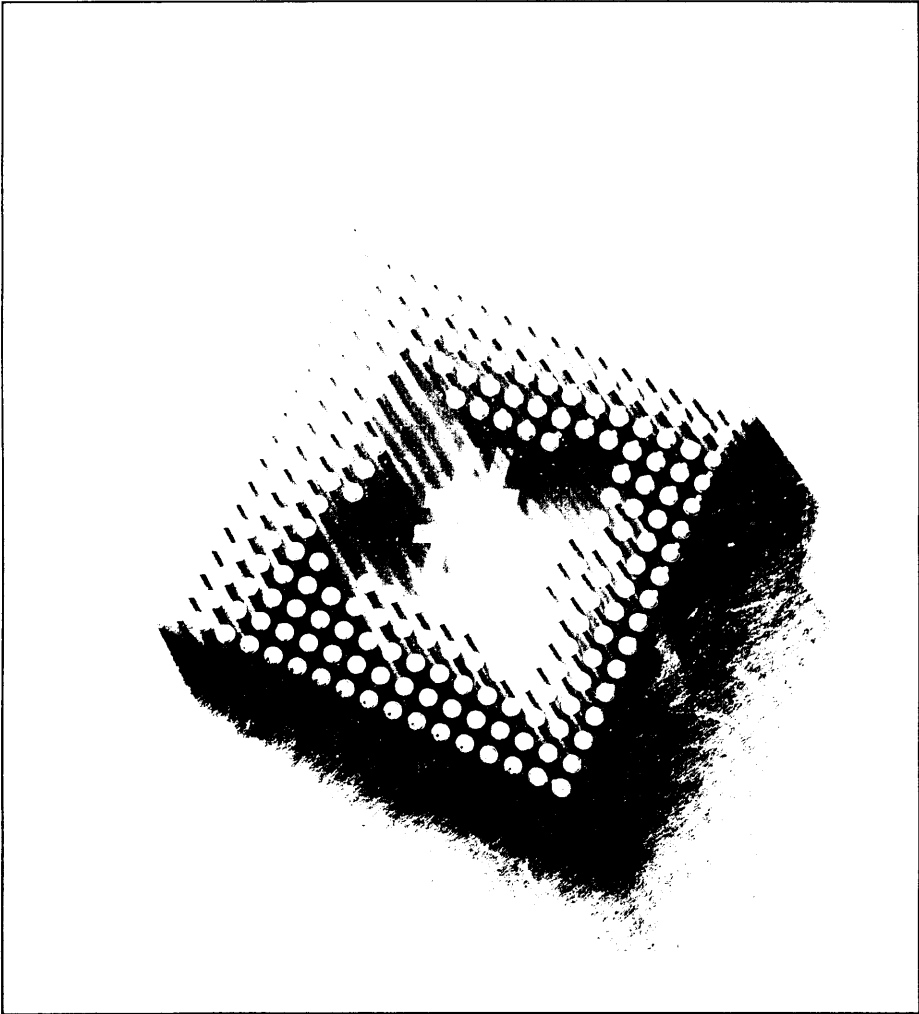
Las clases `Reflection` y `Type` van unidas cuando necesita descubrir información de tipo en tiempo de ejecución. Estas clases permiten examinar obje-

tos. cargar objetos dinámicamente en tiempo de ejecución e incluso generar código en caso de necesidad.



```
C:\WINDOWS\System32\cmd.exe
C:\>DynamicCode.exe
Hello from C# Bible
C:\>
```

Figura 32.4. Los resultados del código generado dinámicamente



Subprocesamiento en C#

La potencia del multiprocesamiento de .NET Framework permite escribir aplicaciones muy estables con varios subprocesos en cualquier lenguaje .NET. En este capítulo aprenderá los entresijos de los multiprocesos. El capítulo comienza con una visión general de los diferentes tipos de subprocesos y de su funcionamiento en .NET Framework, para luego continuar enseñándole lo que puede hacer en sus aplicaciones gracias al multiprocesamiento. A medida que avance en el capítulo, sopesa cuidadosamente los peligros de agregar varios subprocesos a sus aplicaciones antes de implementarlos, porque el multiprocesamiento no es un concepto sencillo.

Subprocesamiento

Antes empezar a escribir aplicaciones con múltiples subprocesos, debe comprender lo que sucede cuando se crean los subprocesos y cómo gestiona el sistema operativo los subprocesos.

Cuando se ejecuta una aplicación, se crea un subproceso primario y el ámbito de la aplicación se basa en ese subproceso. Una aplicación puede crear subprocesos adicionales para realizar tareas adicionales. Un ejemplo de creación de subprocesos primarios sería iniciar Microsoft Word. La ejecución de la aplicación comienza

en el subproceso principal. En la aplicación Word, la impresión en segundo plano de un documento sería un ejemplo de un subproceso adicional creado para controlar otra tarea. Mientras está interactuando con el subproceso principal (el documento Word), el sistema lleva a cabo su petición de impresión. Cuando el subproceso de la aplicación principal termina, todos los otros subprocesos creados a partir de ese proceso también finalizan. Considere estas dos definiciones del Kit de desarrollo de software de Microsoft Foundation Class (MFCSDK):

- Proceso: Una instancia que se ejecuta de una aplicación
- Subproceso: Una ruta de ejecución dentro de un proceso

C++ y MFC llevan muchos años apoyando el desarrollo de aplicaciones multiproceso. Como el corazón del sistema operativo Windows está escrito con estas herramientas, es importante que sean compatibles con la capacidad de crear subprocesos en los que se puedan asignar y crear tareas. En los primeros tiempos de Windows 3.1, la multitarea no existía; este concepto se hizo realidad con Windows NT 3.5 y NT 4.0 y luego en Windows 95, 98, 98SE, ME, 2000 y XP. Para aprovechar las características del sistema operativo, las aplicaciones con varios subprocesos se hicieron más importantes. Hoy en día, la capacidad de realizar más de una tarea a la vez es un rasgo necesario para una aplicación. Visual Basic 6.0 y versiones anteriores compilaban aplicaciones de un solo proceso, lo que significaba que, sin importar lo que pasara, la aplicación de VB sólo podía hacer una cosa a la vez.

En realidad, en un sistema con un solo procesador, no importa qué herramienta use para escribir su aplicación ya que todo sigue sucediendo en un proceso lineal. Si es un programador de C++, puede crear nuevos subprocesos y realizar tareas mientras tienen lugar otros sucesos, pero en realidad sólo se comparte el mismo tiempo con el resto de procesos que se están ejecutando en el sistema. Si sólo hay un procesador, sólo puede suceder una cosa cada vez. Este concepto se llama *multitarea preferente*.

Multitarea preferente

La multitarea preferente divide el tiempo del procesador entre las tareas o subprocesos en ejecución. Cuando una tarea se está ejecutando, usa un *espacio de tiempo*. Cuando el espacio de tiempo de la tarea que se está ejecutando caduca, en aproximadamente 20 milisegundos, dependiendo del sistema operativo que esté usando, se invalida y otra tarea recibe el espacio de tiempo. El sistema guarda el contexto actual de la tarea invalidada y cuando se asigna a la tarea otro espacio de tiempo, se restaura su contexto y el proceso continúa. Este bucle de tarea continúa repetidamente hasta que la tarea es abortada o finaliza. La multitarea preferente da al usuario la impresión de que se realiza más de una tarea a la vez. ¿Por qué finalizan algunas tareas antes que otras, aunque se inicie antes la última en terminar?

Prioridades de subproceso y bloqueo

Cuando se crean subprocesos, el programador o el sistema operativo les asigna una prioridad. Si una aplicación parece estar bloqueando el sistema, tiene la prioridad más alta y está bloqueando el acceso a los tiempos de los demás subprocesos. Las prioridades determinan lo que sucede y en qué orden. Su aplicación puede estar completa en un 90 por ciento con un proceso determinado cuando, de repente, se inicia un nuevo subproceso y se coloca por delante del subproceso que está ejecutando su aplicación, haciendo que a ese subproceso se le asigne una prioridad menor. Esto suele suceder en Windows. Algunas tareas son más prioritarias que otras. Tomemos como ejemplo el nuevo Reproductor de Windows Media. Al iniciar este proceso hace, básicamente, todo lo que se está ejecutando deja de responder hasta que está completamente cargado, incluyendo la página de la Guía multimedia.

Uno de los mayores peligros a los que se enfrentan los programadores al escribir aplicaciones que usan varios subprocesos son las situaciones de bloqueo, en las que dos o más subprocesos intentan usar el mismo recurso. Un bloqueo de subproceso tiene lugar cuando un subproceso accede a un recurso compartido y otro subproceso con la misma prioridad intenta acceder a dicho recurso. Si los dos subprocesos tienen la misma prioridad y no se ha codificado el bloqueo correctamente, el sistema sucumbe lentamente porque no puede liberar ninguno de los subprocesos de alta prioridad que se están ejecutando. Esto puede suceder fácilmente en las aplicaciones con varios subprocesos. Cuando el programador asigna prioridades a los subprocesos y están compartiendo datos globales, debe bloquear el contexto correctamente para que el sistema operativo gestione correctamente el espacio de tiempo.

Multiprocesamiento simétrico

En un sistema *multiprocesador*, puede tener lugar realmente más de una tarea a la vez. Como cada procesador puede asignar espacios de tiempo a las tareas que quieren ejecutarse, puede realizar más de una tarea a la vez. Cuando necesita ejecutar un subproceso largo que requiera mucho trabajo del procesador, como ordenar 10 millones de registros por nombre, dirección, código postal, apellidos y país, si se usan varios procesadores el trabajo concluirá antes que si se usa un solo procesador. Si puede delegar ese trabajo en otro procesador, la aplicación actualmente en curso no se verá afectada en absoluto. Tener más de un procesador permite este tipo de *multiprocesamiento simétrico* (SMP). La figura 33.1 muestra las opciones de procesador para SQL Server 2000.

Si está ejecutando SQL Server en un equipo multiprocesador, puede definir el número de procesadores que debe usar para las tareas largas y que exigen el tipo mencionado. SQL lleva esto un poco más allá, realizando consultas a lo largo de los diferentes procesadores, uniendo los datos cuando se completa el último

subproceso y mostrando los datos al usuario. Esto se conoce como *sincronización de subprocesos*. El subproceso principal, que crea varios subprocesos, debe esperar a que todos los otros subprocesos estén completos antes de continuar con el procesamiento.

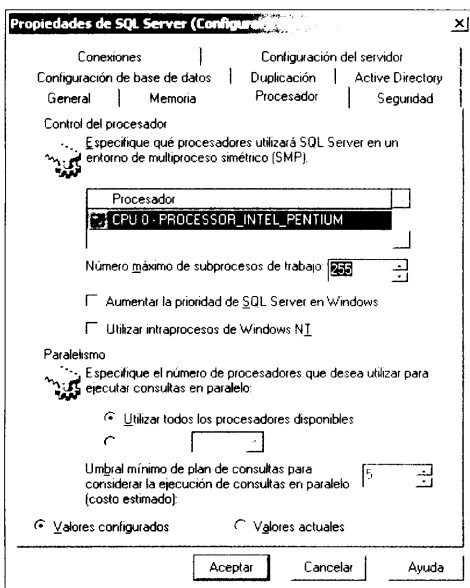


Figura 33.1. Cuadro de diálogo de opciones de SQL Server 2000 Processor

Observe que, cuando se usa un sistema SMP, un solo subproceso sigue ejecutándose en un solo procesador. La aplicación VB6 se ejecuta exactamente igual que si le añade otro procesador adicional. Su aplicación de Access 2.0 de 16 bits tampoco se ejecuta mejor porque 16 bits siguen siendo igual a un solo proceso. Deberá crear nuevos procesos en los demás procesadores para aprovecharlos. Esto significa que no diseñamos una GUI multiprocesador. Desarrolle una GUI que cree otros procesos y pueda reaccionar cuando esos procesos se completen o sean interrumpidos, mientras sigue permitiendo al usuario usar la GUI para otras tareas.

Cómo usar los recursos: cuantos más, mejor

Los subprocesos consumen recursos. Cuando se están usando demasiados recursos, el equipo se ralentiza increíblemente. Si intenta abrir 80 instancias de Visual Studio .NET mientras instala Exchange 2000 en un equipo con 96MB de RAM, percibirá que la pantalla no dibuja correctamente, el ratón no se mueve muy deprisa y la música que estaba escuchando en el reproductor de Windows Media ya no suena. Estos problemas de rendimiento se producen porque hay demasiados subprocesos ejecutándose al mismo tiempo en un sistema operativo con un hardware que no puede gestionar esta cantidad de trabajo. Si intenta hacer

lo mismo en un servidor nuevo, el Unisys de 32 procesadores con 1 terabyte de RAM, no percibirá ninguna pérdida de rendimiento. Cuanta más memoria tenga, más espacio físico tendrán las aplicaciones para crear más subprocesos. Cuando escriba aplicaciones que creen subprocesos, asegúrese de que tiene esto en cuenta. Cuantos más subprocesos cree, más recursos consumirá su aplicación. Esto podría llegar a causar un rendimiento menor que el de una aplicación de un solo proceso: todo depende del SO. "Cuantos más mejor" no se refiere a los subprocesos. Por tanto, tenga cuidado al crear subprocesos en la nueva versión de Tetris con múltiples subprocesos que está escribiendo en C#.

Dominios de aplicación

Anteriormente se indicó que el MFC SDK define un proceso como una instancia de una aplicación que se ejecuta. Cada aplicación que se está ejecutando crea un nuevo subproceso principal, que dura lo que la instancia de la aplicación. Como cada aplicación es un proceso, cada instancia de una aplicación debe tener aislamiento de procesos. Dos instancias de Microsoft Word actúan independientemente entre sí. Cuando hace clic en Ortografía y gramática, la InstanciaA de Word no comprueba los errores ortográficos del documento que se está ejecutando en la InstanciaB de Word. Incluso si la InstanciaA de Word intenta pasar un puntero de memoria a la InstanciaB de Word, la InstanciaB no sabrá qué hacer con él, o siquiera dónde buscarlo, ya que los punteros de memoria sólo se refieren a los procesos en los que se ejecutan.

En .NET Framework, los dominios de aplicación se usan para proporcionar seguridad y aislamiento de aplicación para el código gestionado. Algunos dominios de aplicación pueden ejecutarse en un solo proceso o subproceso, con la misma protección que si las aplicaciones se estuviesen ejecutando en varios procesos. El consumo de recursos se reduce con este concepto, ya que las llamadas no necesitan estar circunscritas a los límites de los procesos si las aplicaciones necesitan compartir datos. Por el contrario, un solo dominio de aplicación puede ejecutarse en varios subprocesos.

Esto es posible gracias al modo en el que el CLR ejecuta el código. Una vez que el código está preparado para ser ejecutado, el compilador JIT ya le ha hecho pasar el proceso de verificación. Este proceso de verificación garantiza que el código no va a realizar acciones inválidas, como accesos a memoria imprevistos, que provoquen un fallo de página.

Este concepto de código *de tipo seguro* garantiza que un código no va a violar ninguna regla después de que el verificador lo haya aprobado al pasar de código MSIL a PE. En las típicas aplicaciones Win32, no existían mecanismos de protección que evitaran que un fragmento de código suplantase a otro, de modo que cada aplicación necesitaba aislamiento de proceso. En .NET, como la seguridad de tipo está garantizada, resulta seguro ejecutar varias aplicaciones de varios proveedores en el mismo dominio de aplicación.

Ventajas de las aplicaciones de varios subprocesos

Algunas aplicaciones pueden sacar partido del multiprocesamiento.

- Aplicaciones con procesos largos.
- Aplicaciones de sondeo y escucha.
- Aplicaciones con botón **Cancelar** en la GUI.

Las siguientes secciones muestran cada una de estas razones.

Aplicaciones con procesos largos

Las aplicaciones que requieren procesos largos en los que el usuario no tiene por qué interactuar se pueden beneficiar del multiprocesamiento porque los procesos de ejecución duradera pueden ser creados en un subproceso trabajador que procese la información en segundo plano hasta que se notifica al proceso que llamó al subproceso que éste ya ha terminado. Entre tanto, el usuario no está obligado a permanecer inactivo, mirando el cursor de reloj de arena para pasar a la siguiente tarea.

Aplicaciones de sondeo y escucha

Las aplicaciones de sondeo y de escucha pueden beneficiarse del multiprocesamiento. Imagine que tiene una aplicación que ha creado subprocesos que están escuchando o sondeando. Cuando sucede algo, un subproceso puede consumir ese evento concreto y los otros subprocesos pueden seguir sondeando o escuchando por si sucede un evento. Un ejemplo de esto es un servicio que escucha las peticiones de un puerto de red o una aplicación de sondeo que comprueba el estado de Microsoft Message Queue (MSMQ) por si hay mensajes. Un ejemplo de una aplicación de sondeo comercial es Microsoft Biztalk Server. Biztalk está constantemente sondeando en busca de cosas como archivos en un directorio o archivos en un servidor SMTP. No puede lograr todo esto con un solo subproceso, de modo que varios subprocesos sondean recursos diferentes. Microsoft Message Queue tiene una extensión para Windows 2000 y una función de Windows XP llamada Message Queue Triggers. Con MSMQ Triggers, puede establecer propiedades que hacen que un desencadenador desencadene un evento. Éste es un servicio de multiprocesamiento que puede manejar miles de solicitudes simultáneas.

Botón Cancelar

Cualquier aplicación que tenga un botón **Cancelar** en un formulario debe seguir este proceso:

1. Abrir y mostrar el formulario modalmente.
2. Iniciar el proceso que tiene lugar en el nuevo proceso.
3. Esperar a que el subproceso termine.
4. Cerrar el formulario.

Al seguir estos pasos, el evento `click` del botón **Cancelar** tiene lugar si el usuario hace clic en el botón mientras otro subproceso se está ejecutando. Si el usuario hace clic en el botón **Cancel**, en realidad hace clic mientras el proceso está ejecutándose en un subproceso que no es el que controla el evento `click`, y el código debe entonces detener el proceso en el otro subproceso que se está ejecutando. Este es un rasgo de la GUI que convierte una buena aplicación en una aplicación genial.

Cómo crear aplicaciones multiproceso

Es hora de empezar a crear aplicaciones multiproceso. El multiproceso se controla a través del espacio de nombres `System.Threading`. Los miembros que usará más a menudo de la clase `Thread` aparecen en la tabla 33.1.

Tabla 33.1. Miembros comunes de la clase `Thread`

Miembro	Descripción
<code>CurrentContext</code>	Obtiene el contexto actual donde se está ejecutando el subproceso
<code>CurrentThread</code>	Obtiene el subproceso actualmente en ejecución
<code>ResetAbort</code>	Restablece una petición de anulación
<code>Sleep</code>	Bloquea el subproceso actual durante el tiempo especificado
<code>ApartmentState</code>	Obtiene o establece el estado de apartamento de un subproceso
<code>IsAlive</code>	Obtiene un valor que indica si el subproceso ha comenzado y no ha finalizado
<code>IsBackground</code>	Obtiene o establece un valor que indica si un subproceso es o no un subproceso en segundo plano
<code>Name</code>	Obtiene o establece el nombre del subproceso
<code>Priority</code>	Obtiene o establece la prioridad del subproceso
<code>Threadstate</code>	Obtiene el estado del subproceso

Miembro	Descripción
Abort	Inicia <code>ThreadAbortException</code> , que puede finalizar el subproceso
Interrupt	Interrumpe un subproceso que se encuentra en estado de subproceso <code>WaitSleepJoin</code>
Join	Espera un subproceso
Resume	Reanuda un subproceso que ha sido suspendido
Start	Inicia la ejecución del subproceso
Suspend	Suspende el subproceso

Cómo crear nuevos subprocesos

La creación de una variable de tipo `System.Threading.Thread` permite crear un nuevo subproceso con el que empezar a trabajar. Como el concepto de subproceso implica la ejecución independiente de otra tarea, el constructor `Thread` necesita la dirección de un procedimiento que hará el trabajo del subproceso que está creando. El delegado `ThreadStart` es el único parámetro que necesita el constructor para empezar a usar el subproceso.

Para probar este código, cree un nuevo proyecto con la plantilla Aplicación de consola. El código del listado 33.1 crea dos nuevos subprocesos y llama al método `Start` de la clase `Thread` para hacer que se ejecute el subproceso.

Listado 33.1. Cómo crear nuevos subprocesos

```
using System;
using System.Threading;

public class Threads
{
    public void Threader1()
    {
    }

    public void Threader2()
    {
    }
}

public class ThreadTest
{
```

```

public static int Main(String[] args)
{
    Threads testThreading = new Threads();

    Thread t1 = new
        Thread(new ThreadStart(testThreading.Threader1));
    t1.Start();

    Thread t2 = new
        Thread(new ThreadStart(testThreading.Threader2));
    t2.Start();

    Console.ReadLine();
    return 0;
}
}

```

Cuando crea una variable de tipo `thread`, el procedimiento que controla el subproceso debe existir para el delegado `ThreadStart`. En caso contrario, se produce un error y la aplicación no compila.

La propiedad `Name` establece o recupera el nombre de un subproceso. Esto le permite usar un nombre con sentido en lugar de una dirección de código hash para hacer referencia a los subprocesos que se están ejecutando. Esto es útil cuando se usan las utilidades de depuración de Visual Studio .NET. En la barra de tareas de depurado hay una lista desplegable con los nombres de los subprocesos en ejecución. Aunque no se puede "salir" del subproceso y saltar a otro subproceso con el depurador, es útil para saber en qué subproceso ha ocurrido un error.

Una vez declaradas, nombradas e iniciadas las variables de los subprocesos, necesita hacer algo en los subprocesos creados. Los nombres de procedimientos que se han pasado al constructor de subprocesos eran `Threader1` y `Threader2`. Ahora puede agregar código a estos métodos y observar cómo actúan. El código debería tener un aspecto como el del listado 33.2.

Listado 33.2. Cómo recuperar información de los subprocesos en ejecución

```

using System;
using System.Threading;

public class Threads
{
    public void Threader1()
    {
        Console.WriteLine (" *** Threader1 Information ***");
        Console.WriteLine
            ("Name: " + Thread.CurrentThread.Name);
        Console.WriteLine
            (Thread.CurrentThread);
        Console.WriteLine
            ("State: " + Thread.CurrentThread.ThreadState);
    }
}

```

```

        Console.WriteLine
            ("Priority: " + Thread.CurrentThread.Priority);
        Console.WriteLine(" *** End Threader1 Information ***");
    }

    public void Threader2()
    {
        Console.WriteLine (" *** Threader2 Information ***");
        Console.WriteLine
            ("Name: " + Thread.CurrentThread.Name);
        Console.WriteLine
            (Thread.CurrentThread);
        Console.WriteLine
            ("State: " + Thread.CurrentThread.ThreadState);
        Console.WriteLine
            ("Priority: " + Thread.CurrentThread.Priority);
        Console.WriteLine(" *** End Threader2 Information ***");
    }
}

public class ThreadTest
{
    public static int Main(String[] args)
    {
        Threads testThreading = new Threads();

        Thread t1 = new
            Thread(new ThreadStart(testThreading.Threader1));
        t1.Name = "Threader1";
        t1.Start();

        Thread t2 = new
            Thread(new ThreadStart(testThreading.Threader2));
        t2.Name = "Threader2";
        t2.Start();

        Console.ReadLine();
        return 0;
    }
}

```

Cuando ejecuta la aplicación, el resultado en su consola debería parecerse al mostrado en la figura 33.2.

El resultado que muestra la figura 33.2 no es muy bonito. Si realiza otra llamada, estará trabajando con subprocesos. Sin establecer una propiedad o dos, el procedimiento `Threader1` nunca terminará antes de que empiece `Threader2`.

Cuando se ejecute el siguiente código:

```
t1.Start();
```

Comenzará la ejecución del código de `Threader1`. Como es un subproceso, tiene apenas 20 milisegundos de espacio de tiempo. en ese periodo de tiempo alcanzará la segunda línea de código de la función, devolverá el control al sistema operativo y ejecutará la siguiente línea de código:

```
t2.start();
```

El procedimiento `Threader2` se ejecuta entonces durante su espacio de tiempo y es sustituido por el subproceso `t1`. Este ir y venir de procesos continúa hasta que los dos procedimientos puedan finalizar.

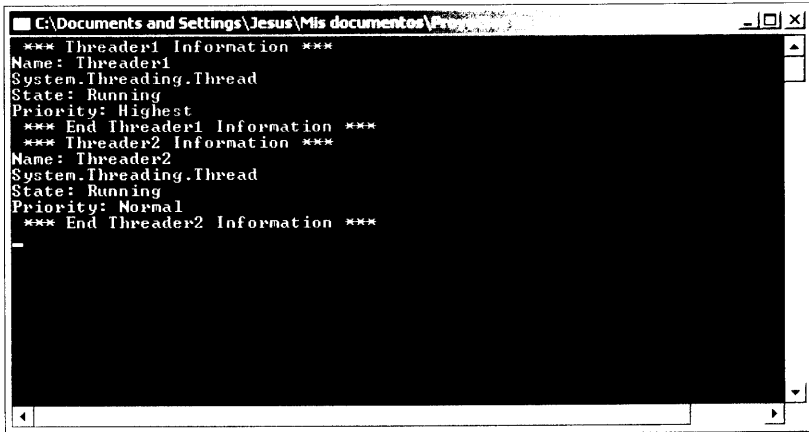


Figura 33.2. Resultado de una aplicación de subprocesamiento

Prioridad de los subprocesos

Para que el procedimiento `Threader1` finalice antes de que el procedimiento `Threader2` empiece, debe darle a la propiedad `Priority` el valor de la enumeración `ThreadPriority` correcta, con el objeto de asegurarse de que el subproceso `t1` tenga prioridad sobre cualquier otro subproceso. Antes de la llamada al método `t1.Start`, agregue el siguiente código:

```
t1.Priority = ThreadPriority.Highest;
```

Cuando establezca la prioridad al máximo, `t1` finalizará antes que `t2`. Si ejecuta la aplicación de nuevo, el resultado deberá ser como el que muestra la figura 33.3. La enumeración `ThreadPriority` establece cómo se planifica un determinado subproceso en relación a los otros subprocesos en ejecución. `ThreadPriority` puede adoptar cualquiera de los siguientes valores: *AboveNormal*, *BelowNormal*, *Highest*, *Lowest* o *Normal*. El algoritmo que determina el orden de los subprocesos varía en función del sistema operativo en el que se ejecutan los subprocesos. Por defecto, cuando se crea un nuevo subproceso, se le concede la prioridad 2, que es *Normal* en la enumeración.

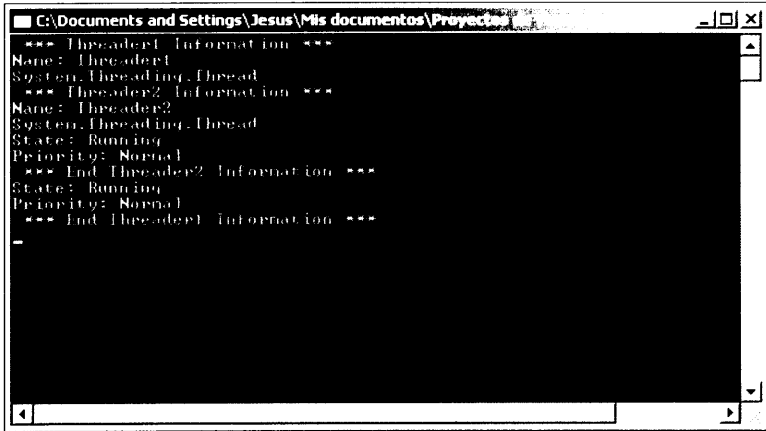


Figura 33.3. Resultado tras establecer la prioridad del subproceso

Estado del subproceso

Al crear un nuevo subproceso, llamamos al método `Start()`. En ese momento, el sistema operativo asigna espacio de tiempo para la dirección del procedimiento pasado al constructor de subprocesos. Aunque el subproceso puede durar mucho tiempo, sigue pasando por diferentes estados mientras los otros subprocesos están siendo procesados por el sistema operativo. Este estado puede resultarle útil en sus aplicaciones. Basándose en el estado del subproceso, puede determinar si alguna otra cosa necesita ser procesada. Aparte de `Start`, los estados de subproceso que usará más frecuentemente son `Sleep` y `Abort`. Al pasar un número de milisegundos al constructor `Sleep`, está indicando al subproceso que abandone el resto de su espacio de tiempo. Si llama al método `Abort` se detiene la ejecución del subproceso. El listado 33.3 muestra código que usa `Sleep` y `Abort`.

Listado 33.3. Cómo usar el método `Thread.Sleep`

```

using System;
using System.Threading;

public class Threads
{
    public void Threader1()
    {
        for(int intX = 0; intX < 50;intX ++)
        {
            if(intX == 5){
                Thread.Sleep(500);
                Console.WriteLine("Thread1 Sleeping");}
        }
    }
}

```

```

public void Threader2()
{
    for(int intX = 0; intX < 50;intX ++)
    {
        if(intX == 5){
            Thread.Sleep(500);
            Console.WriteLine("Thread2 Sleeping");}
        }
    }
}

public class ThreadTest
{
    public static int Main(String[] args)
    {
        Threads testThreading = new Threads();

        Thread t1 = new
            Thread(new ThreadStart(testThreading.Threader1));
        t1.Priority = ThreadPriority.Highest;
        t1.Start();

        Thread t2 = new
            Thread(new ThreadStart(testThreading.Threader2));
        t2.Start();

        Console.ReadLine();
        return 0;
    }
}

```

Observe que se ha establecido la propiedad `Priority` del subproceso `t1` al máximo. Esto significa que, pase lo que pase, se ejecutará antes de que comience `t2`. Sin embargo, en el procedimiento `Threader1`, tiene el siguiente bloque `if`:

```

for(int intX = 0; intX < 50;intX ++)
{
    if(intX == 5){
        Thread.Sleep(500);
        Console.WriteLine("Thread2 Sleeping");}
}

```

Esto indica al subproceso `t1` que se bloquee durante 500 milisegundos, abandonando su actual espacio de tiempo y permitiendo que el subproceso `t2` comience a ejecutarse. Cuando los dos subprocesos se han completado, se invoca al método `Abort` y los subprocesos son eliminados.

Las llamadas al método `Thread.Suspend` bloquean un subproceso indefinidamente, hasta que otro subproceso lo active de nuevo. Si alguna vez ha observado el contador del procesador en el Administrador de tareas llegar al 100 por

ciento cuando no está gastando memoria, puede entender lo que sucede cuando un subproceso se bloquea. Para que el subproceso vuelva a funcionar, debe llamar al método `Resume` desde otro subproceso para que pueda reanudarse. El siguiente código muestra los métodos `Suspend` y `Resume`:

```
Thread.CurrentThread.Suspend;
Console.WriteLine("Thread1 Suspended");
Thread.CurrentThread.Resume;
Console.WriteLine("Thread1 Resumed");
```

Aquí debe tener mucha precaución: Suspend subprocesos puede acarrear resultados indeseados. Debe asegurarse de que otro subproceso reanude el subproceso.

La figura 33.4 muestra lo descrito en el párrafo anterior. Observe en la figura que la ventana de la consola está en la línea de código `T1.Suspended`. Este ejemplo refleja una prueba, de modo que no necesita el método `Resume`. Los resultados del Administrador de tareas indican el estado del sistema.

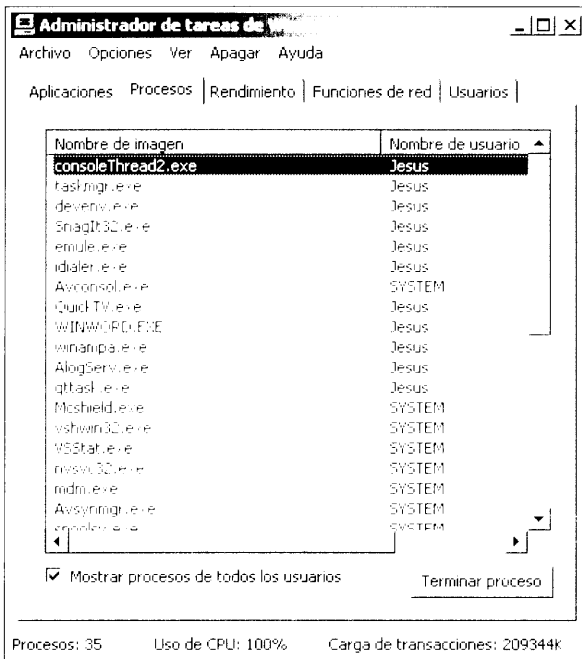


Figura 33.4. Procesador forzado por un subproceso bloqueado

`ThreadState` es una combinación bit a bit de la enumeración `FlagsAttribute`. En cualquier momento, un subproceso puede estar en más de un estado. Por ejemplo, si un subproceso es un subproceso en segundo plano y se está ejecutando en ese momento, el estado puede ser `Running` y `Background`. La tabla 33.2 describe los posibles estados en los que se puede encontrar un subproceso.

Tabla 33.2. Miembros de ThreadState

Miembro	Descripción
Aborted	El subproceso se ha anulado.
AbortRequested	Se ha solicitado la anulación del subproceso.
Background	El subproceso está ejecutándose como subproceso en segundo plano.
Running	El subproceso está ejecutándose.
Suspended	El subproceso se ha suspendido.
SuspendRequested	Se ha solicitado que el subproceso se suspenda.
Unstarted	El subproceso no se ha iniciado.
WaitSleepJoin	El subproceso se ha bloqueado como resultado de una llamada a <code>Wait</code> , <code>Sleep</code> o <code>Join</code> .

Cómo unir subprocesos

El método `Thread.Join` espera a que el subproceso termine antes de seguir el proceso. Esto es útil si crea varios subprocesos para que cumplan una determinada tarea, pero antes de que quiera que la aplicación en segundo plano continúe, debe asegurarse de que todos los subprocesos que creó han terminado. En el siguiente ejemplo, cambie:

```
T2.Join();

por

Console.WriteLine("Writing");
```

La segunda vez que ejecute el código, obtendrá dos conjuntos de resultados. El resultado `Writing` no aparece en la consola hasta que los dos subprocesos han terminado.

Listado 33.4. Cómo unir subprocesos

```
using System;
using System.Threading;

public class Threads
{
    public void Threader1()
    {
        for(int intX = 0; intX < 50;intX ++)
        {
            if(intX == 5)
            {
```

```

        Thread.Sleep(500);
        Console.WriteLine("Thread1 Sleeping");
    }
}

public void Threader2()
{
    for(int intX = 0; intX < 50;intX ++){
        if(intX == 5){
            Thread.Sleep(500);
            Console.WriteLine("Thread2 Sleeping");
        }
    }
}

public class ThreadTest
{
    public static int Main(String[] args)
    {
        Threads testThreading = new Threads();

        Thread t2 = new
            Thread(new ThreadStart(testThreading.Threader2));
        t2.Start();

        Thread t1 = new
            Thread(new ThreadStart(testThreading.Threader1));
        t1.Priority = ThreadPriority.Highest;
        t1.Start();

        /* Invoca a Join para que espere a que todos los
        subprocesos terminen */

        t2.Join();

        Console.WriteLine("Writing");

        Console.PeadLine();
        return 0;
    }
}

```

Como puede ver, el establecimiento de varias propiedades en los subprocesos simplifica mucho su control. Tenga en cuenta que tras suspender un subproceso, debe reactivarlo o su sistema consumirá recursos innecesariamente.

Cómo sincronizar subprocesos

La sincronización de datos es un aspecto importantísimo en la utilización de subprocesos. Aunque no es una tarea compleja de programación, los datos corren el riesgo de estropearse si no los dirige correctamente.

Cuando los subprocesos están ejecutándose, comparten tiempo con otros subprocesos en ejecución. Esto se aprecia perfectamente en el ejemplo que ejecutamos en este capítulo. Si tiene un método que se está ejecutando en varios subprocesos, cada subproceso sólo tiene varios milisegundos del tiempo del procesador antes de que el sistema operativo sustituya el subproceso para darle tiempo a otro subproceso del mismo método. Si está en medio de una instrucción matemática o de la concatenación de un nombre, es probable que el subproceso se detenga algunos milisegundos y otro subproceso en ejecución sobrescriba sus datos. Sin embargo, esto no es el fin del mundo porque varios métodos le permiten hacer que esto no suceda. Observe el siguiente código:

```
{
    int Y;
    int V;
    for(int Z = 0; Z < 20; Z++){
        return Y * V;}
}
```

Es muy probable que durante el bucle, un subproceso en ejecución se detenga para permitir que otro subproceso use este método. Recuerde que esto sólo ocurre si permite que haya múltiples subprocesos accediendo a este bloque de código. Al escribir aplicaciones con múltiples subprocesos, esto sucede muy a menudo, de modo que necesita saber cómo solucionar la situación. El siguiente código resuelve este problema:

```
lock(this){
    int Y;
    int V;
    for(int Z = 0; Z < 20; Z++){
        return Y * V;}
}
```

La instrucción `lock` es un modo de obligar a los subprocesos a unirse. Su implementación es un poco diferente a la del método `Join`. Con `lock`, evaluamos una expresión que se ha pasado al bloque `lock`. Cuando un subproceso llega al bloque `lock`, espera hasta que puede conseguir un bloqueo exclusivo de la expresión que se está evaluando antes de intentar seguir con el proceso. Esto asegura que no se estropeen los datos compartidos al usar varios subprocesos.

La clase `Monitor` permite la sincronización mediante los métodos `Monitor.Enter`, `Monitor.TryEnter` y `Monitor.Exit`. Tras usar un bloqueo en una región de código, puede usar los métodos `Monitor.Wait`, `Monitor.Pulse` y `Monitor.PulseAll` para determinar si un subproceso

debe continuar con un bloqueo o si algunos métodos bloqueados anteriormente están ya disponibles. `Wait` levanta el bloqueo si éste se mantiene y espera a que se le notifique. Cuando se invoca `Wait`, se rompe el bloqueo y vuelve para volver a cerrarlo.

Sondeo y escucha

El sondeo y la escucha son otras dos instancias que representan la utilidad del multiprocesamiento. Las bibliotecas de clases, como `System.Net.Sockets`, incluyen un rango completo de clases de multiproceso que pueden ayudarle a crear agentes de escucha TCP, agentes de escucha UDP y una gran cantidad de tareas de red que requieren multiproceso. Observe la clase `TimerCallback` del espacio de nombres `System.Threading`. Esta clase es muy parecida a las otras que hemos estado usando hasta ahora, excepto que una de las partes del constructor es un temporizador, lo que le permite realizar sondeos en busca de acciones cada ciertos intervalos.

Puede conseguir el mismo resultado agregando un control temporizador al formulario, pero usando la clase `TimerCallback`, el temporizador y la respuesta al procedimiento son automáticos.

El listado 33.5 usa una devolución de llamada temporizada para sondear un directorio en busca de archivos. Si se encuentra un archivo, es rápidamente borrado. Sólo debe ejecutar este código en un directorio de prueba, porque elimina archivos. El siguiente código de ejemplo busca en el directorio `C:\Poll`. El constructor de la clase `TimerCallback` espera una dirección en la que se va a ejecutar el subproceso; un tipo de datos objeto que representa el estado del temporizador; un tiempo de vencimiento, que representa el período de tiempo hasta el que se van a realizar sondeos; y un período, que es una variable en milisegundos que indica cuándo se produce el intervalo de sondeo.

Listado 33.5. Cómo usar el delegado `TimerCallback`

```
using System;

using System.IO;
using System.Threading;

namespace cSharpTimerCallback
{
    class Class1
    {
        public static void Main()
        {
            Console.WriteLine
                ("Checking direcotry updates every 2 seconds.");
            Console.WriteLine
```

```

        (" (Hit Enter to terminate the sample)");
Timer timer = new
    Timer(new TimerCallback(CheckStatus), null, 0, 2000);
Console.ReadLine();
timer.Dispose();
}
    static void CheckStatus(Object state)
    {
string[] str = Directory.GetFiles("C:\\Poll");
if(str.Length>0)
{
    for(int i = 0; i < str.Length;i++)
    {
Console.WriteLine(str[i]);
File.Delete(str[i]);
    }
}
Console.WriteLine("Directory Empty");
    }
}

```

Tras ejecutar este programa durante un tiempo y copiar periódicamente unos cuantos archivos en el directorio C:\Poll, el resultado en la consola tendrá un aspecto parecido al de la figura 33.5.

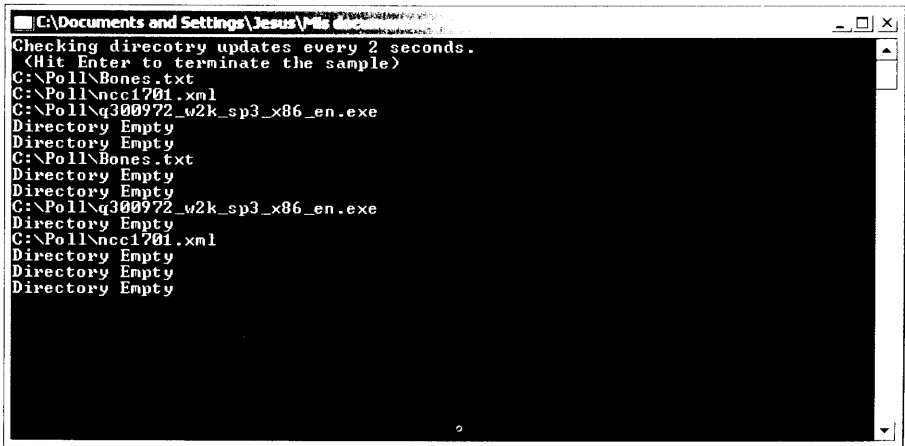


Figura 33.5. Resultado del listado 33.2

Resumen

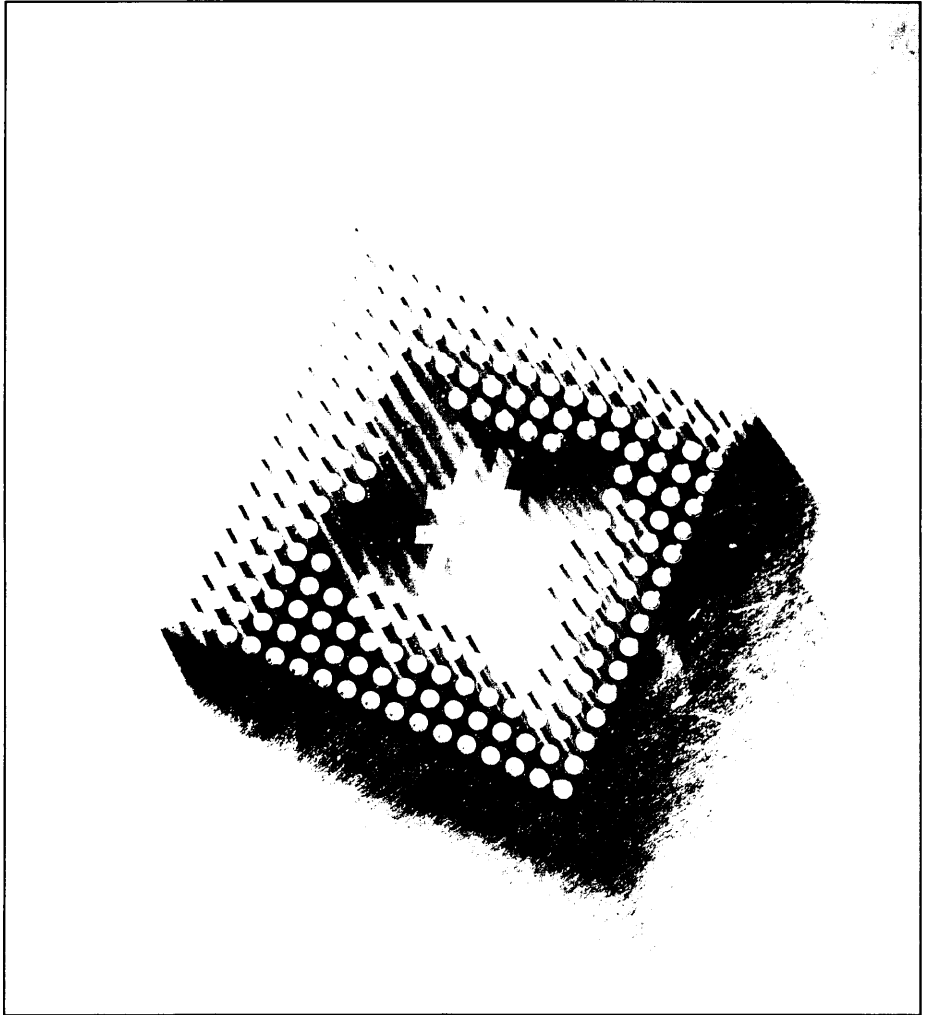
En este capítulo ha aprendido a implementar múltiples subprocesos en C# con el espacio de nombres `System.Thread`.

La idea básica tras el multiprocesamiento es simple: al crear más de un subproceso, puede realizar más de una tarea a la vez. Tiene que probar antes el

número de subprocesos que vaya a crear. Demasiados subprocesos pueden causar problemas de recursos. Si no se crean suficientes subprocesos, puede que la aplicación no trabaje con todo su potencial.

Con el ejemplo que ha creado aquí, debería estar preparado para implementar subprocesos en sus propias aplicaciones. Simplemente evite tomar riesgos porque, como sabe, las aplicaciones con varios subprocesos pueden generar varios problemas.

Como siempre, planifique su aplicación con antelación y decida si el uso del multiprocesamiento es una parte importante de este proceso de planificación.



34 **Cómo trabajar con COM**

Como programador de Windows, probablemente haya creado muchos componentes COM, bien como DLL solitarias o como DLL que se ejecutan en servicios COM+. Con la llegada de .NET, se preguntará si debe reescribir todo con este nuevo lenguaje. Las buenas noticias son que no tiene que reescribir ninguno de sus componentes. Microsoft ha sido suficientemente considerado como para proporcionarnos las herramientas necesarias para usar sus componentes existentes de .NET. Además, estos componentes pueden invocarse de forma segura desde el Entorno común de ejecución. En este capítulo aprenderá lo sencillo que es modificar el código existente y usarlo desde un cliente administrado por .NET. El cliente puede ser cualquier cosa: una aplicación Web, otro componente .NET o incluso una aplicación basada en servicios. No importa: la funcionalidad principal funciona en todo tipo de aplicaciones.

Aunque siempre tiene la opción de reescribir su código, no es necesario. Probablemente querrá usar .NET para todos sus programas, especialmente el desarrollo GUI, ya que es mucho más sencillo de usar que las versiones anteriores. Al mismo tiempo, no querrá reescribir toda la lógica empresarial básica de sus aplicaciones. Con .NET, todo esto es posible: puede convertir sus aplicaciones a .NET mientras sigue usando las miles de líneas de código existente que ya ha escrito en sus componentes.

En este capítulo, aprenderá a usar sus componentes COM existentes desde un cliente .NET usando las herramientas suministradas junto a .NET y verá cómo todo sucede en segundo plano.

Introducción al Contenedor al que se puede llamar en tiempo de ejecución

El código .NET puede acceder a código no administrado mediante un proxy llamado Contenedor al que se puede llamar en tiempo de ejecución o RCW. El RCW permite a una aplicación .NET ver el componente no administrado como si fuera un componente administrado.

Para ello, organiza llamadas a métodos, eventos y propiedades mediante un contenedor creado por su aplicación o manualmente, usando herramientas (como el Importador de la biblioteca de tipos) que proporciona el Framework. Usando información de la biblioteca de tipos COM, el RCW controla la interoperabilidad entre código administrado y el no administrado. Cuando ejecuta su aplicación, no es consciente de si el código que se está ejecutando es el de una DLL no administrada o COM. El usuario de los componentes no necesita tener ningún conocimiento especial de cómo se escribió el código, en qué lenguaje fue escrito o si es un componente .NET.

Todas las características del entorno administrado, como la recogida de elementos no utilizados y el control de excepciones, están disponibles para el cliente .NET como si estuviera usando código administrado. Esto hace que sea extremadamente sencillo transportar módulos desde sus aplicaciones anteriores a .NET hasta .NET, sin tener que realizar un gran trabajo o comprender perfectamente los entresijos de cualquier lenguaje .NET que esté usando: C#, J# o VB .NET, o cualquier otro.

Puede rehacer el código cliente y colocar la lógica de datos y la lógica empresarial en su sitio usando interoperabilidad COM. La figura 34.1 muestra la relación entre COM DLL, el RCW y la aplicación administrada .NET.

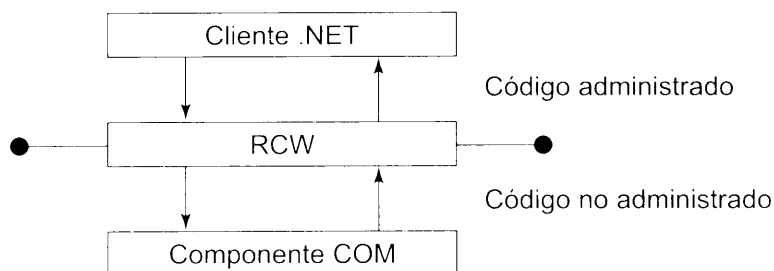


Figura 34.1. Código administrado y no administrado conviviendo juntos sin problema

Cómo crear ensamblados .NET a partir de componentes COM

Para usar sus componentes COM en una aplicación .NET, debe crear el ensamblado de interoperabilidad, o RCW, que organiza las llamadas de método desde el cliente .NET al servidor COM. En .NET hay varios modos de realizar esto. Los dos modos más comunes son los siguientes:

- La utilidad Importador de la biblioteca de tipos, o Tlbimp.exe, proporcionada junto a .NET Framework
- Hacer referencia directamente al COM desde la aplicación de C# VS .NET

El proxy creado para la interoperabilidad se basa en los metadatos expuestos en la biblioteca de tipos del componente COM al que se está intentando acceder. Las bibliotecas de tipos COM pueden hacerse accesibles de una de estas dos formas:

- Las bibliotecas de tipos pueden encontrarse como archivos individuales. Las bibliotecas individuales de tipos suelen tener la extensión TLB. Las bibliotecas individuales de tipos más antiguas pueden tener la extensión OLB. Si está creando una DLL ActiveX de Visual Basic, puede crear una biblioteca individual de tipos para su componente seleccionando la opción Archivos de servidor remoto en el cuadro de diálogo Propiedades del proyecto.
- Las bibliotecas de tipo también pueden encontrarse incrustadas en un servidor COM como un recurso binario. Los servidores COM en curso, empaquetados como DLL y los servidores COM detenidos, empaquetados como EXE, pueden incluir la biblioteca de tipos como un recurso del propio servidor COM. Los componentes COM creados con Visual Basic tienen la biblioteca de tipos compilada dentro de la DLL.

En la siguiente sección aprenderá a crear el ensamblado de interoperabilidad a partir de una DLL COM mediante los dos métodos descritos al principio de esta sección: usando la utilidad Tlbimp y haciendo referencia directamente al DLL desde Visual Studio .NET.

Cómo usar la utilidad Tlbimp

La utilidad Tlbimp es una aplicación individual de consola que crea el ensamblado de interoperabilidad .NET basado en la DLL COM que especifique. Está situada en el directorio Framework SDK en Archivos de programa. El siguiente fragmento de código muestra la sintaxis de Tlbimp:

```
tlbimp [COMDllFilename] [/options]
```

La tabla 34.1 recoge las opciones de línea de comandos para `tlbimp.exe`

Tabla 34.1. Tlbimp.exe Opciones

Opción	Descripción
<code>/asmversion:versionnumber</code>	Especifica el número de versión del ensamblado que se genera
<code>/delaysign</code>	Especifica a Tlbimp.exe que firme el ensamblado resultante con firma postergada
<code>/help</code>	Muestra las opciones de ayuda para tlbimp.exe
<code>/keycontainer:containername</code>	Firma el ensamblado resultante con un nombre seguro utilizando el par de claves pública y privada que se encuentra en el contenedor de claves especificado en el parámetro <i>nombre del contenedor</i>
<code>/nologo</code>	Suprime la presentación de la portada de inicio de Microsoft
<code>/out:filename</code>	Especifica el nombre del archivo resultante que se creará. Por defecto, el archivo resultante tiene el mismo nombre que la DLL COM, pero tenga cuidado si intenta sobrescribir el archivo en caso de que exista en la misma ruta
<code>/primary</code>	Genera un ensamblado de interoperabilidad primario para la biblioteca de tipos
<code>/publickey:filename</code>	Especifica el archivo que contiene la clave pública que se debe usar para señalar el ensamblado resultante
<code>/reference:filename</code>	Especifica el archivo de ensamblado que se utiliza para resolver referencias en tipos definidos fuera de la biblioteca de tipos actual
<code>/silent</code>	Suprime la presentación de mensajes de aprobación
<code>/strictref</code>	No importa una biblioteca de tipos si la herramienta no resuelve todas las referencias incluidas en el ensamblado actual o en los ensamblados especificados con la opción <code>/reference</code>
<code>/sysarray</code>	Importa cualquier SafeArray de estilo COM como un tipo Clase <code>System.Array</code> administrado

Opción	Descripción
/unsafe	Genera interfaces sin comprobaciones de seguridad de .NET Framework. Esta opción no se debe utilizar a no ser que se conozcan los riesgos que supone exponer este código como no seguro
/verbose	Muestra información adicional sobre la biblioteca de tipos importada cuando se ejecuta <code>tlbimp.exe</code>
/?	Muestra ayuda sobre la sintaxis para <code>tlbimp.exe</code>

Este comando produce un ensamblado .NET con una extensión DLL que recibe el mismo nombre base que la biblioteca incrustada en el archivo de la biblioteca de tipos (que puede ser diferente del nombre de archivo de la propia biblioteca de archivos). El comando `tlbimp` admite el nombre de un archivo de biblioteca de tipos como dato introducido:

```
tlbimp server.tlb
```

También puede admitir el nombre de un servidor COM que contiene la biblioteca de tipos incrustada:

```
tlbimp server.dll
```

Al usar la opción `/out`, se puede especificar un nombre alternativo para el ensamblado .NET creado:

```
tlbimp server.dll /out:dotNetServer.dll
```

El ensamblado producido por la herramienta `Tlbimp.exe` es un ensamblado .NET estándar que puede ser visto con `ildasm.exe`. El ensamblado no contiene el código del servidor COM; en su lugar, contiene referencias que ayudan al CLR a encontrar los objetos COM almacenados en el servidor, como los GUID del objeto COM. Imagine el ensamblado generado por `tlbimp` como un puente que conecta el código .NET con el servidor COM. Como el código COM sigue estando en el servidor COM, no debe olvidar instalar e inscribir en el registro todos los servidores COM que piense usar en sus aplicaciones .NET. En realidad, esto supone una gran ventaja para el programador. Como el servidor COM sigue registrado con Windows, las aplicaciones COM estándar que no son conscientes de la existencia de .NET pueden seguir usando el mismo servidor COM sin trasladar código a una plataforma específica .NET.

Cómo crear un componente COM

Antes de emplear la utilidad `Tlbimp` necesita un componente COM con el que trabajar. El listado 34.1 muestra el código para un sencillo DLL ActiveX VB6

con varias funciones de clase común, como establecer y recuperar una propiedad, desencadenar un evento y devolver un valor de un método que tenga parámetros de entrada.

Listado 34.1. Código de servidor COM para Visual Basic 6.0

```
Option Explicit

Private strMessage As String

Public Event COMEvent(Message As String)

Private Sub Class_Initialize()
    strMessage = "Default Message"
End Sub

Public Property Get Message() As String
    message = strMessage
End Property

Public Property Let Message(ByVal vNewValue As String)
    strMessage = vNewValue
End Property

Public Function SquareIt(int1 As Integer, int2 As Integer) As Integer
    squareIt = int1 * int2
End Function

Public Sub FireCOMEvent()
    RaiseEvent COMEvent(strMessage)
End Sub
```

Este código se coloca en un módulo de clase llamado `COMObject`. El módulo de clase está en un proyecto llamado `VB6COMServer`. Visual Basic 6.0 compila este código en un servidor COM en proceso e incrusta una biblioteca de tipo en el servidor. La representación legible de la biblioteca de tipos, escrita en Lenguaje de definición de interfaz (IDL) de COM se muestra en el listado 34.2.

Listado 34.2. Fuente IDL para el servidor COM del listado 34.1

```
// Archivo .IDL generado (por el Visor de objetos OLE/COM)
//
// typelib filename: VB6COMServer.dll

[
    uuid(B409eC50-ACA4-4E1F-8D3E-F36F1EE5F03B),
    version(1.0)
]
library VB6COMServer
{
    // TLib : // TLib : OLE Automation : {00020430-0000-0000-
```

```

C000-0000000000046}
    importlib("stdole2.tlb");

// Declaración anticipada de todos los tipos definidos en
// esta biblioteca de clases
interface _COMObject;
dispinterface __COMObject;

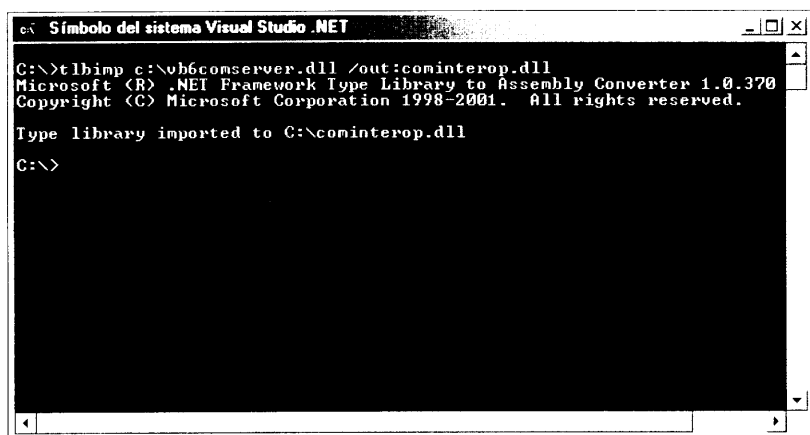
[
    odl,
    uuid(5960D780-FEA2-4383-B2CB-9F78E4677142),
    version(1.0),
    hidden,
    dual,
    nonextensible,
    oleautomation
]
interface _COMObject : IDispatch {
    [id(0x68030000), propget]
    HRESULT Message([out, retval] BSTR* );
    [id(0x68030000), propput]
    HRESULT Message([in] BSTR );
    [id(0x60030002)]
    HRESULT SquareIt(
        [in, out] short* int1,
        [in, out] short* int2,
        [out, retval] short* );
    [id(0x60030003)]
    HRESULT FireCOMEvent();
};

[
    uuid(50730C97-09EB-495C-9873-BEC6399AA62A),
    version(1.0)
]
coclass COMObject {
    [default] interface _COMObject;
    [default, source] dispinterface __COMObject;
};

[
    uuid(A4D4C3D8-DFFF-45DB-9A14-791E4F82EF35),
    version(1.0),
    hidden,
    nonextensible
]
dispinterface __COMObject {
    properties:
    methods:
        [id(0x00000001)]
        void COMEvent([in, out] BSTR* Message);
};
};

```

Para crear el ensamblado de interoperabilidad que permite a la aplicación C# usar la DLL no administrada, debe ejecutar la utilidad Tlbimp descrita en la anterior sección. En la figura 34.2, puede ver que el parámetro `/out:` se usa para dar al ensamblado de interoperabilidad el nombre `cominterop.dll`. El nombre del ensamblado resultante puede ser el que prefiera (incluso puede tener el mismo nombre que el componente COM original).



```
c:\ Símbolo del sistema Visual Studio .NET
C:\>tlbimp c:\vb6conserver.dll /out:cominterop.dll
Microsoft (R) .NET Framework Type Library to Assembly Converter 1.0.370
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Type library imported to C:\cominterop.dll
C:\>
```

Figura 34.2. La utilidad Tlbimp en funcionamiento

El VB6COMServer.dll que se creó usando VB6 ahora puede ser consumido desde cualquier cliente .NET (siempre que la aplicación haga referencia al ensamblado `cominterop.dll` y el componente VB6 esté registrado en el equipo que esté intentando consumir el código). Como el resultado de Tlbimp es ahora un ensamblado .NET, puede usar la utilidad ILDASM para ver los detalles sobre el metadato que se creó a partir de la DLL ActiveX que usa realmente el CLR. La figura 34.3 muestra la utilidad ILDSM cuando se ejecuta el nuevo `cominterop.dll` recién creado.

El ensamblado generado al importar la biblioteca de tipos cuyo código fuente se muestra en el listado 34.3 incluye un espacio de nombres llamado `cominterop`, que es el nombre del ensamblado que se pasó al parámetro `/out` desde la utilidad Tlbimp. Este espacio de nombres debe ser tratado exactamente igual que un espacio de nombres definido por el código o por .NET Framework: el código debe hacer referencia al espacio de nombres cuando use alguna de las clases del espacio de nombres.

La figura 34.3 muestra las clases insertadas en el ensamblado generado por tlbimp. La clase que usamos en el código C# para trabajar con el objeto COM tiene el mismo nombre que el objeto COM en la instrucción `coclass` de la fuente IDL. En el listado 34.3, el objeto COM recibe un nombre `coclass` de `COMObject`. El ensamblado generado por tlbimp incluye una clase .NET con el mismo nombre y es esta clase la que se usa en el código para trabajar con el objeto Visual Basic COM.

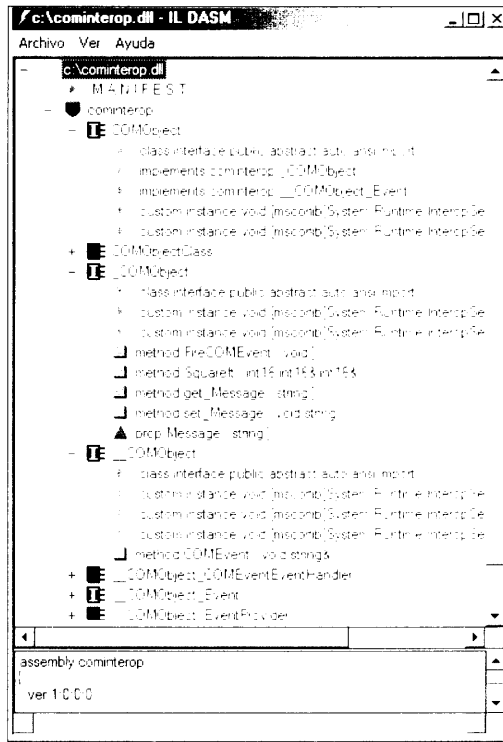


Figura 34.3. ILDASM con un ensamblado generado por Tibimp

Cómo usar el ensamblado de interoperabilidad desde C#

Usar el componente COM desde C# es muy sencillo una vez que ha creado el ensamblado de interoperabilidad. Para usar este ensamblado, siga los siguientes pasos:

1. Cree una aplicación de cliente de prueba. Para hacerlo más sencillo, cree una nueva aplicación de formulario de Windows y llámela Interop.
2. Una vez que ha creado la aplicación, coloque su código en el evento `click` de un botón y agregue un botón a `Form1.cs`. A continuación, haga clic con el botón derecho del ratón sobre el archivo `References` en el Explorador de soluciones y seleccione `Agregar referencia`. Se abrirá el cuadro de diálogo `Agregar referencia`. Es igual al cuadro de diálogo `Agregar referencia` de VB6. Básicamente, debe hacer referencia al ensamblado que necesita usar, que al igual que cualquier otro ensamblado .NET no se añade por defecto a un nuevo proyecto.
3. Para agregar una referencia a la DLL `Cominterop` que creó anteriormente, haga clic en el botón `Examinar` y localice el ensamblado en su disco duro. Una vez hecho, su cuadro de diálogo `Agregar referencia` deberá tener un aspecto similar al de la figura 34.4.

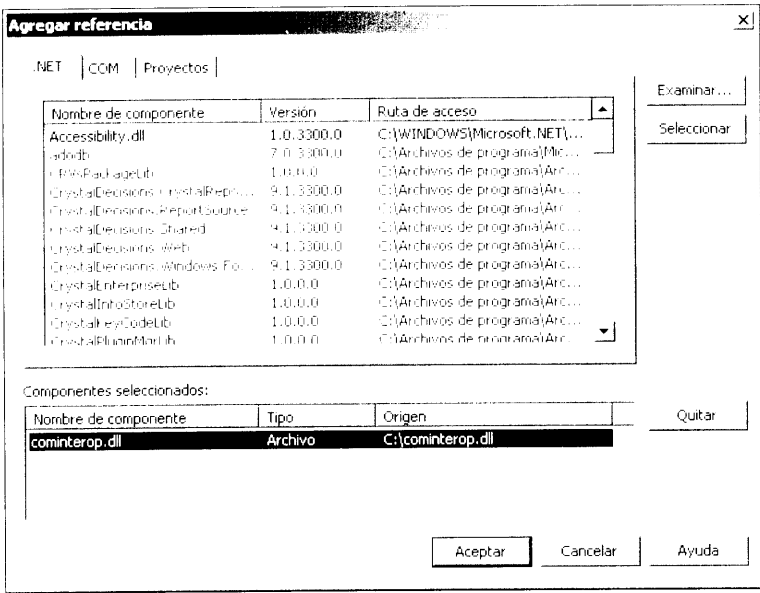


Figura 34.4. Adición de la referencia Cominterop

Una vez que la aplicación hace referencia al ensamblado, puede usarlo exactamente igual que cualquier otro ensamblado .NET. Como Visual Studio .NET dispone de funciones tan útiles (como completar o listar miembros automáticamente) una vez que se agrega la referencia, sus métodos, eventos y propiedades están a su disposición mediante el IDE. La figura 34.5 muestra el listado automático de miembros en funcionamiento después de que la instancia del objeto Cominterop haya sido creada y de que se haga referencia al ensamblado con la instrucción using.

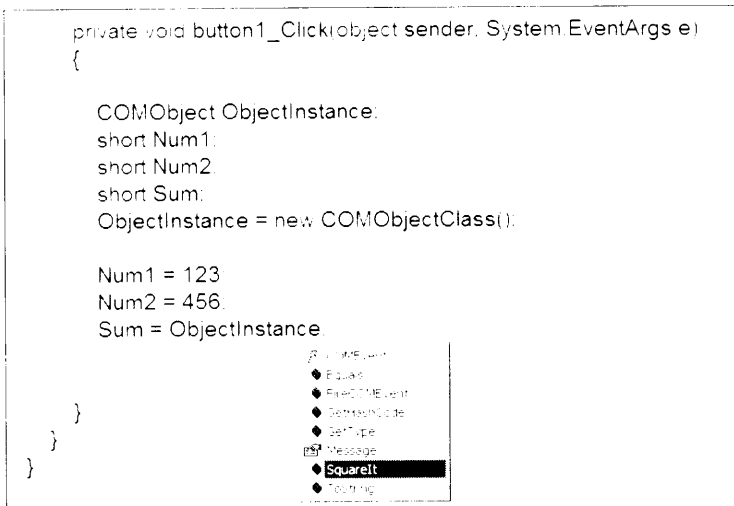


Figura 34.5. Listado automático de miembros en funcionamiento

Para comprobar todos los métodos, propiedades y eventos que ha escrito en la DLL ActiveX, copie el listado 34.3 en la aplicación WindowsForms.

Listado 34.3. Código de cliente COM escrito en C#

```
/// <summary>
/// El principal punto de entrada para la aplicacion.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

// Cree un controlador para el evento
private __COMObject_COMEventEventHandler
COMEventHandlerInstance;

private void button1_Click(object sender, System.EventArgs e)
{
    // cree una nueva instancia de la clase COMObject
    COMObject ObjectInstance;
    short Num1;
    short Num2;
    short Sum;
    ObjectInstance = new COMObjectClass();

    Num1 = 5;
    Num2 = 6;
    // Llame al metodo SquareIt
    Sum = ObjectInstance.SquareIt(ref Num1, ref Num2);

    listBox1.Items.Add (Sum.ToString());
    listBox1.Items.Add (ObjectInstance.Message);

    // Establezca un valor de mensaje diferente del que se
    // ofrece por defecto
    ObjectInstance.Message = "C# Rocks";

    COMEventHandlerInstance = new
__COMObject_COMEventEventHandler(COMEventHandler);
    ObjectInstance.COMEvent += COMEventHandlerInstance;
    ObjectInstance.FireCOMEvent();
}

void COMEventHandler(ref string Message)
{
    listBox1.Items.Add(Message);
}
}
}
```

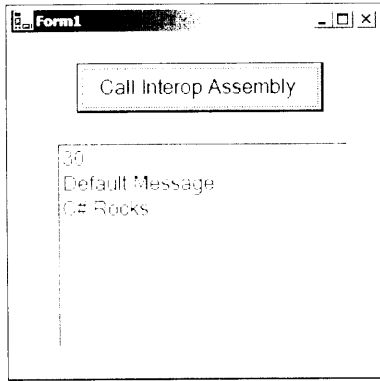


Figura 34.6. Resultado del cliente C# usando el componente COM

Como cualquier otro objeto de .NET, el operador `new` se usa para crear una nueva instancia de la clase `COMObject`, como muestra el siguiente fragmento de código:

```
ObjectInstance = new COMObject();
```

Tras instanciar el nombre de la variable `ObjectInstance`, use el objeto igual que cualquier otro objeto .NET; no hace falta hacer nada especial. El RCW se encarga de toda la interoperabilidad, conversiones de tipos y organización de objetos para los tipos, de modo que no se dará cuenta de ninguno de los preparativos internos COM que se están produciendo.

Si ha usado la interoperabilidad COM de VB.NET, percibirá algunas diferencias respecto al modo en que se pasan los parámetros a los métodos en C#. Si observa el código de C# del método `SquareIt`, verá que se ha agregado la palabra clave `Ref`:

```
Num1 = 5;
Num2 = 3;
// Llame al método SquareIt
Sum = ObjectInstance.SquareIt(ref Num1, ref Num2);
```

Los servidores COM de Visual Basic pueden pasar valores por valor o por referencia. El código C# necesita usar las palabras clave adecuadas cuando pasa parámetros a las llamadas de método COM. Puede usar `ILDASM` para que le ayude a determinar si un parámetro debe pasarse por valor o por referencia.

Abra el ensamblado generado por `Tlbimp` usando la herramienta `ILDASM` y mire la definición del método al que quiere llamar. En este caso, debe llamar al método `SquareIt()`, que se muestra en el ensamblado con la siguiente firma:

```
SquareIt : int16 (int16&,int16&)
```

Tras los dos puntos situamos el tipo de valor devuelto por el método. La firma del método `SquareIt()` muestra un tipo devuelto `int16`, que, en la jerga del lenguaje intermedio, indica un entero de 16 bits. Tras los tipos de parámetro se

coloca un signo & que indica que el parámetro debe pasarse por referencia. Los parámetros que deben ser pasados por referencia deben ir acompañados de la palabra clave `ref` en el cliente C#. Los parámetros que deben pasarse por valor no aparecen con el símbolo & en el ensamblado. En este caso, el cliente C# no necesita usar la palabra clave `ref` en los parámetros.

Cómo hacer referencia a la DLL COM desde C#

En la sección `summary`, aprendió a usar el ensamblado de interoperabilidad creado por `Tlbimp.exe` en una aplicación de formulario Windows de C#. La principal razón para usar `Tlbimp.exe` para crear el ensamblado de interoperabilidades es que se le puede dar un nombre seguro con la utilidad `SN.exe` para después instalarlo en la caché de ensamblados global usando la utilidad `GACUTIL`. Una vez en la GAC, el ensamblado puede ser compartido entre muchos otros ensamblados o proyectos NET. Si está escribiendo una aplicación que usa interoperabilidad COM y el ensamblado no necesita ser compartido, puede simplemente hacer referencia a la DLL COM mediante Visual Studio .NET, lo que creará el RCW por nosotros.

Para agregar una referencia para una DLL COM directamente a su proyecto, siga estos pasos:

1. Haga clic con el botón derecho del ratón en la carpeta `Referencias` del Explorador de soluciones. Se abrirá el cuadro de diálogo `Agregar referencia`, mostrado en la figura 34.7. La segunda ficha, `COM`, muestra todos los objetos COM registrados en el equipo local.

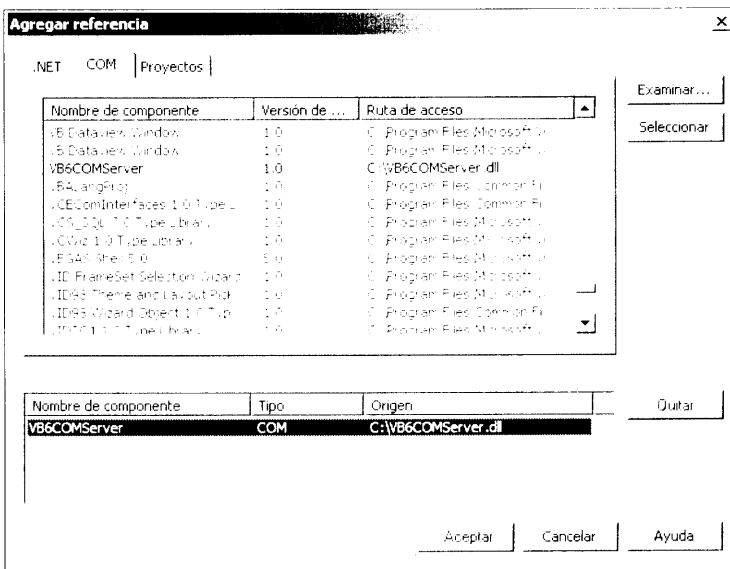


Figura 34.7. Cómo agregar una referencia a un objeto COM directamente

- Tras seleccionar el componente COM que debe usar, puede utilizar el mismo código que usó para escribir la aplicación de formulario de Windows, cambiando solamente el ensamblado al que se hace referencia, que en este caso será VB6ComServer y no Cominterop.

```
using VB6COMServer;
ObjectInstance = new COMObjectClass();
```

Como puede ver, hacer referencia a un componente COM directamente desde IDE es incluso más sencillo usando la utilidad Tlbimp, aunque pierda parte de versatilidad respecto a lo que puede hacer realmente con el componente.

Cómo manejar errores de interoperabilidad

En .NET Framework, el CLR informa de los errores iniciando excepciones cuando algo sale mal. En COM, los HRESULT son el modo de informar de los errores, de modo que el RCW necesita ser capaz de describir el HRESULT de un error dado a la excepción .NET equivalente.

La tabla 34.2 describe los HRESULT estándar de COM a sus excepciones .NET equivalentes.

Tabla 34.2 HRESULT para excepciones .NET

HRESULT	Excepción .NET
MSEE_E_APPDOMAINUNLOADED	AppDomainUnloadedException
COR_E_APPLICATION	ApplicationException
COR_E_ARGUMENT o E_INVALIDARG	ArgumentException
COR_E_ARGUMENTOUTOFRANGE	ArgumentOutOfRangeException
COR_E_ARITHMETIC o ERROR_ARITHMETIC_OVERFLOW	ArithmeticException
COR_E_ARRAYTYPEMISMATCH	ArrayTypeMismatchException
COR_E_BADIMAGEFORMAT o ERROR_BAD_FORMAT	BadImageFormatException
COR_E_COMEMULATE_ERROR	COMEmulateException
COR_E_CONTEXTMARSHAL	ContextMarshalException
COR_E_CORE	CoreException
NTE_FAIL	CryptographicException
COR_E_DIRECTORYNOTFOUND o ERROR_PATH_NOT_FOUND	DirectoryNotFoundException
COR_E_DIVIDEBYZERO	DivideByZeroException
COR_E_DUPLICATEWAITOBJECT	DuplicateWaitObjectException
COR_E_ENDOFSTREAM	EndOfStreamException
COR_E_TYPELOAD	EntryPointNotFoundException
COR_E_EXCEPTION	Exception

HRESULT

Excepción .NET

COR_E_EXECUTIONENGINE	ExecutionEngineException
COR_E_FIELDACCESS	FieldAccessException
COR_E_FILENOTFOUND o ERROR_FILE_NOT_FOUND	FileNotFoundException
COR_E_FORMAT	FormatException
COR_E_INDEXOUTOFRANGE	IndexOutOfRangeException
COR_E_INVALIDCAST o E_NOINTERFACE	InvalidCastException
COR_E_INVALIDCOMOBJECT	InvalidComObjectException
COR_E_INVALIDFILTERCRITERIA	InvalidFilterCriteriaException
COR_E_INVALIDOLEVARIANTTYPE	InvalidOleVariantTypeException
COR_E_INVALIDOPERATION	InvalidOperationException
COR_E_IO	IOException
COR_E_MEMBERACCESS	AccessException
COR_E_METHODACCESS	MethodAccessException
COR_E_MISSINGFIELD	MissingFieldException
COR_E_MISSINGMANIFESTRESOURCE	MissingManifestResourceException
COR_E_MISSINGMEMBER	MissingMemberException
COR_E_MISSINGMETHOD	MissingMethodException
COR_E_MULTICASTNOTSUPPORTED	MulticastNotSupportedException
COR_E_NOTFINITENUMBER	NotFiniteNumberException
E_NOTIMPL	NotImplementedException
COR_E_NOTSUPPORTED	NotSupportedException
COR_E_NULLREFERENCE o E_POINTER	NullReferenceException
COR_E_OUTOFMEMORY o E_OUTOFMEMORY	OutOfMemoryException
COR_E_OVERFLOW	OverflowException
COR_E_PATHTOOLONG o ERROR_FILENAME_EXCED_RANGE	PathTooLongException
COR_E_RANK	RankException
COR_E_REFLECTIONTYPELOAD	ReflectionTypeLoadException
COR_E_REMOTING	RemotingException
COR_E_SAFEARRAYTYPEMISMATCH	SafeArrayTypeMismatchException
COR_E_SECURITY	SecurityException
COR_E_SERIALIZATION	SerializationException
COR_E_STACKOVERFLOW o ERROR_STACK_OVERFLOW	StackOverflowException
COR_E_SYNCHRONIZATIONLOCK	SynchronizationLockException
COR_E_SYSTEM	SystemException
COR_E_TARGET	TargetException
COR_E_TARGETINVOCATION	TargetInvocationException
COR_E_TARGETPARAMCOUNT	TargetParameterCountException
COR_E_THREADABORTED	ThreadAbortException

HRESULT	Excepción .NET
COR_E_THREADINTERRUPTED	ThreadInterruptedException
COR_E_THREADSTATE	ThreadStateException
COR_E_THREADSTOP	ThreadStopException
COR_E_TYPELOAD	TypeLoadException
COR_E_TYPEINITIALIZATION	TypeInitializationException
COR_E_VERIFICATION	VerificationException
COR_E_WEAKREFERENCE	WeakReferenceException
COR_E_VTABLECALLSNOTSUPPORTED	VTableCallsNotSupportedException
Cualquier otro HRESULT	COMException

Si su aplicación debe obtener información de error extendida y el objeto COM admite la interfaz `IErrorInfo`, puede usar el objeto `IErrorInfo` para conseguir información sobre la excepción. La tabla 34.3 describe la información de error adicional.

Tabla 34.3. Información de error extendida de interoperabilidad COM

Campo de la excepción	Información de la fuente COM
ErrorCode	El HRESULT devuelto por la llamada de método
HelpLink	Si <code>IErrorInfo->HelpContext</code> no es cero, la cadena se forma concatenando <code>IErrorInfo->GetHelpFile</code> y "#" y <code>IErrorInfo->GetHelpContext</code> . En caso contrario, la cadena devuelta procede de <code>IErrorInfo->GetHelpFile</code> .
InnerException	Siempre es null
Message	Cadena devuelta de <code>IErrorInfo->GetDescription</code>
Source	Cadena devuelta de <code>IErrorInfo->GetSource</code>
StackTrace	El seguimiento de la pila de esta excepción .NET
TargetSite	El nombre del método que hizo que se devolviese HRESULT a .NET

Obviamente, debe incluir control de errores en sus aplicaciones, aunque esté usando interoperabilidad COM. No hay diferencia fundamental entre el modo de codificar componentes COM y el modo de codificar ensamblados .NET, de modo que el control de excepciones estructurado de .NET debe usarse siempre que escribe código susceptible de causar una excepción.

Cómo usar la invocación de plataforma

Si es un programador de Visual Basic 6, el API Win32 ha sido su modo de aprovechar toda la potencia de la programación. En .NET, todavía puede acceder al API Win32 desde C#, aunque casi toda o toda la funcionalidad que probablemente esté usando ya estaba presente en .NET Framework. La invocación de funciones de DLL de C se consigue usando el servicio invocación de plataforma. La invocación de plataforma es un servicio que permite al código administrado llamar a funciones de DLL COM no administradas.

Mediante la clase `DLLImportAttribute`, puede especificar el nombre de la DLL y de la función DLL que debe usarse en su aplicación C#. Igual que el acceso al API Win32 en VB6, debe saber el nombre de la DLL y la función de la DLL que se quiere ejecutar. Una vez ha logrado esto, puede llamar simplemente a la función usando el atributo `DLLImport` en un método señalado con modificadores estáticos y externos, como muestra el siguiente código:

```
using System.Runtime.InteropServices;
[DllImport("user32.dll")]
public static extern int MessageBox(int hWnd, String text,
    String caption, uint type);
```

Cuando usa la invocación de plataforma, puede necesitar cambiar el comportamiento por defecto de la interoperabilidad entre el código gestionado y el no gestionado. Esto puede conseguirse modificando los campos de la clase `DLLImportAttribute`.

La tabla 34.4 describe los campos de la clase `DLLImportAttribute` que pueden ser personalizados.

Tabla 34.4. Campos `DLLImportAttribute`

Campo de objeto	Descripción
<code>EntryPoint</code>	Especifica el punto de entrada de la DLL a la que se va a llamar.
<code>CharSet</code>	Controla el modo en que los argumentos de cadena deben ser organizados para la función. El valor por defecto es <code>CharSet.Ansi</code> .
<code>ExactSpelling</code>	Evita que un punto de entrada se modifique para que corresponda con el conjunto de caracteres. El valor por defecto varía dependiendo del lenguaje de programación.
<code>CallingConvention</code>	Indica el valor de convención de llamada utilizado para pasar los argumentos del método. El valor por

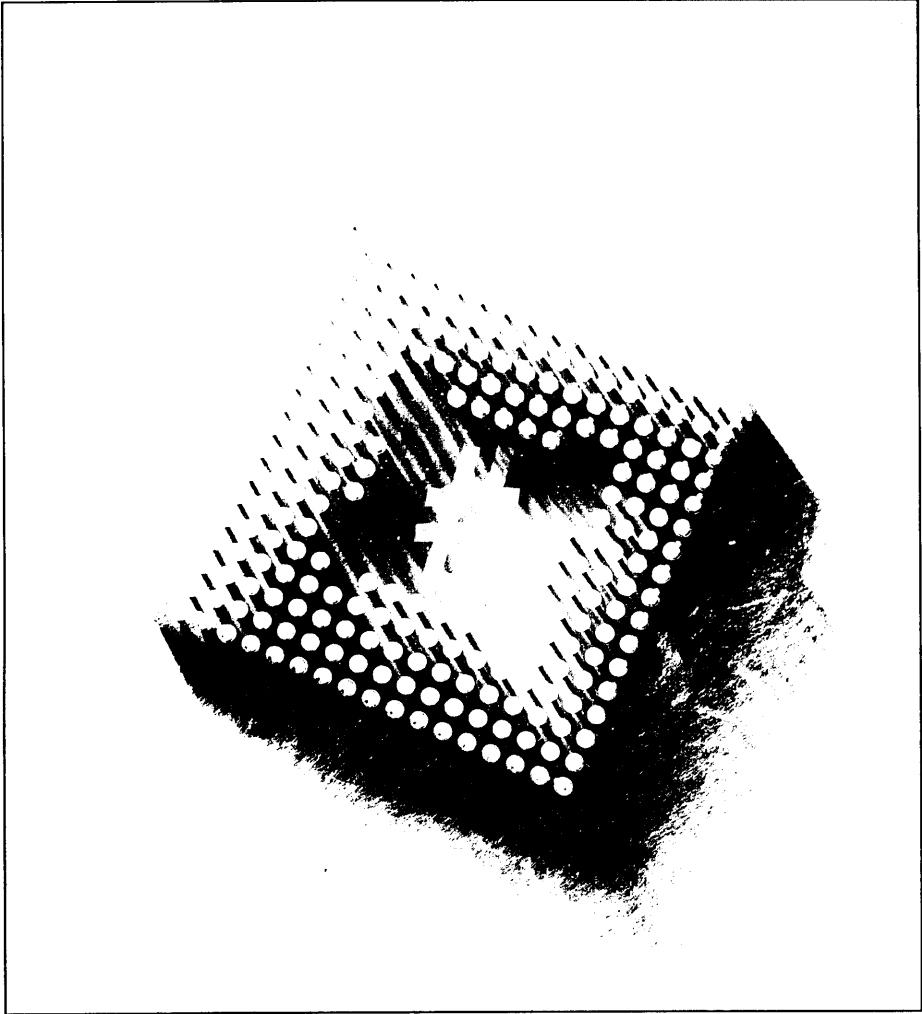
Campo de objeto	Descripción
PreserveSig	<p>defecto es WinAPI, que se corresponde a <code>__stdcall</code> para las plataformas Intel basadas en 32 bits.</p> <p>Indica que la firma administrada del método no se debe transformar en una firma no administrada que devuelve HRESULT y que puede tener un argumento adicional [out, retval] para el valor devuelto.</p> <p>El valor por defecto es True (la firma no se puede modificar).</p>
SetLastError	<p>Permite al invocador usar la función <code>API.Marshal.GetLastWin32Error</code> para determinar si se produjo un error mientras se ejecutaba el método. En Visual Basic, el valor por defecto es True; en C# y C++, el valor por defecto es False.</p>

La operación de llamar a funciones DLL desde C# es similar a hacerlo desde Visual Basic 6. Sin embargo, con el atributo `DLLImport` sólo está pasando el nombre de la DLL y el método que debe llamar.

NOTA: Se recomienda que las llamadas de función DLL estén agrupadas en clases separadas. Esto simplifica la codificación, aísla las llamadas de función externas y reduce la carga.

Resumen

Este capítulo describe cómo usar objetos COM en código .NET y cómo usar la utilidad `Tlbimp` para generar ensamblados .NET. También se estudia brevemente cómo interpretar los ensamblados generados. Además, se ha explicado cómo escribir código cliente COM en C#, incluyendo llamadas a métodos COM y a trabajar con propiedades COM. Como puede ver, .NET Framework permite integrar fácilmente el código COM existente en sus aplicaciones .NET. Esta sencilla integración le da la oportunidad de mover poco a poco partes de una aplicación a .NET, sin tener que reescribir toda la lógica de componentes COM de C#.



35 **Cómo trabajar con servicios COM+**

Microsoft ha mejorado mucho la funcionalidad del subsistema COM desde que se publicó por primera vez en 1993. Una de las mejoras más significativas introducidas en el modelo de programación COM se introdujo en 1997 con la publicación de Microsoft Transaction Server (MTS). MTS, publicado por primera vez como complemento de Windows NT 4.0, permitía a los programadores desarrollar componentes mediante un intermediario que proporcionaba transacciones, seguridad basada en funciones y servicios de agrupación de recursos.

Con la publicación de Windows 2000, Microsoft elevó el modelo de programación que ofrecía MTS a un subsistema de primera clase. COM+ es, en gran parte, una combinación del modelo de programación COM tradicional y el modelo de programación MTS. Por primera vez, Windows proporcionaba compatibilidad con los dos componentes COM (o no configurados) tradicionales con componentes atribuidos (o configurados) del estilo MTS directamente desde el sistema operativo. .NET Framework ofrece ambos estilos de componentes a los programadores que escriben software basado en componentes. Este capítulo estudia cómo desarrollar clases C# que pueden usarse como componentes configurados con COM+.

ADVERTENCIA: Aunque .NET Framework está disponible en muchas plataformas de sistemas operativos, COM+ no está disponible en el mismo

conjunto de plataformas. Los componentes escritos en C# que aprovechan los servicios COM+ sólo pueden ser usados en plataformas que admiten COM+. El código de la clase COM+ compilado en .NET Framework inicia una excepción de clase PlatformNotSupported si el código intenta acceder a una característica que no existe en la plataforma de tiempo de ejecución.

El espacio de nombres System.EnterpriseServices

Cualquier clase C# puede ser usada por clientes COM como un componente COM, independientemente del árbol de herencia de la clase. Las clases C# sólo pueden derivarse de *System.Object* y seguir siendo usadas como componentes COM. Sin embargo, aprovechar los servicios COM+ en clases C# requiere unas normas de herencia más rigurosas.

El espacio de nombres *System.EnterpriseServices* proporciona las clases, enumeraciones, estructuras, delegados e interfaces necesarias para escribir aplicaciones que aprovechen COM+ y sus servicios de nivel de empresa. Si ha escrito componentes en C++ o Visual Basic 6 que terminarán ejecutándose en un tiempo de ejecución de servicios COM+, la mayor parte de este capítulo le resultará familiar. Desde el punto de vista de un programador de COM+ experimentado, el espacio de nombres *System.EnterpriseServices* contiene la funcionalidad a la que antes tenía acceso mediante programación. Si ha escrito componentes en VB6, le alegrará saber que las características que anteriormente no estaban disponibles, como la agrupación de objetos, ahora están a su entera disposición mediante Framework. Servicios como la activación justo a tiempo (JIT), la agrupación de objetos, el procesamiento de transacciones y la administración de propiedades compartidas están disponibles como clases o atributos en el espacio de nombres *System.EnterpriseServices*.

La tabla 35.1 describe cada una de las clases disponibles en el espacio de nombres *System.EnterpriseServices*.

Tabla 35.1. Clases System EnterpriseServices

Clase	Descripción
ApplicationAccessControlAttribute	Permite establecer una configuración de seguridad para la biblioteca o aplicación de servidor que alberga la aplicación. No se puede heredar esta clase.

Clase	Descripción
ApplicationActivationAttribute	Especifica si los componentes del ensamblado se ejecutan en el proceso del creador o en un proceso del sistema.
ApplicationIDAttribute	Especifica el identificador de la aplicación (como GUID) para este ensamblado. No se puede heredar esta clase.
ApplicationNameAttribute	Especifica el nombre de la aplicación COM+ que se utilizará para la instalación de los componentes del ensamblado. No se puede heredar esta clase.
ApplicationQueuingAttribute	Habilita el uso de una cola para el ensamblado marcado y permite a la aplicación leer llamadas a métodos desde colas de Message Queue Server. No se puede heredar esta clase.
AutoCompleteAttribute	Marca el método con atributos como un objeto <code>AutoComplete</code> . No se puede heredar esta clase.
BYOT	Ajusta la clase <code>ByotServerEx</code> y las interfaces <code>DTC</code> , <code>ICreateWithTransactionEx</code> y <code>ICreateWithTipTransactionEx</code> de COM+. No se puede heredar esta clase.
ComponentAccessControlAttribute	Habilita la comprobación de seguridad en las llamadas a un componente. No se puede heredar esta clase.
COMTIIntrinsicsAttribute	Permite pasar propiedades de contexto desde el Integrador de transacciones COM (COMTI) al contexto de COM+.
ConstructionEnabledAttribute	Habilita la posibilidad de construcción de objetos COM+. No se puede heredar esta clase.
ContextUtil	Obtiene información acerca del contexto de objetos de COM+. No se puede heredar esta clase.
DescriptionAttribute	Especifica una descripción para un ensamblado (aplicación), componente, método o interfaz. No se puede heredar esta clase.

Clase	Descripción
EventClassAttribute	Marca la clase con atributos como una clase de eventos. No se puede heredar esta clase.
EventTrackingEnabledAttribute	Habilita el seguimiento de eventos para un componente. No se puede heredar esta clase.
ExceptionClassAttribute	Establece la clase de excepción de cola para la clase en cola. No se puede heredar esta clase.
IISIntrinsicsAttribute	Permite el acceso a valores intrínsecos de ASP desde <code>ContextUtil.GetNamedProperty</code> . No se puede heredar esta clase.
InterfaceQueuingAttribute	Habilita la capacidad de utilizar una cola para la interfaz marcada. No se puede heredar esta clase.
JustInTimeActivationAttribute	Habilita o deshabilita la activación justo a tiempo (JIT). No se puede heredar esta clase.
LoadBalancingSupportedAttribute	Determina si el componente participa en el equilibrio de carga, en caso de que el servicio de equilibrio de carga de componentes esté instalado y habilitado en el servidor.
MustRunInClientContextAttribute	Obliga a crear el objeto con atributos en el contexto del creador, si es posible. No se puede heredar esta clase.
ObjectPoolingAttribute	Habilita y configura el agrupamiento de objetos para un componente. No se puede heredar esta clase.
PrivateComponentAttribute	Identifica un componente como componente privado que sólo puede ser visto y activado por otros componentes de la misma aplicación. No se puede heredar esta clase.
RegistrationErrorInfo	Recupera información de error extendida sobre métodos relativos a múltiples objetos COM+. Esto también incluye métodos que instalan, importan y exportan componentes y aplica-

Clase	Descripción
RegistrationException	ciones COM+. No se puede heredar esta clase.
RegistrationHelper	Excepción que se produce cuando se detecta un error de registro.
ResourcePool	Instala y configura ensamblados en el catálogo de COM+. No se puede heredar esta clase.
SecureMethodAttribute	Almacena objetos en la transacción actual. No se puede heredar esta clase.
SecurityCallContext	Garantiza que la infraestructura realice las llamadas por medio de una interfaz para un método o para cada método de una clase cuando se utiliza el servicio de seguridad. Las clases necesitan utilizar interfaces para poder usar los servicios de seguridad. No se puede heredar esta clase.
SecurityCallers	Describe la cadena de llamadores que conducen hasta la llamada al método actual.
SecurityIdentity	Suministra una colección ordenada de identidades en la cadena de llamadas actual.
SecurityRoleAttribute	Contiene información relativa a una identidad incluida en una cadena de llamadas de COM+.
ServicedComponent	Configura una función para una aplicación o un componente. No se puede heredar esta clase.
ServicedComponentException	Representa la clase base de todas las clases que utilizan servicios de COM+.
SharedProperty	Excepción que se produce cuando se detecta un error en un componente que utiliza servicios.
SharedPropertyGroup	Obtiene acceso a una propiedad compartida. No se puede heredar esta clase.
	Representa una colección de propiedades compartidas. No se puede heredar esta clase.

Clase	Descripción
SharedPropertyGroupManager	Controla el acceso a grupos de propiedades compartidas. No se puede heredar esta clase.
SynchronizationAttribute	Establece el valor de sincronización del componente. No se puede heredar esta clase.
TransactionAttribute	Especifica el tipo de transacción que está disponible para el objeto con atributos. Los valores permitidos son miembros de la enumeración <code>TransactionOption</code> .

Si va a escribir clases que se ejecutan en servicios COM+, estará escribiendo lo que se conoce como *componentes que usan servicios*. Los componentes que usan servicios aprovechan las características del espacio de nombres `System.EnterpriseServices` y le permiten usar las características empresariales de COM+.

La clase `ServicedComponent`

Cualquier clase diseñada para aprovechar los servicios de COM+ debe derivarse directamente de `ServicedComponent` o de una clase que tenga `ServicedComponent` en alguna parte de su árbol de herencia. Todos los servicios COM+ que puede usar están disponibles asignando atributos a las clases que se derivan de la clase `ServicedComponent`.

La clase `ServicedComponent` no admite ninguna propiedad; sin embargo, admite una serie de métodos públicos que pueden ser invocados por clientes de clase. La mayor parte de estos métodos, incluyendo `Activate()`, `Deactivate()` y `CanBePooled()`, describen los métodos definidos por interfaces COM+, como `IObjectControl`. Estos métodos son virtuales y pueden ser reemplazados por clases derivadas para proporcionar funcionalidad específica.

El listado 35.1 muestra una sencilla clase COM+ escrita en C#. Este objeto participa en el agrupamiento de objetos COM+.

Listado 35.1. Componente COM+ agrupable en C#

```
using System.EnterpriseServices;

[ObjectPooling(5, 10)]
public class PooledClass : ServicedComponent
```

```

{
    public PooledClass()
    {
    }

    ~PooledClass()
    {
    }

    public override bool CanBePooled()
    {
        return true;
    }

    public override void Activate()
    {
    }

    public override void Deactivate()
    {
    }
}

```

La clase del listado 35.1 usa un atributo de .NET Framework llamado `ObjectPooling` para indicar que la clase `PooledClass` debe poder agruparse en COM+. El atributo `ObjectPooling` admite varios constructores. El listado 35.1 usa el constructor que acepta dos números enteros que representan el tamaño máximo y mínimo de la agrupación. El código usa los valores 5 y 10, lo que indica a COM+ que debe admitir un mínimo de cinco y un máximo de diez objetos de esta clase en la agrupación de objetos COM+.

Los componentes COM+ escritos en C# que quieren formar parte de una agrupación de objetos COM+ deben reemplazar al método virtual `CanBePooled()` de la clase base `ServiceComponent` y deben devolver `True`. Si se devuelve un valor `False` significa que el componente no debe formar parte de la agrupación de objetos.

Los componentes COM+ que pueden ser agrupados también pueden reemplazar a los métodos `ServiceComponent` virtuales llamados `Activate()` y `Deactivate()`. El método `Activate()` es invocado cuando el objeto es eliminado de la agrupación de objetos y asignado a un cliente y el método `Deactivate()` es invocado cuando el objeto es liberado por un cliente y devuelto a la aplicación. Debe seguir las directrices impuestas por el desarrollo de COM+ estándar y colocar todo el código relevante de destrucción y construcción de estado de objetos en los métodos `Activate()` y `Deactivate()`. El constructor y destructor de su clase son invocados, pero sólo son invocados una vez. El constructor es invocado sólo cuando COM+ crea instancias de su objeto para colocarlas en la agrupación de objetos COM+ y el destructor sólo es llamado cuando COM+ destruye el objeto tras eliminarlo de la agrupación. El método `Activate()` se diferencia del constructor en que se invoca cada vez que la

instancia se asigna a un cliente COM+. El método `Deactivate()` se diferencia del destructor en que se invoca cada vez que la instancia es liberada de un cliente COM+ y devuelta a la agrupación de objetos COM+. Si tiene código que debe realizar alguna inicialización cada vez que se le asigna a un nuevo cliente el uso del objeto, coloque el código en `Activate()`, en lugar de en su constructor de clase. Del mismo modo, si tiene algún código que deba realizar alguna desinicialización cada vez que un nuevo cliente libere el objeto, coloque el código en `Deactivate()`, en lugar de en su destructor de clase.

Cómo registrar clases con COM+

Las clases de C# diseñadas para ser usadas en una aplicación COM+ deben seguir las mismas reglas básicas que las clases de C# diseñadas para ser usadas por los clásicos clientes COM. En el capítulo anterior se describe cómo usar C# para crear componentes COM. Al igual que los componentes COM escritos en C#, los componentes COM+ escritos en C# deben compilarse en un ensamblado basado en DLL y deben tener un nombre seguro (lo que requiere que el ensamblado tenga un par de clave pública e información de versión). Al igual que los componentes COM, esta información puede ser especificada para componentes COM+ mediante atributos especificados en el código fuente C#.

Puede instalar sus clases en una aplicación COM+ mediante una herramienta de línea de comando llamada `regsvcs` que se incluye en .NET Framework. Esta herramienta de línea de comando inscribe todas las clases públicas encontradas en un ensamblado basado en DLL con COM+ y realiza todas las inscripciones necesarias para hacer que las clases sean tan visibles como las clase COM+.

El listado 35.2 es una pequeña modificación al listado 35.1. Contiene los atributos necesarios para preparar el ensamblado generado para que admita un nombre seguro.

Listado 35.2. Objeto COM+ agrupable con atributos de nombre seguro

```
using System.Reflection;
using System.EnterpriseServices;

[assembly:AssemblyKeyFile("keyfile.snk")]
[assembly:AssemblyVersion("1.0.*")]

[ObjectPooling(5, 10)]
public class PooledClass : ServicedComponent
{
    public PooledClass()
    {
    }

    ~PooledClass()
    {
    }
}
```

```

    {
    }

    public override bool CanBePooled()
    {
        return true;
    }

    public override void Activate()
    {
    }

    public override void Deactivate()
    {
    }
}

```

Puede exponer esta clase como una clase COM+ con sólo unas cuantas herramientas de línea de comando. En primer lugar, genere un nuevo par de claves para el nombre seguro del ensamblado con la herramienta de línea de comando estándar sn:

```
sn -k keyfile.snk
```

A continuación, compile el código en un ensamblado basado en DLL:

```
csc /target:library Listing35-2.cs
```

Tras generar el ensamblado, puede usar la herramienta `regsvcs` para registrar el ensamblado con COM+:

```
regsvcs /appname:Listing28-2App Listing35-2.dll
```

TRUCO: La infraestructura de interoperabilidad .NET/COM+ admite aplicaciones COM+ basadas en la caché de ensamblados global. Si varios clientes usan el código, quizás quiera instalar su ensamblado en la caché de ensamblado global antes de registrarla con COM+.

El argumento `/appname` para la herramienta `regsvcs` especifica el nombre de la aplicación COM+ creada para almacenar las clases públicas encontradas en el ensamblado. Si cuando se ejecuta `regsvcs` ya existe una aplicación COM+ con su nombre, las clases se agregan a la aplicación preexistente. La figura 35.1 muestra el explorador COM+ ejecutándose con el ensamblado generado a partir del código del listado 35.2 registrado con COM+. `PooledClass` es detectado automáticamente por el proceso de registro y agregada a la aplicación COM+.

Inicie el explorador COM+ realizando los siguientes pasos:

1. Haga clic en el botón **Inicio** del Explorador de Windows. Aparecerá el menú Inicio.

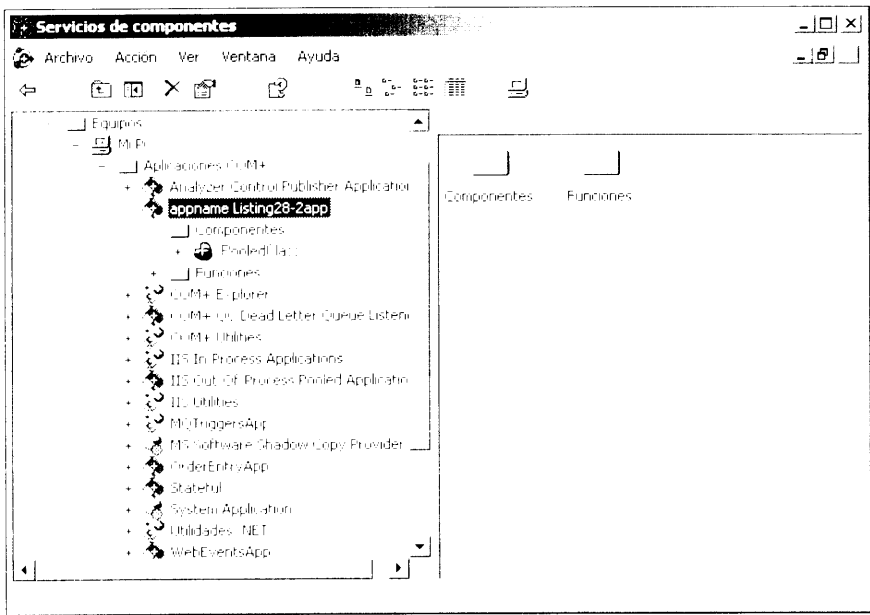


Figura 35.1. Explorador COM+ con un ensamblado .NET registrado

2. Escoja **Programas>Herramientas administrativas**. Aparecerán los iconos de las aplicaciones en el grupo de programas **Herramientas administrativas**.
3. Seleccione **Servicios de componentes**. Aparecerá el **Explorador COM+**.

La figura 35.2 muestra la hoja de propiedades de la clase `PooledClass`. Observe que la información de agrupación de objeto especificada en los atributos del listado 35.2 es detectada automáticamente por el proceso de registro y agregada a la aplicación COM+.

Cómo usar atributos para clases COM+

El atributo de agrupación de objetos usado en el listado 35.2 sólo es uno de los muchos atributos .NET que puede usar en sus clases C#. .NET Framework admite varios atributos que pueden usarse para configurar aspectos COM+ para sus clases C#. Todos los atributos .NET relativos a COM+ se incluyen en el espacio de nombres `System.EnterpriseServices`. Las siguientes secciones describen más atributos interesantes del servicio COM+.

NOTA: Para los atributos COM+ del espacio de nombres `System.EnterpriseServices`, el valor predeterminado sin configurar hace referencia al valor que COM+ asigna al atributo cuando el atributo no

aparece en el código. Un valor predeterminado configurado hace referencia al valor asignado al atributo, si se asigna, pero omite su valor.

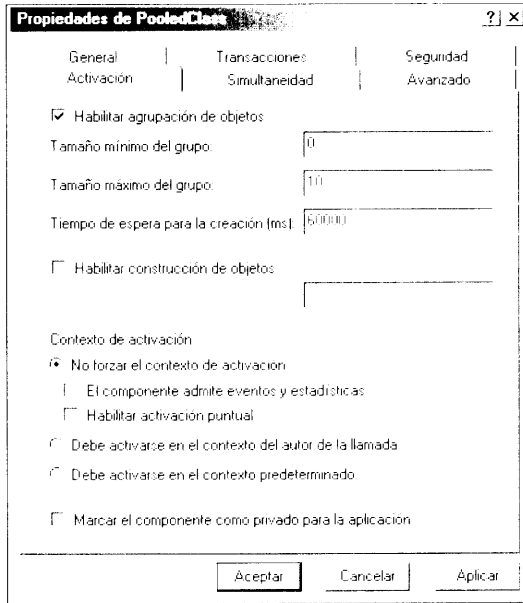


Figura 35.2. Página de la propiedad de clase COM+ con información de agrupación de objetos

ApplicationAccessControl

El atributo `ApplicationAccessControl` especifica si se puede configurar la seguridad de un ensamblado. Este atributo recibe un valor booleano que debe ser `True` si se permite la configuración de seguridad y `False` en caso contrario. El valor predeterminado no configurado es `False` y el valor predeterminado configurado es `True`.

ApplicationActivation

El atributo `ApplicationActivation` es un atributo de nivel de ensamblado que especifica si se debe agregar la clase a una biblioteca o a una aplicación de servidor COM+. El atributo recibe como parámetro un tipo de enumeración llamado `ActivationOption` que admite los siguientes valores:

- `Library`, que especifica una aplicación de biblioteca COM+.
- `Server`, que especifica una aplicación de servidor COM+.

El valor predeterminado no configurado es `Library`.

ApplicationID

El atributo `ApplicationID` puede usarse para especificar el GUID que se va a asignar a la aplicación COM+ creada para que contenga la clase COM+. El GUID se especifica usando su representación de cadena, que se envía al constructor del atributo. El atributo `ApplicationID` debe aplicarse en el nivel de ensamblado, como en el siguiente fragmento de código:

```
[assembly:ApplicationID("{E3868E19-486E-9F13-FC8443113731}")]  
public class MyClass  
{  
}
```

El atributo recibe una cadena como parámetro que describe el GUID de la aplicación.

ApplicationName

El atributo `ApplicationName` se utiliza para especificar el nombre que se va a asignar a la aplicación COM+ creada para que contenga la clase COM+. Se le debe proporcionar el nombre del constructor del atributo. Si se especifica este atributo en el código, no será necesario usar el argumento `/appname` de la herramienta de línea de comando `regsvcs`.

El atributo `ApplicationName` debe aplicarse en el nivel de ensamblado, como en el siguiente fragmento de código:

```
[assembly:ApplicationName("MyName")]  
public class MyClass  
{  
}
```

El atributo recibe como parámetro una cadena que describe el nombre de la aplicación y su valor predeterminado es el nombre del ensamblado para un valor predeterminado no configurado.

ApplicationQueuing

El atributo `ApplicationQueuing` se usa para especificar que la clase debe ser configurada como un componente en cola. El atributo no admite parámetros. El atributo `ApplicationQueuing` debe aplicarse en el nivel de ensamblado, como en el siguiente fragmento de código:

```
[assembly:ApplicationQueuing]  
public class MyClass  
{  
}
```

El atributo no admite parámetros.

AutoComplete

El atributo `AutoComplete` puede aplicarse a métodos en una clase COM+. Si la llamada de método se hace en el ámbito de una transacción y la llamada de método se completa con normalidad, las llamadas a métodos marcados como métodos `AutoComplete` van inmediatamente seguidas por una llamada de .NET Framework a `SetComplete()`. No es necesario realizar una llamada explícita a `SetComplete()` para los métodos `AutoComplete`. Si un método `AutoComplete()` inicia una excepción, se invoca a `SetAbort()` y la transacción es eliminada. El atributo `AutoComplete` debe aplicarse en el nivel de método, como en el siguiente fragmento de código:

```
public class MyClass
{
    [AutoComplete]
    public MyMethod()
    {
    }
}
```

El atributo `AutoComplete` no acepta parámetros. El valor por defecto para el valor predeterminado no configurado es `False` y para el valor predeterminado configurado es `True`.

ComponentAccessControl

El atributo `ComponentAccessControl` activa o desactiva las comprobaciones de seguridad en las llamadas a instancias de clase. El atributo recibe como parámetro un valor booleano, que debe ser `True` si la comprobación del nivel de seguridad de la llamada debe estar activa o `False` en caso contrario.

El atributo `ComponentAccessControl` debe aplicarse en el nivel de clase, como en el siguiente fragmento de código:

```
[ComponentAccessControl]
public class MyClass
{
}
```

El atributo `ComponentAccessControl` no acepta parámetros. El valor por defecto para el valor predeterminado no configurado es `False` y para el valor predeterminado configurado es `True`.

ConstructionEnabled

El atributo `ConstructionEnabled` permite la construcción de objetos COM+. El mecanismo de construcción de objetos COM+ permite que se pueda

pasar una cadena como cadena de constructor a las instancias de objetos con instancias. El atributo no especifica la cadena; en cambio, simplemente permite que se admita la construcción de objetos COM+. El atributo admite un valor booleano como parámetro, que debe ser `True` si la construcción de objetos COM+ debe estar activada para la clase y `False` en caso contrario. Las clases de C# que admiten la construcción de objetos deben implementar la interfaz `IObjectConstruct`. El método `Construct()` de la interfaz es invocado por COM+ para pasar la cadena de constructor al objeto.

El atributo `ConstructionEnabled` debe aplicarse en el nivel de clase, como en el siguiente fragmento de código:

```
[ConstructionEnabled]
public class MyClass
{
}
```

El atributo `ConstructionEnabled` no admite parámetros. El valor por defecto del valor predeterminado no configurado es `False` y el del valor predeterminado configurado es `True`.

JustInTimeActivation

El atributo `JustInTimeActivation` habilita o deshabilita la activación justo a tiempo (JIT) de una clase. El atributo admite un valor booleano como parámetro, que debe ser `True` si se va a permitir la activación JIT en la clase y `False` en caso contrario. La activación JIT siempre debe estar habilitada en los objetos que toman parte en transacciones.

El atributo `JustInTimeActivation` debe aplicarse en el nivel de clase, como en el siguiente fragmento de código:

```
[JustInTimeActivation]
public class MyClass
{
}
```

El atributo `JustInTimeActivation` no admite parámetros. El valor por defecto para el valor predeterminado no configurado es `False` y para el valor predeterminado configurado es `True`.

LoadBalancingSupported

El atributo `LoadBalancingSupported` habilita o deshabilita la compatibilidad con el equilibrio de cargas de una clase. El atributo admite un valor booleano como parámetro que debe ser `True` si la compatibilidad con el equilibrio de cargas va a estar habilitada y `False` en caso contrario.

El atributo `LoadBalancingSupported` debe aplicarse en el nivel de clase, como en el siguiente fragmento de código:

```
{LoadBalancingSupported}
public class MyClass
{
}
```

El atributo `LoadBalancingSupported` no admite parámetros. El valor por defecto para el valor predeterminado no configurado es `False` y para el valor predeterminado configurado es `True`.

SecurityRole

El atributo `SecurityRole` especifica una función de seguridad. Este atributo puede aplicarse a una clase, un método o un ensamblado completo. El constructor recibe como argumento una cadena que debe especificar el nombre de la función a la que deben pertenecer los miembros, como muestra el siguiente fragmento de código:

```
{assembly:SecurityRole("MySecurityRole")}
public class MyClass
{
}
```

Cómo procesar transacciones

La compatibilidad de transacciones de COM+ fue uno de los principales responsables de su popularidad. Gracias a la compatibilidad de transacciones, puede escribir código que realice más de una tarea, pero la aplicación lo ve como una sola unidad de trabajo, como actualizar una base de datos y borrar un registro en otra tabla de una base de datos completamente distinta. Con las transacciones, puede garantizar que se va a aplicar un modelo de todo o nada en estas situaciones. Si la eliminación falla en la segunda base de datos, la actualización de la primera base de datos también es cancelada.

Sin las transacciones, los datos podrían no compararse correctamente y no podría escribir aplicaciones de nivel de empresa. Una de las primeras aplicaciones de ejemplo en usar la compatibilidad de transacciones que publicó Microsoft se llamó ExAir. ExAir es una compañía aérea ficticia. El concepto básico tras la aplicación es un agente de billetes que recibe reservas de vuelos. Cuando un cliente solicita un vuelo, también debe solicitar un tipo de comida, como carne, vegetariana o pasta. La parte de comida de la aplicación intenta introducir los datos en una base de datos distinta de la base de datos con el diagrama de asientos de la compañía aérea. La base de datos diferente representa una transacción dis-

tribuida a otra compañía, el proveedor de alimentos. Si el primer método contiene código que introduce la petición de billete en la base de datos de la línea aérea, y un segundo método contiene código que intenta introducir datos en la base de datos del proveedor de alimentos, ¿qué ocurrirá si no se encuentra la base de datos del proveedor de alimentos? La información original que contiene los detalles del vuelo se introducirá en la base de datos de la compañía aérea, pero cuando los pasajeros aparezcan, no tendrán comida, porque la segunda parte de la transacción no se pudo realizar. Evidentemente, esto no es una situación deseable para una aplicación de empresa. Si el proveedor de alimentos no está disponible, el billete no se introducirá en la base de datos de la compañía aérea. Este problema puede solucionarse fácilmente envolviendo las dos llamadas de método en una transacción.

NOTA: En el ejemplo ExAir, a la compañía no le interesa dejar de reservar un billete sólo porque el vínculo a la base de datos de su proveedor de alimentos no funciona. En esta situación, la solución podría ser usar algo como los componentes en cola o Microsoft Message Queue. Con los servicios Message Queue, que se incluyen en el espacio de nombres System.Messaging, puede garantizar la petición eventual de envío de comida enviándola a una cola de mensajes en lugar de intentar escribir inmediatamente los datos en la base de datos remota. Con este tipo de arquitectura, la aplicación siempre puede aceptar las peticiones de billetes y el proveedor sólo tendrá que extraer mensajes de la cola de mensajes cuando pueda procesar los pedidos de alimentos.

Propiedades ACID

Para que funcionen las transacciones, deben atenerse a las propiedades ACID. ACID es el acrónimo de atomicidad, coherencia, aislamiento y permanencia. La tabla 35.2 describe las definiciones de las propiedades ACID.

Tabla 35.2. Propiedades ACID

Propiedad	Descripción
Atomicidad	Todo el trabajo es atómico o sucede en una sola unidad de trabajo.
Coherencia	Todo dato que se usa en la transacción es abandonado en un estado consistente.
Aislamiento	Cada transacción está aislada de las otras transacciones, lo que permite que las transacciones pue-

Propiedad	Descripción
Permanencia	<p>dan sobrescribir los datos de otros procesos, lo que mantiene la uniformidad.</p> <p>Cuando se completa la transacción, todos los datos deben estar en un almacén permanente, como una base de datos. Si la transacción falla, todos los datos usados en ella deben ser eliminados. La permanencia garantiza que los datos sobrevivan a acontecimientos imprevistos, como cortes de luz o huracanes.</p>

Cómo escribir componentes de transacciones

El atributo `Transaction` especifica un nivel de compatibilidad con transacciones que debe estar disponible para este objeto. El atributo recibe como parámetro un valor de una enumeración en el espacio de nombres `System.EnterpriseServices` llamado `TransactionOption`, que es compatible con cualquiera de los siguientes valores:

- `Disabled`, que especifica que el objeto debe pasar por alto cualquier transacción en el contexto actual
- `NotSupported`, que especifica que el objeto debe crear el componente en un contexto sin una transacción que lo controle
- `Required`, que especifica que el objeto debe compartir una transacción si ya existe una o crear una nueva transacción en caso de que sea necesario
- `RequiresNew`, que especifica que el objeto debe crear el componente con una nueva transacción, independientemente del estado del actual contexto
- `Supported`, que especifica que el objeto debe compartir una transacción si ya existe una

El atributo `Transaction` debe aplicarse en el nivel de clase, como en el siguiente fragmento de código:

```
[Transaction(TransactionOption.Supported)]
public class MyClass
{
}
```


El atributo recibe un solo parámetro que nombra a un valor de la enumeración `TransactionOption` y describe el nivel de transacción que admite el diseño de la clase. El atributo también puede usarse sin parámetros, como en el siguiente fragmento de código:

```
[Transaction]
public class MyClass
{
}
```

Si se especifica el atributo `Transaction` sin especificar un parámetro, el nivel de compatibilidad con el parámetro de la clase pasa a ser `Required`.

El listado 35.3 muestra el código completo para el contexto de `ExAir`. Existen dos métodos y cada uno accede a recursos diferentes. Como están contenidos en una transacción, queda garantizado que el procesamiento del pedido tiene lugar como una sola unidad de trabajo; el proceso está aislado de los otros posibles pedidos; los datos del pedido se dejan en un estado coherente; y una vez que el pedido está consignado, permanecen en un almacén permanente.

Listado 35.3. Ejemplo de base de datos transaccional

```
namespace TransactionSupport
{

    using System;
    using System.Data.SqlClient;
    using System.EnterpriseServices;

    [Transaction(TransactionOption.Required)]
    public class ExAirMain : ServicedComponent
    {
        public void Process()
        {
            /* llama a metodos para agregar informacion de Food y de
            Ticket */

            AddFood process1 = new AddFood();
            AddAirline process2 = new AddAirline();
            process1.Add();
            process2.Add();

        }
    }

    [Transaction(TransactionOption.Supported)]
    [AutoComplete]
    public class AddFood : ServicedComponent
    {
        public void Add()
        {

```

```

SqlConnection cnn = new
    SqlConnection("FoodSupplierConnection");
SqlCommand cmd = new SqlCommand();
cnn.Open();
cmd.ActiveConnection = cnn;
cmd.CommandText = ""; // Introduce una instruccion en DB
cmd.ExecuteNonQuery();
cnn.Close();
}
}

[Transaction(TransactionOption.Supported)]
[AutoComplete]
public class AddAirline : ServicedComponent
{
public void Add()
{
SqlConnection cnn = new
SqlConnection("AirlineConnection");
SqlCommand cmd = new SqlCommand();
cnn.Open();
cmd.ActiveConnection = cnn;
cmd.CommandText = "" // Introduce una instruccion en DB
cmd.ExecuteNonQuery();
cnn.Close();
}
}
}

```

Cómo acceder al contexto de objetos

El espacio de nombres `System.EnterpriseServices` incluye una clase llamada `ContextUtil` que puede ser utilizada por clases C# para acceder a un contexto de tiempo de ejecución COM+ de un objeto. En Visual Basic 6, se accede al contexto de objetos del componente en uso mediante el objeto `ObjectContext`, como muestra el siguiente código:

```

Dim ctx as ObjectContext
ctx = GetObjectContext

```

La clase `ContextUtil` contiene varias propiedades y métodos que conceden a los invocadores el acceso a información de estado de contexto COM+. Todos los métodos y propiedades de la clase son estáticos, lo que significa que se puede acceder a los miembros directamente desde la clase `ContextUtil` sin crear un objeto de la clase. La tabla 35.3 describe las propiedades de la clase `ContextUtil` y la tabla 35.4 describe los métodos de la clase `ContextUtil`.

El código del listado 35.4 implementa un componente COM+ transaccional que implementa un método público llamado `DoWork()`. El método `DoWork()`

comprueba la propiedad `IsCallerInRole()` para determinar la función COM+ del invocador. Si la función del invocador es `ClientRole`, entonces la transacción del objeto se realiza con una llamada a `SetComplete()`. Si la función del invocador es distinta de la de `ClientRole`, entonces la transacción del objeto es cancelada con una llamada a `SetAbort()`.

Tabla 35.3. Propiedades de la clase `ContextUtil`

Propiedad	Descripción
<code>ActivityId</code>	Obtiene un identificador GUID que representa la actividad que contiene el componente
<code>ApplicationId</code>	Obtiene un identificador GUID para la aplicación actual
<code>ApplicationInstanceId</code>	Obtiene un identificador GUID para la instancia de aplicación actual
<code>ContextId</code>	Obtiene un identificador GUID para el contexto actual
<code>DeactivateOnReturn</code>	Obtiene o establece el bit hecho en el contexto de COM+
<code>IsInTransaction</code>	Obtiene un valor que indica si el contexto actual es transaccional
<code>IsSecurityEnabled</code>	Obtiene un valor que indica si la seguridad basada en funciones está activa en el contexto actual
<code>MyTransactionVote</code>	Obtiene o establece el bit consistente en el contexto de COM+
<code>PartitionId</code>	Obtiene un identificador GUID para la partición actual
<code>Transaction</code>	Obtiene un objeto que describe la transacción de DTC actual de COM+
<code>TransactionId</code>	Obtiene el identificador GUID de la transacción de DTC actual de COM+.

Tabla 35.4. Propiedades de la clase `ContextUtil`

Propiedad	Descripción
<code>DisableCommit</code>	Asigna a los bits consistente y hecho el valor <code>False</code> en el contexto de COM+.
<code>EnableCommit</code>	Asigna al bit consistente el valor <code>True</code> , y al bit hecho el valor <code>False</code> , en el contexto de COM+

Propiedad	Descripción
GetNamedProperty	Devuelve una propiedad con nombre desde el contexto COM+
IsCallerInRole	Determina si el llamador se incluye en la función especificada
SetAbort	Asigna al bit consistente el valor <code>False</code> , y al bit hecho el valor <code>True</code> , en el contexto de COM+
SetComplete	Asigna a los bits consistente y hecho el valor <code>True</code> en el contexto de COM+.

Listado 35.4. Cómo acceder al contexto COM+ mediante la clase `ContextUtil`

```
using System.Reflection;
using System.EnterpriseServices;

[assembly:AssemblyKeyFile("keyfile.snk")]
[assembly:AssemblyVersion("1.0.*")]

[ObjectPooling(5, 10)]
[Transaction(TransactionOption.Required)]
[SecurityRole("ClientRole")]

public class PooledClass : ServicedComponent
{
    public PooledClass()
    {
    }

    ~PooledClass()
    {
    }

    public override bool CanBePooled()
    {
        return true;
    }

    public override void Activate()
    {
    }

    public override void Deactivate()
    {
    }

    public void DoWork()
    {
        bool IsInRole;
    }
}
```

```
IsInRole = ContextUtil.IsCallerInRole("ClientRole");  
if(IsInRole == true)  
    ContextUtil.SetComplete();  
else  
    ContextUtil.SetAbort();
```

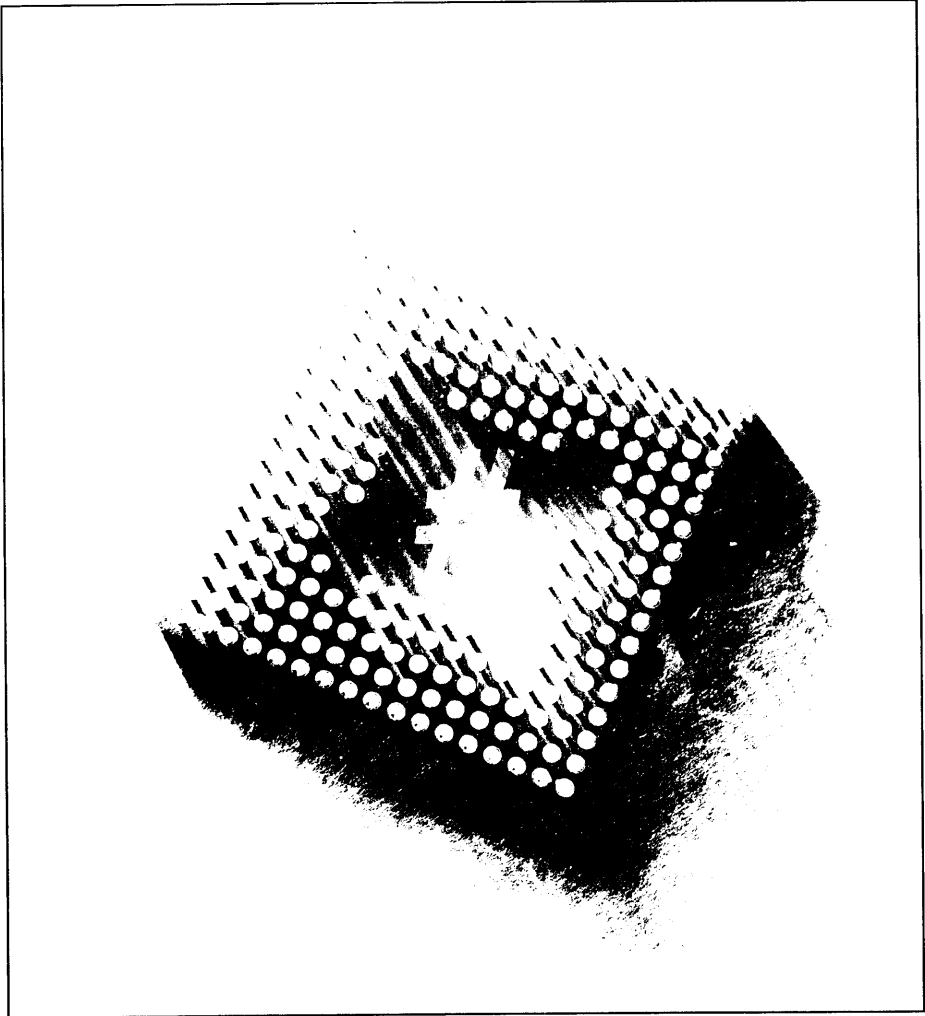
Resumen

La exposición de una clase C# como si fuera una aplicación COM+ no supone ningún esfuerzo y resulta más sencillo que implementar la misma funcionalidad usando versiones anteriores de Visual Studio 6.0. Las aplicaciones de COM+ escritas en Visual C++ 6.0 necesitaban mucho más código para realizar las mismas tareas y algunas características de COM+ (como la agrupación de objetos) ni siquiera estaban disponibles en Visual Basic 6.0.

El desarrollo de componentes COM+ mediante C# implica cuatro sencillos conceptos:

- Derivar la clase de `ServiceComponent`
- Agregar atributos para describir las configuraciones de la aplicación
- Usar la herramienta `regsvcs` para construir una aplicación COM+ para las clases públicas
- Invocar a métodos y propiedades en la clase `ContextUtil` para acceder al contexto COM+ en el tiempo de ejecución.

Microsoft ofreció pistas sobre este modelo de programación COM+ hasta 1997. Entonces, describieron un modelo basado en programación con atributos, en el que los componentes COM deberían ser descritos con atributos y el tiempo de ejecución se encargaría de detalles como los generadores de clases y el cálculo de referencias al estilo `IUnknown`. Ahora es evidente que el modelo .NET de desarrollo de componentes COM+ es la culminación de esa visión original.



36 **Cómo trabajar con los servicios remotos de .NET**

.NET Framework proporciona varios mecanismos que le permiten escribir aplicaciones que no existen en el mismo dominio de aplicación, procesos de servidor o equipo. Basándose en los requisitos de la aplicación, como la capacidad de los servidores que no sea de .NET para acceder a sus datos, puede escoger cualquiera de los diferentes tipos de métodos de comunicación de objetos. En este capítulo, estudiaremos el entorno remoto .NET. Al usar un entorno remoto, puede recodificar objetos y llamadas de métodos a lo largo de límites de procesos y pasar datos entre aplicaciones eficazmente. Anteriormente aprendimos que los servicios Web de ASP.NET y XML también eran excelentes medios para pasar objetos y datos entre límites de procesos, pero dependiendo de la infraestructura de la aplicación, esos servicios podían no ser la mejor opción disponible. El entorno remoto se ocupa de los aspectos que no cubrían estos servicios. En este capítulo, aprenderá a implementar entornos remotos, a crear los objetos cliente y servidor de un entorno remoto y a pasar datos usando el entorno remoto a través de límites de procesos.

Introducción al entorno remoto

El entorno remoto .NET permite a las aplicaciones comunicarse entre objetos que están en servidores diferentes, procesos diferentes o dominios de aplicación

diferentes. Antes de la llegada de .NET Framework, se podían pasar objetos a través de límites de procesos mediante COM o DCOM. DCOM funcionaba bien, pero tenía limitaciones, como los tipos de datos que podían pasarse y el contexto de seguridad pasado entre el llamador cliente y la activación del servidor. Además, estaba basado en COM, lo que significaba que aunque podía comunicarse a través de límites de equipo, todos los equipos debían estar ejecutando un sistema operativo de Microsoft. Esto no era una gran limitación, pero limitaba nuestras opciones respecto a lo que podíamos hacer con la infraestructura existente. En .NET, el entorno remoto se ocupa de estos aspectos y mejora lo que DCOM ofrecía como un método viable para establecer una comunicación remota entre objetos.

El entorno remoto permite implementar un servidor o una aplicación de servidor y una aplicación cliente. En el servidor o el cliente, la aplicación puede ser cualquiera de las plantillas de aplicación .NET disponibles, incluyendo aplicaciones de consola, aplicaciones de servicios Windows, aplicaciones ASP.NET, aplicaciones WindowsForms y aplicaciones IIS. En el servidor, se configura mediante programación (o se emplea un archivo de configuración para especificarlo) el tipo de activación que permitirán los clientes. Los clientes pueden usar uno de los varios tipos de métodos de activación, incluyendo `Singleton` y `SingleCall`, como se explicará en la sección "Cómo activar el objeto remoto," más tarde en este mismo capítulo. Es en este punto donde se especifica el canal y el puerto a través de los cuales se comunica el objeto y el formato que tendrán los datos cuando pasen entre el servidor y el cliente. Aprenderá a implementar canales y puertos poco después. El formato de los datos es importante, según el diseño del sistema: puede usar datos binarios, SOAP o un formato personalizado, para recodificar los datos. Tras especificar el canal, el puerto y el formato, según el tipo de servidor remoto que esté exponiendo, debe determinar cómo exponer los metadatos a los clientes. Puede hacerlo de varias maneras, como permitiendo al invocador descargar el ensamblado o haciendo que la fuente esté disponible para el invocador.

En cualquiera de los dos casos, el cliente debe saber qué objeto está creando, de modo que los metadatos en varias formas deben estar disponibles para el invocador. Cuando el servidor está configurado y creado adecuadamente, puede escribir el cliente. En el cliente, todo lo que debe hacer es crear una instancia del objeto en el canal y puerto especificados que esperen peticiones del servidor. Puede lograr esto mediante programación o mediante un archivo de configuración. En este punto, las llamadas de método no son diferentes de cualquier otro objeto de una aplicación .NET que se pueda usar. Tras crear objetos, llame a los métodos, establezca y recupere propiedades y desencadene eventos igual que haría con un objeto que no esté usando el entorno remoto.

Estos pueden parecer muchos pasos, pero en realidad es muy sencillo una vez que lo ha hecho una vez. Puede resumir todo el proceso en las siguientes tareas, esquematizadas en la figura 36.1.

1. Especifique los canales y puertos que recodifican los objetos entre el servidor y el cliente.
2. Use formateadores (explicados más adelante en este capítulo) para especificar el formato en el que los datos son serializados y deserializados entre el servidor y el cliente.
3. Determine cómo se activan los objetos del servidor y cuánto dura la activación.

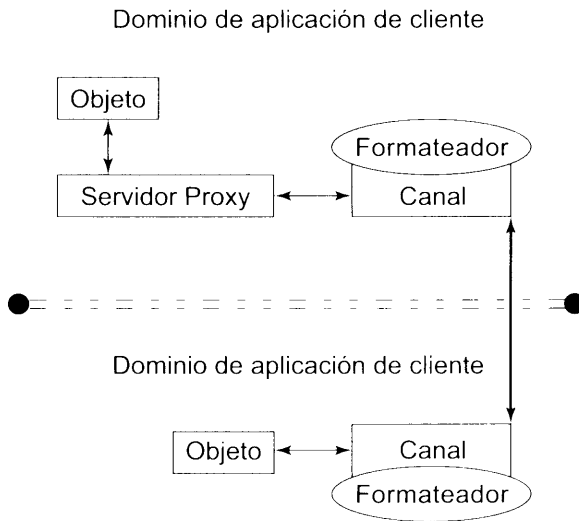


Figura 36.1. Vista del entorno remoto .NET

En las siguientes secciones, aprenderá a crear la aplicación anfitriona en un contexto remoto, incluyendo los detalles específicos de los formateadores, canales y puertos y cómo se puede activar el servidor. Después de construir el servidor, aprenderá a consumir el objeto remoto de una aplicación cliente.

Cómo crear un ensamblado de servidor remoto

Para empezar con la aplicación remota, necesita crear un ensamblado que contenga las llamadas de método reales que usará la aplicación anfitriona. Una vez creado el ensamblado, cree la aplicación anfitriona que acepte peticiones de los métodos del ensamblado por parte de los clientes. En los siguientes pasos, creará el ensamblado que implementa los métodos que se van a llamar:

1. Cree una nueva aplicación C# de Biblioteca de clases y llámela `ServidorObject`. Para hacerlo más sencillo, yo creé un directorio en

mi unidad C llamado `cSharpRemoting` y le agregué tres subcarpetas llamadas `Servidor`, `ServidorObject` y `Client`. Ya se imaginará lo que se pretende. La aplicación de biblioteca de clases `ServidorObject` debe ser creada en el directorio `cSharpRemoting\ServidorObject`. Esto hace que le sea más sencillo ejecutar las aplicaciones de consola que creará posteriormente.

2. Tras crear la aplicación de biblioteca de clases `ServidorObject`, agregue un método público que reciba un parámetro, llamado `customerID` y devuelva el nombre del cliente de la base de datos `Northwind` en el SQL Server basado en el `customerID` que se ha pasado. La clase completa para la aplicación `ServidorObject` debería tener un aspecto parecido al listado 36.1.

Listado 36.1. Cómo crear la aplicación `ServidorObject`

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace ServidorObject
{
    public class Class1: MarshalByRefObject
    {
        public string thisCustomer;

        public Class1()
        {
            Console.WriteLine("ServidorObject has been activated");
        }

        public string ReturnName(string customerID)
        {
            // Crea una conexcion, envia al objeto al SQL

            string cnStr = "Initial Catalog=Northwind;Data" +
                "Source=localservidor;Integrated Security=SSPI;";

            SqlConnection cn = new SqlConnection(cnStr);

            string strSQL =
                ("Select CompanyName from Customers " +
                 " where CustomerID = '" + customerID + "'");

            SqlCommand cmd = cn.CreateCommand();

            cmd.CommandText = strSQL;

            cn.Open();

            SqlDataReader rdr = cmd.ExecuteReader
```

```

        (CommandBehavior.CloseConnection);

while (rdr.Read())
{
    thisCustomer = rdr.GetString(0);
}

Console.WriteLine(thisCustomer +
    " was returned to the client.");

return thisCustomer;
}
}

```

El código anterior realiza una simple petición a SQL Server para tomar el campo `CompanyName` en la base de datos `Customers` basada en el parámetro `customerID`, que se pasa al método. Como puede ver, este código no es diferente de cualquier otra biblioteca de clases que haya creado en C#. El siguiente paso es crear la aplicación servidor que atiende a las peticiones de esta clase de biblioteca por parte del cliente.

Cómo crear un servidor remoto

Para crear la aplicación que distribuirá el ensamblado `ServidorObject`, que es donde realmente empezará a usar algunas de las características remotas creadas en el listado 36.1, debe crear una aplicación de consola llamada `Servidor` en el directorio `C:\cSharpRemoting\Servidor`. Esta aplicación anfitriona es el auténtico servidor remoto que usa las características del espacio de nombres `System.Runtime.Remoting`.

Antes de empezar a codificar, deben describirse varias características clave del entorno remoto. El espacio de nombres que contiene la funcionalidad remota es `System.Runtime.Remoting`, cuyas clases se describen en la tabla 36.1.

Tabla 36.1. Clases de `System.Runtime.Remoting`

Clase	Descripción
<code>ActivatedClientTypeEntry</code>	Almacena valores de un tipo de objeto registrado en el cliente como un tipo que puede activarse en el servidor
<code>ActivatedServiceTypeEntry</code>	Almacena valores de un tipo de objeto registrado en el servicio como un tipo que puede activarse cuando se solicita desde un cliente

Clase	Descripción
ObjectHandle	Ajusta referencias de objetos calculadas por valor, de este modo, se pueden devolver a través de un direccionamiento indirecto
ObjRef	Almacena toda la información relevante necesaria para generar un proxy y establecer comunicación con un objeto remoto
RemotingConfiguration	Proporciona varios métodos estáticos para configurar la infraestructura remota
RemotingException	Excepción que se inicia cuando se produce algún tipo de error durante la interacción remota
RemotingServices	Proporciona varios métodos para utilizar y publicar servidores proxy y objetos remotos. No se puede heredar esta clase
RemotingTimeoutException	Excepción que se inicia cuando no se puede obtener acceso al servidor o al cliente en el período de tiempo previamente especificado
ServerException	Excepción que se inicia para comunicar errores al cliente cuando éste se conecta a aplicaciones distintas de .NET Framework que no pueden iniciar excepciones
SoapServices	Proporciona varios métodos para utilizar y publicar objetos remotos en formato SOAP
TypeEntry	Implementa una clase base que contiene la información de configuración utilizada para activar una instancia de un tipo remoto
WellKnownClientTypeEntry	Contiene valores de un tipo de objeto registrado en el cliente como objeto de tipo conocido (llamada única o singleton)
WellKnownServiceTypeEntry	Contiene valores de un tipo de objeto registrado en el servicio como objeto de tipo conocido (llamada única o singleton)

Aunque no use todas estas clases cuando escriba aplicaciones remotas, varias de estas clases son extremadamente importantes para implementar una infraestructura remota: es decir, la clase `ObjRef`, la clase `RemotingConfiguration`, la clase `RemotingServices` y la enumeración `WellKnownObjectMode`. Aprenderá más de cada una de ellas más tarde en esta misma sección mientras escribe el código de su aplicación anfitriona.

Para empezar a escribir la aplicación anfitriona, debe comprender lo que la infraestructura remota necesita para funcionar. Para recordarle los pasos necesarios para crear la aplicación anfitriona, revise los siguientes pasos reseñados con anterioridad:

1. Especifique los canales y puertos que recodifican los objetos entre el servidor y el cliente.
2. Use formateadores para especificar el formato en el que los datos son serializados y deserializados entre el servidor y el cliente.
3. Determine cómo se activan los objetos del servidor y cuánto dura la activación.

Las siguientes secciones estudian cada uno de estos pasos.

Cómo especificar canales y puertos

En la infraestructura remota, los canales procesan el transporte de mensajes o datos entre los objetos cliente y servidor. Recuerde lo que está sucediendo realmente cuando está usando objetos remotos: está cruzando un límite, como un dominio de aplicación, un proceso de servidor o un equipo físico. El canal específico que proporciona controla todos los detalles subyacentes de trasladar los datos a y desde los objetos remotos; simplemente especifique un tipo de canal y le harán todo el trabajo sucio. La clase `System.Runtime.Remoting.Channels` proporciona las implementaciones para crear los canales que serán usados en el servidor remoto. Cuando registra un canal en su aplicación, debe asegurarse de que se registra antes de intentar acceder a los objetos remotos. Si no registra correctamente los canales se producirá un error. Si otra aplicación está escuchando en el canal al que esté intentando escuchar, se produce un error y su aplicación anfitriona no se cargará. Debe saber qué canales se están usando y qué canal debe usar su aplicación, basándose en las solicitudes del cliente. Tras declarar una instancia del tipo de canal que va a usar, llame al método `RegisterChannel()` de la clase `ChannelServices`, que registra el canal para su uso. La tabla 36.2 describe los métodos de la clase `ChannelServices` disponibles.

Tabla 36.2. Métodos de la clase `ChannelServices`

Método	Descripción
<code>AsyncDispatchMessage</code>	De forma asincrónica, envía el mensaje especificado a las cadenas del servidor, en función de la dirección URI incrustada en el mensaje

Método	Descripción
CreateServerChannelSinkChain	Crea una cadena de receptores de canal para el canal especificado
DispatchMessage	Envía las llamadas remotas de entrada
GetChannel	Devuelve un canal registrado con el nombre especificado
GetChannelSinkProperties	Devuelve una <code>IDictionary</code> de propiedades para un proxy determinado
GetUrlsForObject	Devuelve una matriz de todas las direcciones URL que pueden utilizarse para alcanzar el objeto especificado
RegisterChannel	Registra un canal con los servicios de canal
SyncDispatchMessage	De forma asincrónica, envía el mensaje de entrada a las cadenas del servidor, en función de la dirección URI incrustada en el mensaje
UnregisterChannel	Anula el registro de un canal determinado de la lista de canales registrados

En esta tabla no aparece una propiedad de la clase `ChannelServices` llamada `RegisteredChannels`, que obtiene o asigna los canales registrados de la actual instancia de objetos.

El siguiente fragmento de código crea y registra un canal TCP y otro HTTP en los puertos específicos usando el método `RegisterChannel` de la clase `ChannelServices`:

```
TcpChannel chan1 = new TcpChannel(8085);
ChannelServices.RegisterChannel(chan1);

HttpChannel chan2 = new HttpChannel(8086);
ChannelServices.RegisterChannel(chan2);
```

Cuando crea un canal, también especifica un tipo de formateador para el canal. Las siguientes secciones describen los tipos de formateadores disponibles.

Cómo especificar un formato de canal

Al mismo tiempo que crea un canal, también especifica un formato para el tipo de canal escogido. En el espacio de nombres `System.Runtime.Remoting.Channels` hay dos formateadores predeterminados disponibles: el canal TCP y el canal HTTP.

Espacio de nombres System.Runtime.Remoting.Channels.Tcp

El espacio de nombres `System.Runtime.Remoting.Channels.Tcp` contiene canales que usan el protocolo TCP para transportar datos entre objetos remotos. La codificación predeterminada para TCP es la codificación binaria, lo que hace que sea un modo eficaz de pasar datos entre objetos remotos. Los datos binarios siempre ocupan menos espacio que los datos XML equivalentes pasados mediante SOAP en un canal HTTP. La desventaja de usar el protocolo TCP es que es un formato propietario, de modo que sólo funciona en sistemas que comprenden este tipo de formato. Para que el objeto remoto sea más accesible debe usar el canal HTTP para codificar, ya que estará pasando datos en el protocolo SOAP. La tabla 36.3 resume las clases disponibles en el espacio de nombres `System.Runtime.Remoting.Channels.Tcp`.

Tabla 36.3. Espacio de nombres System.Runtime.Remoting.Channels.Tcp

Clase	Descripción
<code>TcpChannel</code>	Proporciona una implementación de un canal emisor-receptor que utiliza el protocolo TCP para transmitir mensajes. Esta clase es una combinación de la clase <code>TcpClientChannel</code> y la clase <code>TcpServerChannel</code> , lo que permite la comunicación en ambos sentidos a través de TCP.
<code>TcpClientChannel</code>	Proporciona una implementación de un canal de cliente que utiliza el protocolo TCP para transmitir mensajes.
<code>TcpServerChannel</code>	Proporciona una implementación de un canal de servidor que utiliza el protocolo TCP para transmitir mensajes.

Espacio de nombres System.Runtime.Remoting.Channels.Http

El espacio de nombres `System.Runtime.Remoting.Channels.Http` contiene canales que usan el protocolo HTTP para transportar datos entre objetos remotos. La codificación predeterminada para el protocolo HTTP es SOAP, lo que le convierte en un modo versátil de pasar datos entre objetos remotos. La tabla 36.4 resume las clases disponibles en el espacio de nombres `System.Runtime.Remoting.Channels.Http`.

Hasta aquí, puede agregar los espacios de nombres correctos y crear código para registrar un canal HTTP y un canal TCP para la aplicación anfitriona. El listado 36.2 muestra cómo la aplicación anfitriona debe ocuparse de que se registren los canales.

Tabla 36.4. Espacio de nombres System.Runtime.Remoting.Channels.Http

Clase	Descripción
HttpChannel	Proporciona una implementación de un canal emisor-receptor que utiliza el protocolo HTTP para transmitir mensajes. Esta clase es una combinación de las clases HttpClientChannel HttpServerChannel, lo que permite la comunicación en ambos sentidos a través de HTTP.
HttpClientChannel	Proporciona una implementación de un canal de cliente que utiliza el protocolo HTTP para transmitir mensajes.
HttpRemotingHandler	Implementa un controlador ASP.NET que envía solicitudes al canal HTTP remoto.
HttpRemotingHandlerFactory	Inicializa nuevas instancias de la clase HttpRemotingHandler.
HttpServerChannel	Proporciona una implementación de un canal de servidor que utiliza el protocolo HTTP para transmitir mensajes.

Listado 36.2. Cómo registrar canales

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;

namespace Client
{
    /// <summary>
    /// Descripción resumida de Class1.
    /// </summary>
    class RemotingClient
    {

        [STAThread]
        static void Main(string[] args)
        {
            TcpChannel chan1 = new TcpChannel(8085);
            ChannelServices.RegisterChannel(chan1);

            HttpChannel chan2 = new HttpChannel(8086);
            ChannelServices.RegisterChannel(chan2);
        }
    }
}
```

NOTA: No necesita usar los canales HTTP y TCP en la aplicación anfitriona. Si está permitiendo a los clientes que llamen en los dos tipos de canales, puede registrar los dos tipos de canales; sin embargo, por lo general, usará un formateador, TCP o HTTP, basado en el tipo de los clientes que están accediendo al objeto remoto.

Cómo activar el objeto remoto

Para hospedar el objeto remoto sólo queda registrar el ensamblado con el entorno remoto. En la aplicación anfitriona, antes de poder activar el ensamblado que contiene los métodos, debe agregar una referencia al ensamblado. Haga clic con el botón derecho del ratón en el objeto `References` en el Explorador de soluciones, lo que hace que aparezca el cuadro de diálogo `Agregar referencia`. En la aplicación que estamos escribiendo, deberá buscar el directorio `C:\cSharpRemoting\ServidorObject` y agregar el ensamblado `ServidorObject.dll` a la aplicación. Tras hacerlo, puede agregar el espacio de nombres `ServidorObject` a su archivo de clase usando la instrucción `using`, como muestra el siguiente fragmento:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using ServidorObject;
```

Tras agregar una referencia al ensamblado remoto creado anteriormente, puede agregar el código que registra el objeto mediante el entorno remoto. Hay dos maneras de hacerlo:

- Use el método `RegisterWellKnownServiceType()` de la clase `RemotingConfiguration` para pasar el tipo de objeto que está creando, el URI del objeto y el modo de activación del objeto.
- Use el método `Configure()` de la clase `RemotingConfiguration` para pasar un archivo de configuración con los detalles de activación del objeto.

Cada método de activación funciona igual, pero almacenar los detalles de activación en un archivo de configuración le otorgan más flexibilidad si cualquiera de los detalles de activación cambia, como el número de puerto del canal que está usando. Ya sopesaremos las ventajas y desventajas de ambos tipos de activa-

ción, pero antes examine los métodos y propiedades de la clase `RemotingConfiguration` disponibles, descritos en la tabla 36.5 y en la tabla 36.6, respectivamente. Además de los métodos de activación descritos anteriormente, puede usar muchos métodos y propiedades muy prácticos de esta clase para descubrir información en tiempo de ejecución sobre los objetos que está ejecutando.

Tabla 36.5. Métodos de la clase `RemotingConfiguration`

Método	Descripción
Configure	Lee el archivo de configuración y configura la infraestructura remota.
GetRegisteredActivatedClientTypes	Recupera una matriz de tipos de objetos registrados en el cliente como tipos que se activarán de forma remota.
GetRegisteredActivatedServiceTypes	Recupera una matriz de tipos de objetos registrados en el servicio que se pueden activar cuando lo solicita un cliente.
GetRegisteredWellKnownClientTypes	Recupera una matriz de tipos de objetos registrados en el cliente como tipos conocidos.
GetRegisteredWellKnownServiceTypes	Recupera una matriz de tipos de objetos registrados en el servicio como tipos conocidos.
GetType (heredado de Object)	Obtiene el tipo de la instancia actual.
IsActivationAllowed	Devuelve un valor booleano que indica si el tipo especificado está autorizado para ser cliente activado.
IsRemotelyActivatedClientType	Sobrecargado. Comprueba si el tipo de objeto especificado se registra como tipo de cliente activado de forma remota.
IsWellKnownClientType	Sobrecargado. Comprueba si el tipo de objeto especificado está registrado como tipo de cliente conocido.
RegisterActivatedClientType	Sobrecargado. Registra un tipo de objeto en el cliente como un tipo

Método	Descripción
RegisterActivatedServiceType	que se puede activar en el servidor. Sobrecargado. Registra un tipo de objeto en el servicio como un tipo que se puede activar a petición del cliente.
RegisterWellKnownClientType	Sobrecargado. Registra un tipo de objeto registrado en el cliente como objeto de tipo conocido (llamada única o singleton).
RegisterWellKnownServiceType	Sobrecargado. Registra un tipo de objeto en el servicio como objeto de tipo conocido (llamada única o singleton).

Tabla 36.6. Propiedades de la clase `RemotingConfiguration`

Propiedad	Descripción
<code>ApplicationId</code>	Obtiene el ID de la aplicación que se ejecuta actualmente
<code>ApplicationName</code>	Obtiene o establece el nombre de una aplicación remota
<code>ProcessId</code>	Obtiene el ID del proceso que se ejecuta actualmente

Cómo registrar objetos con `RegisterWellKnownServiceType`

Para registrar un objeto con el método `RegisterWellKnownServiceType()` de la clase `RemotingConfiguration`, sólo tiene que pasar el nombre de la clase, que es `ServerObject.Class1`; el URI del objeto remoto, que es `ReturnName`; y el tipo de modo en el que el objeto será creado, que en este caso es `SingleCall`. Más adelante estudiaremos la enumeración `WellKnownObjectMode`. El listado 36.3 completa la aplicación anfitriona usando el método `RegisterWellKnownServiceType`.

Listado 36.3. Cómo usar `RegisterWellKnownServiceType`

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
```

```

using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using ServidorObject;

namespace Servidor
{
    /// <summary>
    /// Descripción resumida de Class1.
    /// </summary>

    class Class1

    {
        /// <summary>
        /// El principal punto de entrada a la aplicación.
        /// </summary>

        [STAThread]

        static void Main(string[] args)
        {

            TcpChannel chan1 = new TcpChannel(8085);

            ChannelServices.RegisterChannel(chan1);

            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(ServidorObject.Class1), "ReturnName",
                WellKnownObjectMode.SingleCall);

            Console.WriteLine("Press any key to exit");
            Console.ReadLine();

        }
    }
}

```

Como la aplicación anfitriona es una aplicación de consola, agregue la instrucción `Console.ReadLine` al final para que la ventana de la consola permanezca abierta mientras los objetos están usando el objeto remoto. La duración del canal es la cantidad de tiempo que la ventana permanece abierta. Tras cerrar la ventana de consola, el canal se destruye y termina la concesión de ese canal en particular en el entorno remoto.

La enumeración `WellKnownObjectMode` contiene dos miembros que definen cómo se crean los objetos. Si `WellKnownObjectMode` es `SingleCall`, cada petición de un cliente es atendida por una nueva instancia de objeto. Esto puede representarse mediante el siguiente pseudo-código:

```

Cree objeto X
Llame al metodo del objeto X
Devuelva datos al llamador

```

```
Destruya el objeto X
Recogida de elementos no utilizados
```

Si `WellKnownObjectMode` es `Singleton`, cada petición de un cliente es atendida por la misma instancia de objeto. Esto puede representarse mediante el siguiente pseudo-código:

```
Cree objeto X
Llame al metodo del objeto X
Devuelva datos al llamador
Llame al metodo del objeto X
Devuelva datos al llamador
... continua hasta que el canal sea destruido
```

Este bucle continúa hasta que el canal con el que se registra este objeto en el entorno remoto es destruido.

Dependiendo del tipo de aplicación que esté escribiendo, determine el modo de activación que debe usar según los siguientes factores:

- **Coste:** Si crear el objeto remoto consume recursos y tiempo, usar el modo `SingleCall` puede no ser el modo más efectivo de crear su objeto, ya que el objeto es destruido después de que cada cliente lo use.
- **Información de estado:** Si está almacenando información de estado, como propiedades, en el objeto remoto, use objetos `Singleton`, que pueden mantener datos de estado.

Cómo registrar objetos con el método `Configure`

Si necesita un modo más flexible de mantener los datos de configuración que necesita el entorno remoto para registrar el objeto, puede usar el método `Configure()` de la clase `RemotingConfiguration`. La información de configuración almacenada en el archivo es la misma información que puede usar en el método `RegisterWellKnownServiceType()`. Las ventajas de usar un archivo de configuración es que, si cualquiera de las configuraciones del objeto cambia, puede modificar el archivo de configuración sin cambiar el código. El esquema para realizar la configuración aparece en el listado 36.4 y la explicación de cada elemento en la tabla 36.7.

Listado 36.4. Archivo de configuración remota

```
<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime>
      <channels> (Instance)
      <channel> (Instance)
      <serverProviders> (Instance)
      <provider> (Instance)
```

```

        <formatter> (Instance)
    <clientProviders> (Instance)
        <provider> (Instance)
        <formatter> (Instance)
</client>
    <wellknown> (Client Instance)
    <activated> (Client Instance)
</service>
    <wellknown> (Service Instance)
    <activated> (Service Instance)
</soapInterop>
    <interopXmlType>
    <interopXmlElement>
    <preLoad>
    <channels> (Template)
</channel> (Template)
    <serverProviders> (Instance)
        <provider> (Instance)
        <formatter> (Instance)
    <clientProviders> (Instance)
        <provider> (Instance)
        <formatter> (Instance)
</channelSinkProviders>
    <serverProviders> (Template)
        <provider> (Template)
        <formatter> (Template)
    <clientProviders> (Template)
        <provider> (Template)
        <formatter> (Template)
</debug>

```

Aunque hay muchas opciones en el archivo de configuración, sólo necesita usar las necesarias para su aplicación.

Por ejemplo, el fragmento de código del listado 36.5 representa un archivo de configuración para un objeto activado con HTTP que es un objeto de modo SingleCall.

Listado 36.5. Ejemplo de configuración de archivo

```

<configuration>
  <system.runtime.remoting>
    <application>

      <client url="http://localservidor/ServidorObject">
        <wellknown type="ServidorObject.Class1,
          ReturnName"
          url="http://localservidor/ServidorObject/
Class1.soap" />
      </client>

    <channels>

```

```

        <channel ref="http" />
    </channels>
</application>
</system.runtime.remoting>
</configuration>

```

Tras crear el archivo de configuración, la operación de crear el objeto anfitrión es mucho más simple que registrar un objeto con el método `RegisterWellKnownServiceType()` de la clase `RemotingConfiguration`. El código del listado 36.6 muestra cómo registrar el objeto mediante el método `Configure()`.

Listado 36.6. Cómo usar un archivo de configuración remota en la clase anfitriona

```

namespace Servidor
{
    /// <summary>
    /// Descripción resumida de Class1.
    /// </summary>

    class Class1
    {
        /// <summary>
        /// El principal punto de entrada a la aplicación.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {

            RemotingConfiguration.Configure("Servidor.Exe.Config");

            Console.WriteLine("Press any key to exit");
            Console.ReadLine();

        }
    }
}

```

Cómo puede ver, el código se ha visto reducido de quince líneas a una.

NOTA: El nombre del archivo de configuración debe ser el nombre del ejecutable, incluida la extensión `exe`, con la extensión adicional `config` añadida. En el caso de la aplicación anfitriona, el archivo de configuración recibirá el nombre `servidor.exe.config` y estará en el directorio `Bin` donde está el archivo ejecutable de la aplicación anfitriona.

La tabla 36.7 recoge todos los elementos disponibles y sus usos para el esquema de archivos de configuración remota.

Tabla 36.7. Esquema para el archivo de configuración de valores remotos

Elemento	Descripción
<system.runtime.remoting>	Contiene información sobre objetos y canales remotos
<application>	Contiene información sobre los objetos remotos que la aplicación consume y expone
<lifetime>	Contiene información sobre el período de duración de todos los objetos activados en el cliente que atiende esta aplicación
<channels> (Instancia)	Contiene los canales que la aplicación utiliza para comunicar con objetos remotos
<channel> (Instancia)	Configura el canal que la aplicación utiliza para comunicar con objetos remotos
<serverProviders> (Instancia)	Contiene los proveedores de receptores de canal que van a formar parte de la cadena de llamadas de receptores de canal predeterminada del servidor correspondiente a esta plantilla de canal cuando se hace referencia a la plantilla en otro lugar del archivo de configuración
<provider> (Instancia)	Contiene el proveedor de receptores de canal correspondiente a un receptor de canal que se ha de insertar en la cadena de receptores de canal
<formatter> (Instancia)	Contiene el proveedor de receptores de canal para un receptor de formateador que se ha de insertar en la cadena de receptores de canal
<clientProviders> (Instancia)	Contiene los proveedores de receptores de canal que van a formar parte de la cadena de llamadas de receptores de canal predeterminada del cliente correspondiente a esta plantilla de canal cuando se hace referencia a la plantilla en otro lugar del archivo de configuración

Elemento	Descripción
<client>	Contiene los objetos que la aplicación consume
<wellknown> (Instancia de cliente)	Contiene información sobre los objetos (conocidos) activados en el servidor y que la aplicación desea consumir
<activated> (Instancia de cliente)	Contiene los objetos activados en el cliente que consume una aplicación de cliente
<service>	Contiene los objetos que la aplicación expone a otros dominios de aplicación o contextos
<wellknown> (Instancia de servicio)	Contiene información sobre los objetos (conocidos) activados en el servidor y que la aplicación desea publicar
<activated> (Instancia de servicio)	Contiene información sobre los objetos activados en el cliente y que la aplicación expone a los clientes
<soapInterop>	Contiene las asignaciones de tipos utilizadas con SOAP
<interopXmlType>	Crea una asignación bidireccional entre un tipo de Common Language Runtime y un tipo XML y espacio de nombres XML
<interopXmlElement>	Crea una asignación bidireccional entre un tipo de Common Language Runtime y un elemento XML y espacio de nombres XML
<preLoad>	Especifica el tipo para cargar las asignaciones de las clases que extienden <code>SoapAttribute</code>
<channels> (Plantilla)	Contiene las plantillas de canal que la aplicación utiliza para comunicar con objetos remotos
<channel> (Plantilla)	Contiene la plantilla de canal que la aplicación puede especificar y configurar para comunicar o escuchar las solicitudes de objetos remotos
<channelSinkProviders>	Contiene plantillas para proveedores de receptores de canal de cliente y servi-

Elemento	Descripción
	dor. Se puede hacer referencia a todos los proveedores de receptores de canal especificados debajo de este elemento en cualquier lugar donde esté registrado un proveedor de receptores de canal
<serverProviders> (Plantilla)	Contiene plantillas de receptores de canal que se pueden insertar en una cadena de llamadas de canales de servidor
<provider> (Plantilla)	Contiene la plantilla de proveedores de receptores de canal correspondiente a un receptor de canal que se ha de insertar en la cadena de receptores de canal del servidor o cliente
<formatter> (Plantilla)	Contiene el proveedor de receptores de canal para un receptor de formateador que se ha de insertar en la cadena de receptores de canal del cliente o servidor
<clientProviders> (Plantilla)	Contiene plantillas de receptores de canal que se pueden insertar en una cadena de llamadas de canales de cliente
<debug>	Especifica si se van a cargar tipos en el archivo de configuración cuando se inicia la aplicación

Hasta ahora, este capítulo ha explicado los entresijos de la creación de la aplicación anfitriona que registra el objeto remoto con el entorno remoto. Ahora debe escribir la aplicación cliente que realiza las peticiones al objeto remoto, que es el tema de la siguiente sección.

Cómo escribir el cliente remoto

Hasta ahora, ha creado el objeto anfitrión y la aplicación de servidor anfitrión que gestiona las peticiones del objeto anfitrión por parte del cliente para el mediante el entorno remoto. El último paso para escribir esta aplicación remota es escribir la aplicación cliente que realiza las peticiones al objeto remoto. En este caso, el cliente llama al método `ReturnName` del ensamblado `Server-`

Object y pasa un parámetro `customerID` que usa el método `ReturnName()` para buscar el nombre de la compañía del cliente en la base de datos Northwind.

Para empezar, cree una nueva aplicación de consola llamada `Client` en el directorio `C:\cSharpRemoting\Client`. Puede llamar al objeto remoto con cualquier tipo de aplicación, pero para hacerlo más sencillo, crearemos una aplicación de consola.

Se puede llamar al objeto remoto desde el cliente de una de estas tres formas:

- Llamando al método `GetObject()` de la clase `Activator`, con lo que es activado en el servidor.
- Llamando al método `CreateInstance()` de la clase `Activator`, con lo que es activado en el cliente.
- Usando la palabra clave `new`, con lo que puede ser activado en el servidor o en el cliente.

La diferencia entre activación en el cliente y activación en el servidor es *cuan-*do se crea realmente el objeto. Cada tipo de activación puede conseguirse mediante programación o mediante un archivo de configuración (usando el mismo formato descrito en la tabla 36.7), pero para la activación en cliente, se hace un viaje de ida y vuelta al servidor para crear el objeto cuando se invoca al método `CreateInstance()`. Por el contrario, cuando un objeto es activado en el servidor, el objeto servidor no se crea hasta que se hace una llamada al método desde el cliente. Los objetos activados en el servidor crean un proxy que el cliente puede usar para descubrir las propiedades y métodos disponibles en el objeto servidor. La principal desventaja de la activación en el servidor es que sólo se permiten constructores predeterminados, de modo que si necesita pasar varios parámetros a un constructor de método, deberá usar una activación en el lado del cliente mediante el método `CreateInstance()` de la clase `Activator`. Todos los métodos de la clase `Activator` aparecen en la tabla 36.8.

Tabla 36.8. Métodos de la clase `Activator`

Método	Descripción
<code>CreateComInstanceFrom</code>	Crea una instancia del objeto COM cuyo nombre se especifica, utilizando el archivo de ensamblado con nombre y el constructor que mejor coincida con los parámetros especificados
<code>CreateInstance</code>	Sobrecargado. Crea una instancia del tipo especificado utilizando el constructor que mejor coincida con los parámetros especificados
<code>CreateInstanceFrom</code>	Sobrecargado. Crea una instancia del tipo cuyo nombre se especifica, utilizando el archivo de

Método	Descripción
GetObject	ensamblado con nombre y el constructor que mejor coincida con los parámetros especificados Sobrecargado. Crea un proxy para un objeto remoto en ejecución, un objeto conocido activado en el servidor o un servicio Web XML

Tras decidir el tipo de aplicación necesaria para la aplicación, puede escribir el código cliente. El listado 36.7 muestra el código completo de la aplicación cliente.

Como sucedía en el código anfitrión, debe registrar un canal en primer lugar. Cuando registra un canal desde el cliente, no especifica el número de canal. La llamada al final del URI indica al cliente la dirección en la que se encuentra el canal correcto, porque está incluido en la llamada de método `GetObject()`: especifique el objeto que está intentando crear y la localización del objeto. Tras crear la clase, puede llamar a métodos y establecer propiedades del mismo modo que en cualquier otra clase.

Listado 36.7. Aplicación de cliente remoto

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using ServerObject;

namespace Client
{
    /// <summary>
    /// Descripción resumida de Class1.
    /// </summary>
    class RemotingClient
    {

        [STAThread]
        static void Main(string[] args)
        {

            ChannelServices.RegisterChannel(new TcpChannel());

            ServerObject.Class1 x = (Class1)Activator.GetObject(
                typeof(Class1),
                "tcp://localhost:8085/ReturnName", null);

            Console.WriteLine(x.ReturnName("ALFKI"));
            Console.ReadLine();
        }
    }
}
```

Tras escribir el cliente, puede ejecutar la aplicación Servidor.exe y, a continuación, ejecutar la aplicación cliente; y debería obtener unos resultados similares a los de la figura 36.2.

La aplicación anfitriona permanece siempre abierta, o hasta que la cierre, mientras cada llamada de cliente a la aplicación anfitriona devuelve el nombre de la compañía para ALFKI customerID, que es la identificación de comprador pasada a la aplicación cliente.

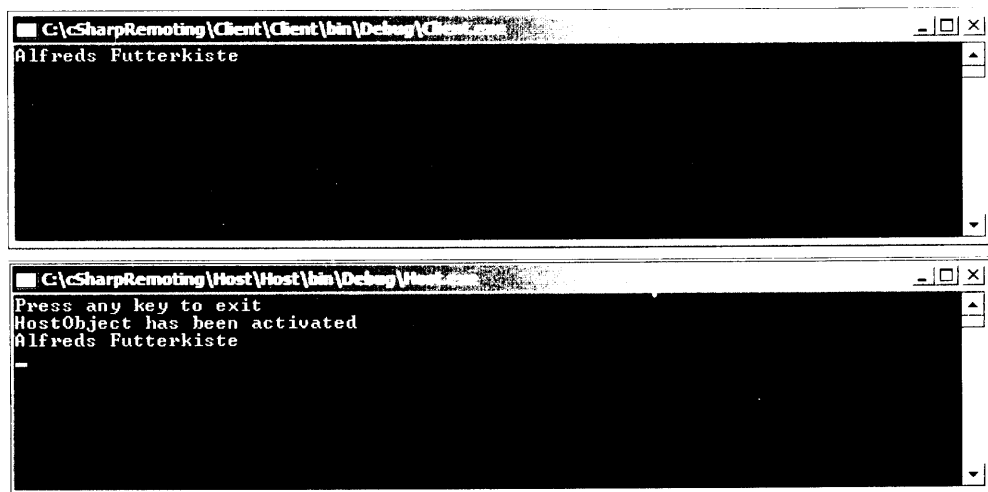


Figura 36.2. Resultados de ejecutar Servidor.exe y Client.exe

Si deja el código de la aplicación anfitriona original en modo `SingleCall`, cada vez que ejecute la aplicación cliente, el objeto servidor se destruirá y se volverá a crear.

Al cambiar `WellKnownObjectMode` a `Singleton`, observará la diferencia entre los modos `SingleCall` y `Singleton`.

El siguiente fragmento de código muestra la aplicación anfitriona que crea el objeto en modo `Singleton`:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(ServidorObject.Class1), "ReturnName",
    WellKnownObjectMode.Singleton);
```

La figura 36.3 muestra la diferencia entre los resultados de la aplicación anfitriona tras ejecutar la aplicación cliente varias veces.

Como puede ver, el modo `Singleton` no destruye el objeto cuando el método sale del contexto, mientras que el modo `SingleCall` necesita volver a crear el objeto cada vez que se llama al método `ReturnName()`.

The image shows two screenshots of a Windows command prompt window. The top window is titled 'C:\SharpRemoting\Host\Host\bin\Debug\Host.exe' and displays the following text: 'Press any key to exit - this is SingleCall', 'HostObject has been activated', 'HostObject has been activated', 'Alfreds Futterkiste', 'HostObject has been activated', 'Alfreds Futterkiste', 'HostObject has been activated', and 'Alfreds Futterkiste'. The bottom window is titled 'C:\SharpRemoting\Host\Host\bin\Debug\Host.exe' and displays the following text: 'Press any key to exit - this is Singleton', 'HostObject has been activated', 'Alfreds Futterkiste', 'Alfreds Futterkiste', and 'Alfreds Futterkiste'.

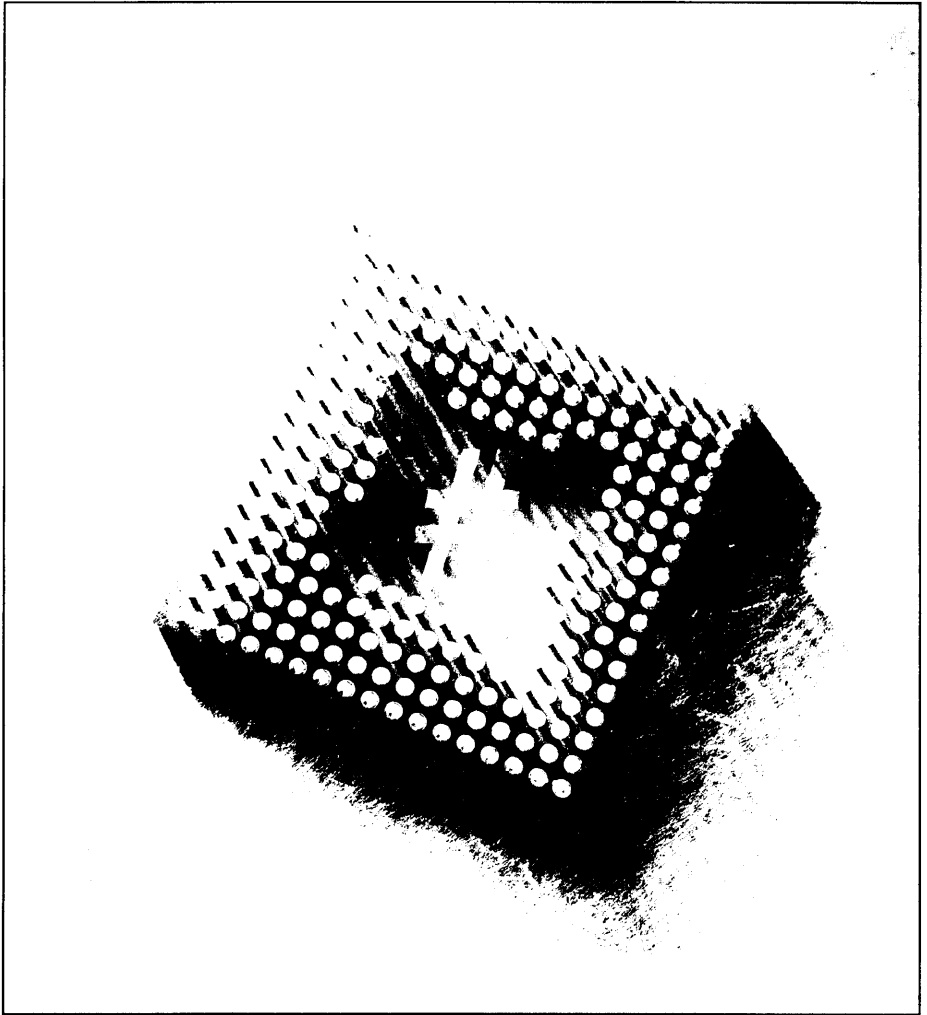
Figura 36.3. Aplicación anfitriona ejecutándose en modo SingleCall y en modo Singleton

Resumen

Este capítulo analiza detalladamente el entorno remoto de .NET. Al usar el entorno remoto, puede activar objetos a través de límites de proceso, dominios de aplicación y límites de equipo. Si va a implementar un entorno remoto, hay algunos temas más avanzados en el SDK que quizás le convenga leer antes de empezar:

- Crear formateadores de usuario: Puede crear formateadores de usuario si los formateadores TCP y HTTP no satisfacen sus necesidades de uso de datos. Busque Receptores y Cadenas de receptores en Framework SDK.
- Acceso remoto asincrónico: El acceso remoto es otra tecnología .NET con capacidades asincrónicas integradas. Busque RPC asincrónico en Framework SDK para aprender a usar delegados y eventos con procedimientos remotos.

Hay muchas buenas razones para estudiar los accesos remotos, pero antes de empezar, asegúrese de estudiar las capacidades de los servicios Web XML y ASP.NET para conseguir comunicación entre procesos. Puede ahorrarse mucho tiempo y esfuerzo creando las aplicaciones anfitrionas y modificando el modo en que sus clientes instancian objetos.



37 C# y seguridad .NET

Una de las cosas más importantes que debe recordar cuando se traslade a C# y .NET Framework es la seguridad. Debe asegurarse de que cuando cree aplicaciones con n-niveles, la seguridad sea de la máxima prioridad porque las probabilidades de que se produzca una brecha en una aplicación distribuida son mucho mayores que en una aplicación independiente. Es por esto que .NET Framework se creó pensando en la seguridad, lo que se refleja en cada aspecto del entorno. .NET Framework es capaz de ejecutarse de manera remota, realizar descargas dinámicas de nuevos componentes e incluso de ejecución dinámica. Con este tipo de entorno, si un programador debe crear el modelo de seguridad, probablemente tarde más en codificarlo que en crear el propio programa.

Cuando cree aplicaciones, el modelo de seguridad suele basarse en el nivel de usuario o el nivel de grupo. La aplicación realizará ciertas acciones o no. .NET Framework proporciona a los programadores medios para *seguridad basada en funciones*, que trabaja de una manera muy parecida a la seguridad de nivel de usuario y de nivel de grupo. La seguridad basada en funciones se puede resumir en principios e identidades, aunque también proporciona seguridad de nivel de código, al que se hace referencia generalmente como *seguridad de acceso a código* o *seguridad basada en pruebas*.

Cuando un usuario inicia una aplicación que usa seguridad de acceso a código, puede tener acceso a un recurso (por ejemplo, una unidad de red), pero si el

código contenido en la aplicación no es fiable, el programa no puede acceder a la unidad de red. Este tipo de seguridad se basa en código móvil. Quizás no quiera usar una aplicación móvil y dejar a esa aplicación que acceda a todos los recursos a los que se ha encomendado. La seguridad basada en funciones evita que los programadores malintencionados escriban aplicaciones que puedan ejecutarse como si las estuviéramos ejecutando nosotros y realizar todo tipo de acciones en nuestro equipo local o a través de nuestra red corporativa.

La seguridad de .NET Framework se coloca sobre la seguridad ya presente en su sistema operativo (OS). Este segundo nivel de seguridad es mucho más extensible que la seguridad OS. Ambos tipos de seguridad, OS y .NET Framework, pueden complementarse entre sí.

Este capítulo le acerca a varios temas relacionados con la seguridad, como el uso de funciones de Windows para determinar permisos. Aprenderá a solicitar y denegar permisos dentro del código mientras realiza operaciones de registro. Por último, aprenderá a usar permisos basados en atributos para definir los derechos de su código en tiempo de ejecución.

Seguridad de código

La seguridad de acceso a código determina si se permite a un ensamblado ejecutarse basándose en varias unidades de prueba, como la URL de la que procede el ensamblado y quién autoriza el control. Al instalar .NET Framework, están configurados los permisos predeterminados, lo que reduce enormemente las posibilidades de que un control que no es de confianza procedente de Internet o de una intranet local pueda ejecutarse en su equipo. Puede haber visto esto si ha intentado ejecutar algunas aplicaciones o usar algunos controles desde una unidad de red que exija privilegios de seguridad especiales. Estos privilegios de seguridad especiales incluyen la escritura en un archivo de disco, leer o escribir en y desde el registro, además de operaciones de red. Normalmente recibirá una excepción de seguridad como las siguientes cuando intente hacer estas acciones si no cambia la directiva de seguridad para que permita este tipo de comportamiento:

```
Unhandled Exception: System.Security.SecurityException: Request
for the permission of type
System.Security.Permissions.FileIOPermission
...
The state of the failed permission was:
<IPermission
class="System.Security.Permissions.FileIOPermission, mscorlib,
Vers
ion=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
  version="1"
  Read="Z:\test.dat"
  Write="Z:\test.dat"/>
```

La seguridad de acceso a código solamente funciona en código verificable. Durante la compilación justo a tiempo (JIT), se examina el lenguaje intermedio de Microsoft (MSIL) para garantizar la seguridad de tipo. El código de seguridad de tipo sólo tiene acceso a las posiciones de memoria para las que tiene derechos. Acciones como las operaciones de puntero están prohibidas, de modo que solo se puede entrar y salir de las funciones desde los puntos de entrada y salida predefinidos. Esto no es un método infalible: pueden producirse errores. Sin embargo, impide que una unidad de código malintencionado pueda forzar un error en su aplicación y aprovechar algún error en el sistema operativo, consiguiendo así acceder a la pila. En estas circunstancias, cuando una unidad de código malicioso ha forzado un error, el código que generó el error solo puede acceder a las posiciones de memoria que el JIT determinó que eran accesibles para él.

Directiva de seguridad de código

La seguridad de acceso a código permite a una plataforma asignar un nivel de seguridad a una aplicación o ensamblado. Como esto se consigue con pruebas tomadas del elemento en cuestión, la seguridad de acceso a código también recibe el nombre de seguridad basada en pruebas. Las pruebas recogidas del código podrían ser la ubicación en Internet desde la que se descargó el código, una firma digital ubicada en el código o código escrito por el propio autor.

Las directivas de seguridad de código definen varios grupos de código, cada uno de los cuales tiene un conjunto de permisos. Cuando una aplicación se ha ejecutado, es analizada en busca de pruebas. Según la prueba del código, éste se coloca en un grupo de código, heredando así los permisos de ese grupo. Estas directivas de seguridad pueden establecerse en el nivel de dominio de empresa, equipo, usuario o aplicación, proporcionando así un alto grado de control sobre lo que se ejecuta y con qué acceso. Puede haber permitido a su código que tenga derechos ilimitados, pero su administrador de red puede definir algunas directivas de seguridad que superen a las suyas.

Permisos de código

El CLR, cuando concede permisos de seguridad, sólo concede permisos al código en las operaciones que se le permite realizar. El CLR usa objetos llamados *permisos* para implementar este tipo de seguridad en código gestionado. Los principales usos de los permisos son los siguientes:

- El código puede solicitar los permisos que pretende usar o que posiblemente necesite. .NET Framework tiene la tarea de determinar si estas peticiones son válidas. Las peticiones de seguridad se conceden sólo si las pruebas recogidas del código lo permiten. El código nunca recibe más permisos de los permitidos por la seguridad actual. Por otra parte, el código puede recibir menos permiso que el especificado en la petición.

- El CLR concede permiso al código basándose en varios factores: La identidad del código (como la URL de la que fue obtenido, quién escribió el código y similares), los permisos que se solicitan y la cantidad de código que es de confianza, definida por las diferentes directivas de seguridad.
- El código puede hacer una petición para obtener un determinado permiso. Si se realiza una petición mediante el código, todo el código que se ejecute en el contexto de la aplicación debe tener acceso al permiso para que éste sea concedido.

El código puede recibir tres clases de permisos, cada uno de los cuales tiene un propósito específico:

- Los permisos de código de acceso representan el acceso a un recurso protegido o la autoridad para realizar una operación protegida.
- Los permisos de identidad indican que el código tiene credenciales que admiten un tipo de identidad particular, como código que puede tener una identidad "Administrador" y, por tanto, ejecutarse con todos los permisos que pueda tener un administrador.
- Los permisos de seguridad basada en funciones proporcionan un mecanismo para descubrir si un usuario (o el agente que actúa en nombre del usuario) tiene una identidad particular o es un miembro de un cargo específico. `PrincipalPermission` es el único permiso de seguridad basada en funciones.

El tiempo de ejecución proporciona clases de permiso integradas en varios nombres de espacios y proporciona compatibilidad para diseñar e implementar clases de permiso personalizadas.

Seguridad de usuario

Casi todos los sistemas de seguridad actuales implementan algo llamado *seguridad de usuario*. Estos tipos de sistemas de seguridad requieren información de los usuarios que solicitan acceso. Por ejemplo, deben saber quién es esa persona y a qué elementos tiene acceso ese usuario. La seguridad de usuario desempeña un papel muy importante en los sistemas computerizados porque, cuando ejecuta una aplicación en su equipo, la aplicación suele guardar la identidad de la persona que la está ejecutando. Por tanto, si ejecuta una aplicación, esa aplicación tiene todos los derechos y permisos en su equipo local y a través de la red que tendríamos nosotros.

A diferencia de los servicios Windows, que le permiten configurar quién parece estar ejecutando la aplicación, una aplicación Windows normal nunca había otorgado este tipo de control con anterioridad. Este hecho ha facilitado la prolife-

ración de muchos virus y troyanos a los que tienen que enfrentarse diariamente los usuarios y empresarios de ordenadores. Al permitirle determinar el tipo de permiso que tienen las aplicaciones en su equipo, se reduce enormemente la posibilidad de un ataque por parte de un código maligno. Operaciones, como leer el registro, sobrescribir archivos de sistema o recorrer su libreta de direcciones personal, no serán posibles. Puede probar rápidamente si sus aplicaciones se ejecutan según el usuario que las ejecuta probando el programa del listado 37.1.

Listado 37.1. Variables de entorno para tareas de seguridad sencillas

```
using System;
namespace SimpleSecurity
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("I am currently running as:");
            Console.WriteLine("User : {0}", Environment.UserName);
            Console.WriteLine("Domain :
{0}", Environment.UserDomainName);
        }
    }
}
```

Cuando ejecuta este programa, debería ver el nombre que usa para conectarse a Windows, además del nombre de su dominio, como muestra la figura 37.1.

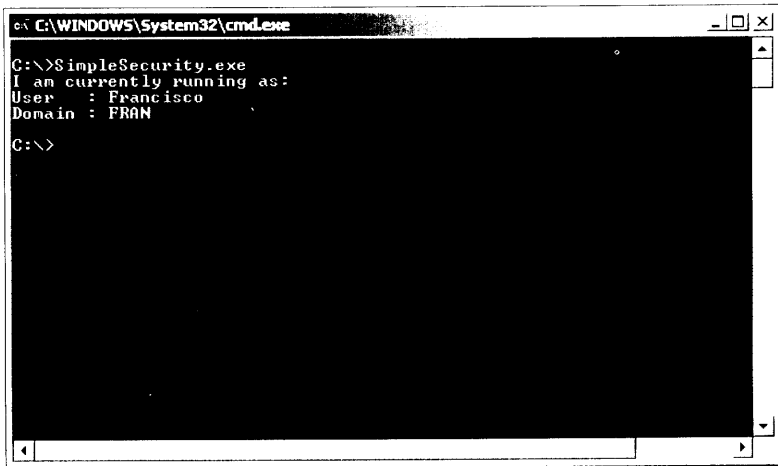


Figura 37.1. La clase Environment puede ser utilizada para tareas de seguridad sencillas

Si no está conectado a un dominio de red, simplemente verá el nombre de su sistema como el nombre de dominio. El tipo de seguridad más sencillo que proba-

blemente pueda implementar en este momento sería hacer que se compare el nombre de usuario y el nombre de dominio para validar una operación y, si todo es correcto, continuar con el programa. Esto es válido hasta que lleva su aplicación a otro equipo y deja de funcionar porque ha incluido nombres seguros en su código. La siguiente sección revisa este tipo de seguridad junto con otros tipos sencillos.

Seguridad .NET y basada en funciones

La seguridad basada en funciones se basa en la clase `PrincipalPermission`. Puede usar `PrincipalPermission` para determinar si el usuario actual tiene un nombre concreto (como John Doe) o si el usuario pertenece a un grupo particular. Esta clase es el único permiso de seguridad basado en funciones proporcionado por la biblioteca de clases .NET Framework.

Tras definir los objetos `Identity` y `Principal`, puede realizar comprobaciones de seguridad con ellas de uno de estos modos:

- Usando comprobaciones de seguridad imperativa
- Usando comprobaciones de seguridad declarativa
- Accediendo directamente al objeto `Principal`

Al utilizar código administrado, puede emplear comprobaciones de seguridad imperativa o declarativa para determinar si un objeto principal concreto es miembro de una función conocida, tiene una identidad conocida o representa una identidad conocida que actúa en una función. Para realizar la comprobación de seguridad utilizando seguridad imperativa o declarativa, se debe efectuar una solicitud de seguridad para un objeto `PrincipalPermission`. Durante la comprobación de seguridad, el entorno de ejecución común examina el objeto principal de quien efectúa la llamada para determinar si su identidad y su función coinciden con las representadas por el objeto `PrincipalPermission` que se demanda. Si el objeto principal no coincide, se inicia una excepción `SecurityException`. Cuando esto sucede, sólo se comprueba el objeto principal del subproceso actual. La clase `PrincipalPermission` no produce un recorrido de pila con permiso de acceso a código, ya que podría causar graves problemas de seguridad. Además, se puede acceder directamente a los valores del objeto principal y realizar comprobaciones sin un objeto `PrincipalPermission`. En este caso, basta con leer los valores del subproceso principal actual o utilizar la autorización de ejecución del método `IsInRole`.

Cómo asignar las funciones Windows

Por regla general, cuando se necesita asignar varios usuarios a funciones específicas, es mejor usar la funcionalidad de grupo integrada en Windows NT 4.0.

Windows 2000 y Windows XP. En lugar de añadir privilegios para cada usuario, puede crear un nuevo grupo con ciertos derechos de acceso y luego agregar los usuarios al grupo en concreto. Estas funciones ahorran una cantidad considerable de tiempo y permiten a los administradores del servidor controlar una gran cantidad de usuarios. Empecemos agregando un nuevo grupo en Windows 2000/Windows XP:

1. Haga clic con el botón derecho del ratón en **Mi PC** y seleccione **Administrar**. Cuando se abra la consola de **Administración de equipos**, expanda la vista del árbol en el cuadro izquierdo haciendo clic en **Usuarios locales y grupos** para que aparezca **Grupos** y haga clic en **Grupos**.
2. Cuando haga clic en **Grupos**, verá una lista de aproximadamente siete grupos integrados en el sistema operativo Windows, como muestra la figura 37.2.

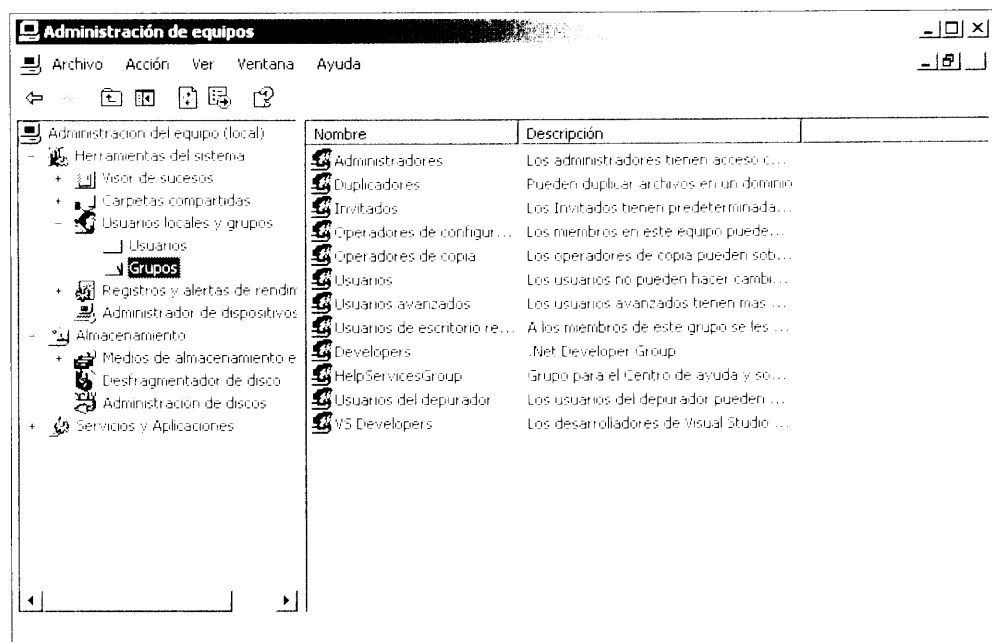


Figura 37.2. La administración de grupos se consigue con la consola Administración de equipos

3. Haga clic con el botón derecho del ratón en el cuadro a la derecha y seleccione **Grupo nuevo**. Llame a este grupo **Developers**, como muestra la figura 37.3.
4. Tras crear este grupo, haga clic en el botón **Agregar** y agregue su cuenta de usuario al grupo. Para este ejemplo, asegúrese de que no está en el grupo **Administradores**. Si es un administrador, quizás quiera probar la siguiente aplicación con otra cuenta Windows.

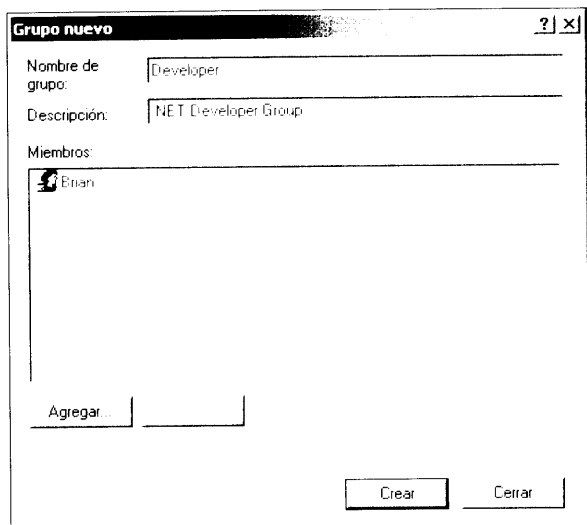


Figura 37.3. Agregue un grupo Developers a Windows

Una vez que ha colocado el nuevo grupo, vamos a estudiar las clases `WindowsPrincipal` y `WindowsIdentity`. Cuando se usan en conjunto, estas dos clases pueden determinar si el usuario actual de Windows pertenece a algún grupo específico. Examine la aplicación de ejemplo del listado 37.2.

Listado 37.2. `WindowsPrincipal` le permite comprobar la pertenencia a una función

```
using System;
using System.Security.Principal;

class Class1
{
    static void Main()
    {
        WindowsIdentity wi = WindowsIdentity.GetCurrent();
        WindowsPrincipal wp = new WindowsPrincipal(wi);

        // Esto comprueba los derechos del administrador local
        // si se encuentra en un dominio
        if (wp.IsInRole(WindowsBuiltInRole.Administrator))
            Console.WriteLine("You are an Administrator!");
        else
            Console.WriteLine("You are not an Administrator.");

        if (wp.IsInRole("POWERHOUSE\\Developer"))
            Console.WriteLine("You are in the Developer group!");
        else
            Console.WriteLine("You are not in the Developer
group.");
    }
}
```

Este código crea un nuevo objeto `WindowsIdentity` (basándose en la identidad del actual usuario) con el método `GetCurrent`.

El objeto `WindowsPrincipal` usa este objeto de identidad como parámetro en su constructor, de modo que puede recuperar cierta información sobre la persona u objeto. A continuación llama al método `IsInRole` de la clase `WindowsPrincipal` para determinar si el usuario pertenece al grupo `Administradores`. El método `IsInRole` tiene tres variaciones sobrecargadas de las cuales puede usar dos.

La primera recibe una enumeración `WindowsBuiltInRole`. Cuando compruebe la pertenencia a alguno de los grupos integrados en Windows, debe usar esta enumeración. Dependiendo de si es un administrador, verá uno de dos mensajes.

A continuación, el código comprueba si el usuario actual pertenece al nuevo grupo `Developer`, usando la segunda versión del método `IsInRole`. Esta versión simplemente recibe un parámetro de cadena que especifica el equipo o el nombre de dominio seguido por el nombre de grupo.

En el código anterior, sustituya la palabra `POWERHOUSE` por el nombre de su dominio o equipo. Si no pertenece al grupo `Administrador` y `Developer` puede observar que esta aplicación de ejemplo sólo le reconoce en el grupo `Administradores`, como muestra la figura 37.4.



```
C:\WINDOWS\System32\cmd.exe
C:\>IsAdmin.exe
Your are an Administrator!
You are not in the Developer group.
C:\>_
```

Figura 37.4. Pertenencia al grupo Administradores puede confundir a `IsInRole`

Esta confusión se produce porque, si es un administrador, forma parte inherente de todos los grupos y tiene acceso a todo. Por tanto, cuando compruebe la pertenencia a funciones en sus aplicaciones, es aconsejable comprobar la pertenencia al grupo específico y a todos los otros grupos que estén por encima del grupo que está comprobando. (Por ejemplo, Administradores, Usuarios preferentes y similares).

Principales

Cada subproceso de una aplicación .NET está asociada con un principal del CLR. El principal contiene una identidad que representa la identidad del usuario que está ejecutando ese subproceso. Si usa una propiedad estática llamada `Thread.CurrentPrincipal`, puede devolver el principal actual asociado al subproceso.

Los objetos principales implementan la interfaz `IPrincipal` que sólo contiene un método y una propiedad. La propiedad `Identity` devuelve el objeto actual de identidad y el método `IsInRole` se usa para determinar si un usuario pertenece a una función o grupo de seguridad determinado. Actualmente, .NET Framework contiene dos clases principales: `WindowsPrincipal` y `GenericPrincipal`. La clase `GenericPrincipal` se emplea cuando hace falta implementar un principal propio. La clase `WindowsPrincipal` representa un usuario de Windows y sus funciones o grupos asociados.

Un objeto `Identity` implementa la interfaz `IIdentity`, que sólo tiene tres propiedades:

- `Name` es la cadena asociada a la identidad actual. El sistema operativo del proveedor de autenticación pasa la cadena al entorno de ejecución común. Un ejemplo de proveedor de autenticación es NTLM (Windows NT Challenge/Response), que autentica conexiones de Windows NT.
- `IsAuthenticated` es un valor booleano que indica si el usuario ha sido autenticado.
- `AuthenticationType` es una cadena que indica qué tipo de autenticación se ha usado. Algunos tipos posibles de autenticación son autenticación básica, Forms, Kerberos, NTLM y autenticación de pasaporte.

Permisos de acceso a código

Antes de ejecutar cualquier aplicación .NET, debe pasar una serie de pruebas de seguridad que dan a la aplicación permiso para realizar ciertas operaciones. Los permisos concedidos al código también pueden ser solicitados en el código o denegados por código.

Todos estos permisos están determinados por una directiva de seguridad en la que .NET Framework confía. Estas directivas de seguridad contienen permisos para acceder a recursos, como recoge la tabla 37.1.

Cuando ejecuta una aplicación, los derechos de cualquiera de los permisos anteriormente mencionados se basan únicamente en si el código tiene derecho al permiso. Es independiente del usuario que está ejecutando el código real. Por tanto, estos permisos reciben el nombre de seguridad de acceso a código.

Tabla 37.1. Permisos de acceso a código más comunes

Recurso	Permiso	Descripción
DNS	DNSPermission	Acceso al sistema de nombres de dominio.
Variables de entorno	EnvironmentPermission	Acceso a las variables de entorno del sistema.
Registro de eventos	EventLogPermission	Acceso a los registros de eventos, incluyendo los registros de eventos existentes y la creación de nuevos registros de eventos.
Operaciones de archivo	FileIOPermission	Acceso a realizar operaciones como leer un archivo y escribir en un archivo.
Registro	RegistryPermission	Acceso al registro de Windows.
Interfaz de usuario	UIPermission	Acceso a la funcionalidad de la interfaz.
Web	WebPermission	Acceso a realizar o aceptar conexiones en una dirección Web.

Cómo crear una sencilla solicitud de código de permiso

En esta sección comprenderá lo sencillo que es solicitar permisos mediante código para realizar una acción específica. En este ejemplo, intentaremos leer una clave del registro, que indica a nombre de quién está registrado el sistema operativo en uso.

Al usar la clase `RegistryPermission`, debe especificar el tipo de acceso solicitado al registro (leer, escribir y similares) y la clave específica a la que se quiere acceder. Por lo general, si sólo necesita un acceso de lectura a una clave concreta del registro, sólo debe solicitar permiso de lectura. De este modo se asegura de que no va a sobrescribir información del registro accidentalmente y de que códigos posteriores, posiblemente malintencionados, puedan cambiar la información. Además, siempre debe envolver sus solicitudes de permiso con algún tipo de controlador de errores. Si el entorno de ejecución común rechaza la solici-

tud de permiso, se inicia una `SecurityException`. Si efectúa esta solicitud en un bloque `try/catch`, no obtendrá ninguna advertencia porque el error es controlado. Aunque puede saber que posee este tipo de permiso en su equipo, no puede predecir las directivas de seguridad que pueden bloquear este acceso en otros equipos o redes.

Tras crear una solicitud de permiso, sólo tiene que llamar al método `Demand()` de la clase `RegistryPermission`. Si `Demand()` se ejecuta sin producir ninguna excepción, se ha aceptado su solicitud de permiso. El listado 37.3 contiene el ejemplo de aplicación.

Listado 37.3. Solicitud de permiso con un controlador de errores estructurado

```
using System;
using Microsoft.Win32;
using System.Security.Permissions;

class Class1
{
    static void Main(string[] args)
    {
        try
        {
            RegistryPermission regPermission = new
            RegistryPermission(RegistryPermissionAccess.AllAccess,
            "HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows
            NT\\CurrentVersion");
            regPermission.Demand();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            return;
        }

        RegistryKey myRegKey=Registry.LocalMachine;
        myRegKey=myRegKey.OpenSubKey
        ("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
        try
        {
            Object oValue=myRegKey.GetValue("RegisteredOwner");
            Console.WriteLine("OS Registered Owner:
            {0}",oValue.ToString());
        }
        catch (NullReferenceException)
        {
        }
    }
}
```

No olvide que aunque la directiva de seguridad .NET permita a este código ejecutarse, la directiva de seguridad del sistema operativo subyacente también

debe concederle permiso para ejecutarse. Tras solicitar los permisos para la clave del registro adecuada, sólo tiene que leer la clave `RegistryOwner` y mostrar la información en la ventana de consola.

Denegación de permisos

Al igual que en el método `Demand`, también puede llamar al método `Deny()`, que elimina los permisos para una operación. Por lo general, es aconsejable eliminar antes de hacer la llamada cualquier permiso que sepa que no va a necesitar. Puede solicitar permisos a medida que el código los vaya necesitando. Use el método `Deny()` cuando haya completado una operación y sepa que ya no van a ser necesarias más operaciones.

La denegación de permisos tiene varias funciones. Por ejemplo, si está usando biblioteca de terceros, querrá asegurarse de que, tras manipular el registro, ningún otro código pueda hacerlo. La denegación de permisos es un modo de conseguirlo.

El código del listado 37.4 usa una versión modificada del ejemplo anterior para denegar en primer lugar un permiso de registro. Tras negar el permiso, intenta leer la clave de registro, lo que da como resultado una `SecurityException`. Si quiere deshacer una operación `Deny` en el código, sólo tiene que usar el método `RevertDeny()` para eliminar la denegación de permiso; y cualquier intento posterior de leer la clave del registro solicitada se llevará a cabo con éxito.

Listado 37.4. Denegación de permisos a los que no desea que se acceda

```
using System;
using Microsoft.Win32;
using System.Security.Permissions;

class Class1
{
    static void Main(string[] args)
    {
        try
        {
            RegistryPermission regPermission = new
RegistryPermission(RegistryPermissionAccess.AllAccess,
"HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion");
            regPermission.Deny();
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
            return;
        }
    }
}
```

```

RegistryKey myRegKey=Registry.LocalMachine;
myRegKey=myRegKey.OpenSubKey
("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
try
{
    Object oValue=myRegKey.GetValue("RegisteredOwner");
    Console.WriteLine("OS Registered Owner:
{0}", oValue.ToString());
}
catch (NullReferenceException)
{
}
}
}

```

Si está ejecutando este ejemplo en Visual Studio, la aplicación deberá detenerse en las líneas de manipulación del registro. Al ejecutar esta aplicación desde la consola se genera una larga lista de errores de excepción que indican cuál es el problema.

Cómo usar permisos basados en atributos

Las solicitudes de permisos de atributo son un modo de asegurarse de que tiene suficientes permisos para varios recursos antes de ejecutar la aplicación realmente. JIT y CLR analizan los atributos cuando se compila la aplicación.

En el listado 37.5, usa `RegistryPermissionAttribute` y `Demand` en `SecurityAction`. Si se concede este permiso en el tiempo de compilación, la aplicación no se ejecuta. No se trata siempre del mejor modo de codificar una aplicación: por lo general tendrá modos más eficaces de controlar errores de este tipo.

Por ejemplo, al crear un programa de chat en red, no es aconsejable evitar que el programa se ejecute cuando no tiene derechos de E/S de archivos, ya que siempre puede solicitar al usuario los parámetros de las operaciones. No obstante, sería lógico no permitir que la aplicación se ejecute si no tiene acceso a operaciones de red. Este tipo de petición de seguridad es crucial para la operación de dicha aplicación.

Listado 37.5. Cómo usar permisos de atributo

```

using System;
using Microsoft.Win32;
using System.Security.Permissions;

[RegistryPermissionAttribute(SecurityAction.Demand)]
class Class1
{
    static void Main(string[] args)
    {

```

```

RegistryKey myRegKey=Registry.LocalMachine;
myRegKey=myRegKey.OpenSubKey
("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion");
try
{
    Object oValue=myRegKey.GetValue("RegisteredOwner");
    Console.WriteLine("OS Registered Owner:
{0}",oValue.ToString());
}
catch (NullReferenceException)
{
}
}
}

```

Directiva de seguridad

Las directivas de seguridad son el corazón de la seguridad basada en pruebas. Después de que se obtiene una prueba de un ensamblado, ese código es asignado a un grupo de código. Este grupo de código, a su vez, tiene un conjunto de permisos que definen lo que el código puede y no puede hacer. No sólo puede modificar la directiva de seguridad para que se ajuste a sus necesidades, puede modificarla a varios niveles y puede crear grupos de código personalizados que complementen las directivas de seguridad que ha definido.

Niveles de directiva de seguridad

Hay cuatro niveles de directivas de seguridad: empresa, equipo, dominio de aplicación y usuario. Todos estos niveles tienen que concordar con un permiso de seguridad o el permiso será denegado, como muestra la figura 37.5.

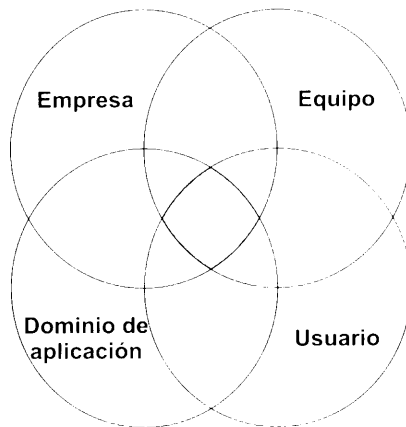


Figura 37.5. Los niveles de se solapan para determinar un nivel de seguridad final

Si cambia las directivas de su equipo para permitir ciertos tipos de operaciones de, por ejemplo, el código descargado de Internet, su administrador de red puede aplicar una directiva de seguridad de empresa para prohibir esas operaciones.

Grupos de código

Todos los niveles de directivas de seguridad contienen grupos de código que, a su vez contienen zonas para cada grupo de código. Este resultado es un ajuste de configuración de seguridad muy detallado a lo largo de todos los niveles de directivas y permite que haya diferentes tipos de seguridad en cada nivel de directiva, dependiendo de la zona del código en cuestión.

Inmediatamente dentro del grupo de código se sitúa un nodo `All_Code`. Como el propio nombre indica, estos conjuntos de permisos se aplican a todo el código. Además de este nodo `All_Code`, puede agregar más nodos para satisfacer sus necesidades. Por ejemplo, puede crear nodos para el código que recibe de los consultores o de cualquier otro tipo de fuente.

Cuando evalúe niveles de seguridad, no olvide el modo en el que la directiva de código se evalúa realmente. Los permisos para un ensamblado se unen a cada nivel de directiva de seguridad. Al unir todos estos permisos, debe trabajar con un enorme conjunto de permisos. Cada uno de estos conjuntos de permisos se solapan para que se pueda realizar una comparación y el valor más restrictivo para cada permiso se usa para el conjunto de permisos final.

Conjuntos de permisos con nombre

Un conjunto de permisos con nombre es un conjunto de permisos al que los administradores o los programadores pueden asociar un grupo de código. Un conjunto de permisos con nombre consiste en, al menos, un permiso y un nombre y una descripción para ese conjunto de permisos en particular. Los administradores pueden usar conjuntos de permisos con nombre para establecer o modificar la directiva de seguridad para grupos de código, de forma parecida a cómo se usan los grupos de Windows NT para gestionar los grupos de usuarios. Puede asociar más de un grupo con el mismo conjunto de permisos con nombre.

La tabla 37.2 describe el conjunto de permisos con nombre integrado proporcionado por el entorno común de ejecución.

Tabla 37.2. Conjuntos de permisos con nombre integrados

Conjunto de permisos	Descripción
Nothing	Sin permisos (no se puede ejecutar código)
Execute	Permiso para ejecutarse, pero no para utilizar recursos protegidos

Conjunto de permisos	Descripción
Internet	Conjunto de permisos de directiva predeterminado, adecuado para contenido de origen desconocido
LocalIntranet	Conjunto de permisos de directiva predeterminado establecido en una empresa
Everything	Todos los permisos estándar (integrados) excepto el permiso de omitir la comprobación
FullTrust	Acceso completo a todos los recursos

Cómo alterar directivas de seguridad

Antes de experimentar realmente con técnicas de codificado para solicitar y denegar permisos, debería familiarizarse con las herramientas disponibles para modificar las configuraciones de seguridad. Las configuraciones de seguridad estudiadas hasta ahora se guardan en archivos XML. La directiva de seguridad del equipo se guarda en el archivo `security.config` ubicado en el directorio `\WINNT\Microsoft.NET\Framework\vx.y.z.mxxx\CONFIG`. Las configuraciones de seguridad del usuario se encuentran en `security.config`, ubicado en el directorio `\Documents and Settings\<Nombre de usuario>\Application Data\Microsoft\CLR Security Config\vx.y.z.mxxx`.

Puede dirigirse al Panel de control, seleccionar **Herramientas administrativas** y a continuación, seleccionar **Configuración de Microsoft .NET Framework** para modificar todas sus necesidades de configuración. Esta herramienta, no sólo tiene varios asistentes integrados que facilitan el proceso de configuración, sino que resulta mucho más sencilla de usar que un editor de XML.

Tras abrir la herramienta de configuración, expanda el nodo **Directiva de seguridad en tiempo de ejecución**, como se muestra en la figura 37.6.

Aquí puede ver realmente los diferentes niveles de seguridad, los grupos de código para cada nivel, los conjuntos de permisos y los ensamblados de directiva.

La operación de agregar nuevos grupos de códigos es muy sencilla. Haga clic con el botón derecho del ratón en el cuadro izquierdo y seleccione **Nuevo**. Se abrirá un asistente que solicitará el nombre de este nuevo grupo de código y preguntará si debe ser creado como un grupo ya existente o si tiene permisos personalizados (véase figura 37.7).

Este asistente le guía a través de todos los permisos disponibles e incluso le ofrece la opción de empaquetar la directiva de seguridad para distribuirla en su empresa.

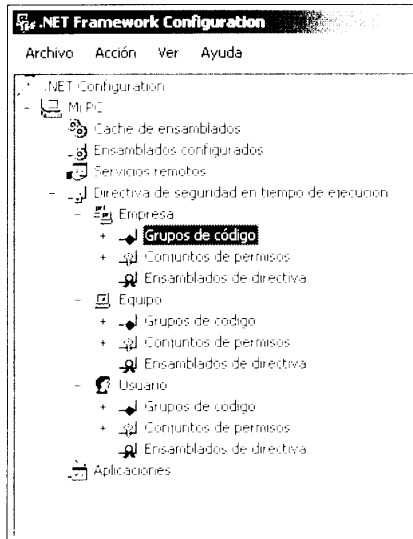


Figura 37.6. Herramienta de configuración de Microsoft .NET Framework

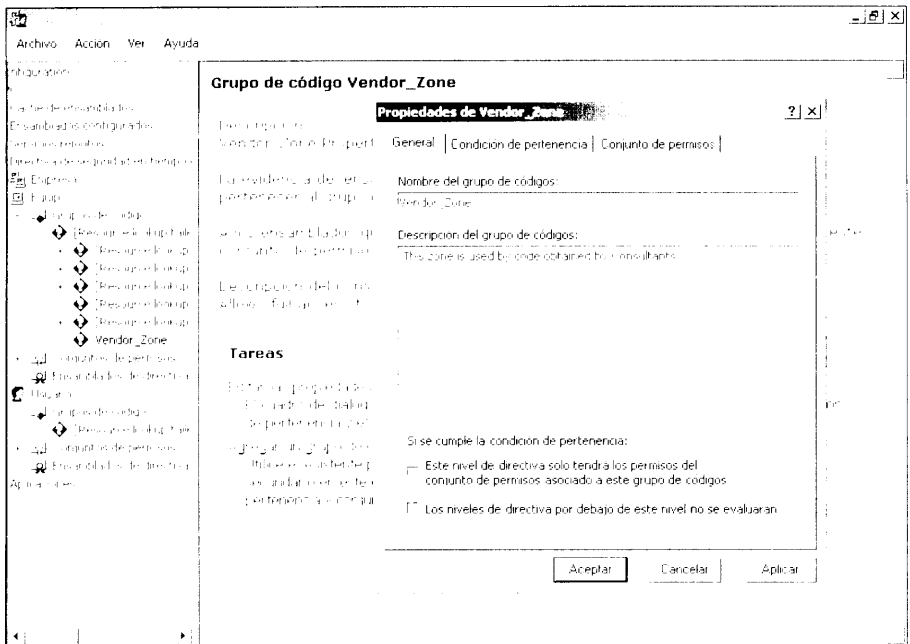


Figura 37.7. Un asistente le ayuda a crear directivas de seguridad personalizadas

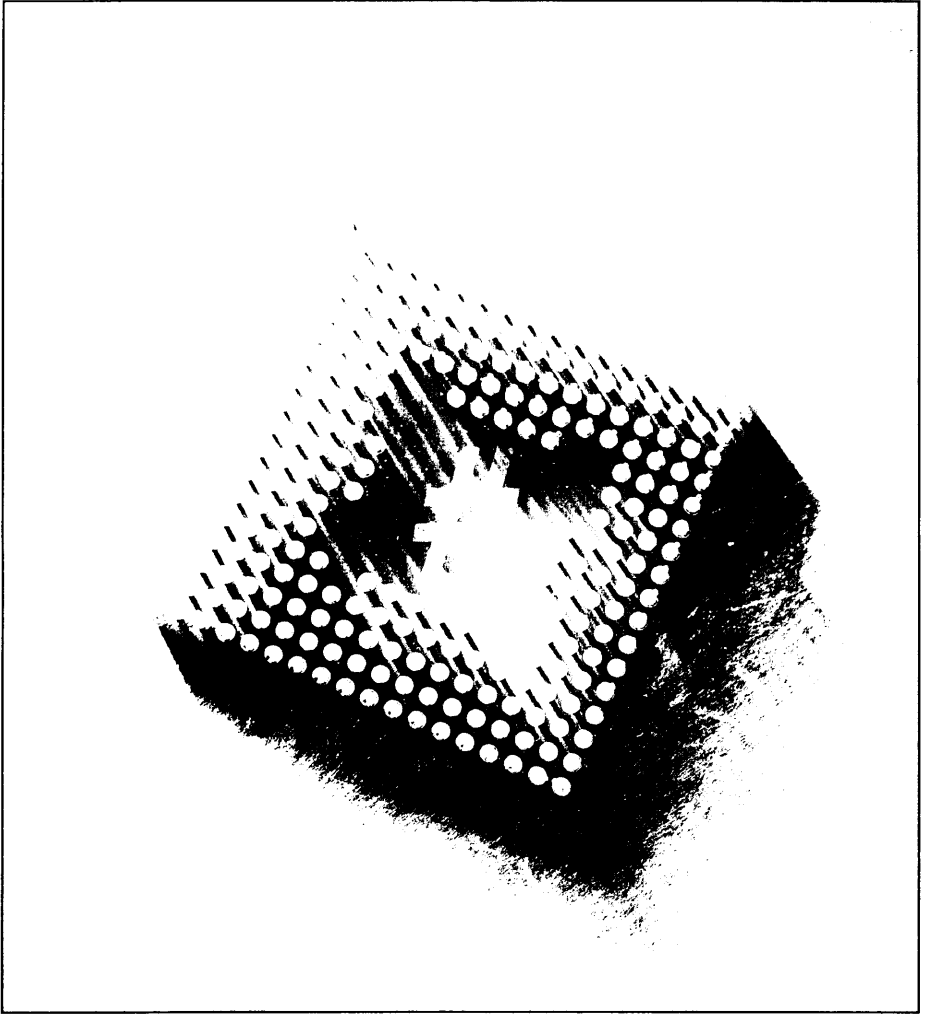
Resumen

.NET Framework se basa en una inmensa cantidad de código de seguridad que vigila cada aspecto de una aplicación o usuario. Este entorno de seguridad permi-

te al programador y al administrador de empresa controlar lo que permite realizar a una aplicación. Hemos estudiado las seguridades de identidad de usuario y de acceso a código. Si se usan conjuntamente con la seguridad del sistema operativo subyacente, puede crear aplicaciones más seguras.

Parte VI

Apéndices



A Manual de XML

A menos que haya vivido en una cueva durante los últimos años, ya habrá oído hablar de XML. Sin lugar a dudas, XML ha recibido muy buenas críticas, más de las que merece. Sin embargo, a pesar de lo que pueda haber oído en algún lustroso folleto de marketing, no es probable que XML solucione el hambre del mundo, traiga la paz mundial o cure todas las enfermedades. Tras leer esta sección, dominará las bases de XML y sus estándares asociados, como esquemas y espacios de nombre. En pocas palabras, XML es un dialecto SGML simplificado diseñado para la interoperabilidad y está considerado el ASCII del futuro. En la última década, ASCII ha sido el estándar tradicional para el intercambio de datos de texto, pero está siendo desplazado rápidamente por XML como el nuevo estándar. A lo largo de esta sección, aprenderá a apreciar la rara elegancia de XML: su combinación única de pura sencillez y potencia en bruto. También aprenderá que otros estándares complementan a XML. La familia XML de estándares complementarios ha crecido mucho en los últimos años, de modo que, para abreviar, sólo le mostraremos los estándares más relevantes.

Objetivos de diseño de XML

XML es un lenguaje de marcado que es extensible. Por supuesto, esto no es un gran descubrimiento dado que sus siglas significan (Extensible Markup Language).

lenguaje de marcado extensible). pero merece la pena señalar este hecho evidente porque capta la esencia de XML. *Extensible* significa que puede agregar nuevas palabras al lenguaje para que se adecuen a sus propósitos específicos. Un lenguaje marcado incluye símbolos especiales en un documento para cumplir alguna función específica.

Esta función varía de un lenguaje de marcado a otro. Uno de los puntos fuertes de XML es que sus funciones son muy amplias: sirve como lenguaje universal de texto para los datos estructurados.

El Lenguaje de marcado de hipertexto (HTML), Lenguaje estándar universal de marcado (SGML) y Formato de texto enriquecido (RTF) son otros ejemplos de lenguajes de marcado de los que seguramente habrá oído hablar.

NOTA: Como XML es un lenguaje informático universal, se ha acuñado el término "Esperanto para ordenadores" como un modo de referirse a XML. Éste es un buen simil, excepto que el esperanto no suele considerarse un éxito.

Antes de introducirnos en la sintaxis y la gramática de XML, merece la pena examinar los diez objetivos del diseño de XML como los estipularon sus creadores. Estos objetivos se enumeran a continuación y se explican con detalle más adelante.

Algunos de estos objetivos son de naturaleza bastante técnica y se aclararán más tarde en este apéndice, cuando algunos de los términos que mencionan (por ejemplo, definición de tipo de documento) sean explicados. Sin embargo, la mayor parte de estos objetivos proporcionan un importante entendimiento de las pretensiones de XML.

1. XML debe ser fácilmente utilizable en Internet.
2. XML debe admitir una amplia variedad de aplicaciones.
3. XML debe ser compatible con SGML.
4. Debe ser sencillo escribir programas que procesen documentos XML.
5. El número de características opcionales en XML debe mantenerse al mínimo, preferentemente a cero.
6. Los documentos XML deben ser legibles para las personas y razonablemente claros.
7. El diseño de XML debe ser preparado rápidamente.
8. El diseño de XML debe ser formal y conciso.
9. Los documentos XML deben ser fáciles de crear.
10. La concisión del marcado XML es de mínima importancia.

Objetivo 1: XML debe ser fácilmente utilizable en Internet

Este objetivo no quiere decir que los documentos XML deban ser legibles para la actual generación de navegadores. En vez de eso, este objetivo se refiere a una imagen más amplia: tener en cuenta las necesidades de las aplicaciones distribuidas que se ejecutan en un entorno de red a gran escala, como Internet. Los servicios Web cumplen este objetivo. Respecto a los navegadores que admiten XML, Internet Explorer 5.x y posteriores, además de Netscape Navigator 6.x admiten XML.

Objetivo 2: XML debe admitir una amplia variedad de aplicaciones

Este segundo objetivo puede entenderse como una contrapartida del primero. XML está diseñado para funcionar perfectamente en Internet, pero no se limita a Internet. La prueba de que este objetivo se ha conseguido es la gran cantidad de dominios de aplicación existentes fuera de la red, en las que se emplea XML, como publicaciones, intercambio de datos y aplicaciones de base de datos. Además, la rápida aceptación de XML se ha visto facilitada por una proliferación de herramientas: herramientas de autor, filtros sencillos, motores de pantalla, motores de formato y conversores.

Objetivo 3: XML debe ser compatible con SGML

Este objetivo se formuló para que las herramientas SGML pudieran procesar (es decir, analizar) documentos XML. Este objetivo consta de 4 objetivos secundarios:

1. Las herramientas SGML serán capaces de leer y escribir datos XML.
2. Las instancias XML son documentos SGML tal cual, sin cambios en la instancia.
3. Para cualquier documento XML, se puede generar una definición de tipo de documento (DTD) tal que SGML pueda producir "el mismo análisis" que un procesador XML.
4. XML debe tener esencialmente la misma potencia expresiva que SGML.

Aunque este objetivo (y sus objetivos secundarios) garantizan que un documento XML también sea un documento SGML, lo contrario no se produce: un documento SGML NO es un documento XML. Esto es debido a que XML no incluye muchas de las complejas características de SGML.

Objetivo 4: Debe ser sencillo escribir programas que procesen documentos XML

Este objetivo se medía originalmente con la prueba de que un licenciado en informática debería ser capaz de escribir un procesador XML básico en una o dos semanas. A posteriori, este objetivo cuantitativo ha resultado demasiado ambicioso, pero la gran cantidad de procesadores XML disponibles (la mayoría gratuitos) es un claro indicador de que se ha conseguido este objetivo cualitativamente. Sin embargo, la reciente proliferación de estándares relacionados con XML (XML Schema, X-Path, X-Link, etc.) ha hecho que se comente que XML no ha logrado alcanzar este objetivo concreto.

Objetivo 5: El número de características opcionales en XML debe mantenerse al mínimo, preferentemente a cero

Este objetivo se formuló para garantizar que existe una característica coherente entre todos los procesadores XML porque sólo habrá una característica posible que implementar. Por tanto, cada procesador XML existente debe ser capaz de leer todos los documentos XML existentes (siempre que pueda decodificar sus caracteres). SGML, por otra parte, tiene muchas características opcionales en su especificación. En la práctica, esto significa que la posibilidad de intercambiar un documento SGML, creado con un procesador SGML, a otro depende de las características opcionales implementadas en cada procesador.

Objetivo 6: Los documentos XML deben ser legibles para las personas y razonablemente claros

Este objetivo habla por sí mismo y tiene la ventaja de que se puede emplear un editor de texto, incluso uno muy básico como el Bloc de notas, para crear XML funcional.

Objetivo 7: El diseño de XML debe ser preparado rápidamente

Este objetivo se formuló para ganar la carrera por publicar un estándar. Los creadores de XML se dieron cuenta de que si esperaban demasiado, otra organización podría encontrar otro estándar.

Objetivo 8: El diseño de XML debe ser formal y conciso

Este objetivo está muy relacionado con el objetivo de facilitar la programación (el nº 4). Un formato de datos es fácil de usar por el usuario solamente si el programador puede entender fácilmente la especificación. Para conseguirlo, la especificación XML usa una notación empleada por los científicos informáticos cuando describen los lenguajes informáticos: Extended Backus-Naur Form (EBNF).

- EBNF es un conjunto de reglas. llamadas producciones
- Cada regla describe un fragmento específico de sintaxis
- Un documento es válido si puede ser reducido a una sola regla específica, sin ninguna entrada libre, mediante la repetida aplicación de las reglas

Objetivo 9: Los documentos XML deben ser fáciles de crear

Este objetivo amplía los objetivos 4 y 6. Aunque un editor de texto es perfecto para pequeños documentos XML, los documentos grandes se crean más fácilmente usando las herramientas específicas. Este objetivo expresa la intención de diseñar XML para que sea sencillo programar y crear sistemas de edición XML.

Objetivo 10: La concisión del marcado XML es de mínima importancia

Este objetivo indica que, cuando se deba elegir entre la claridad y la concisión, se prefiera la claridad.

Breve lección de HTML

Como HTML es muy parecido a XML, le ofrecemos una breve sinopsis de este lenguaje. Si ya conoce HTML, la curva de aprendizaje de XML será menos pronunciada. (Si no conoce HTML, no se preocupe, le explicaremos todo paso a paso.) Para simplificar la presentación, nuestra explicación de HTML omite algunos detalles (por ejemplo, puede sugerir que algo es necesario cuando en realidad es opcional) y se limita a lo que tiene en común con XML. Por supuesto, ya es consciente de la principal diferencia entre los dos lenguajes: XML es extensible mientras que HTML no lo es (esto se explicará posteriormente).

HTML es el lenguaje usado para describir páginas Web. Una *página Web* es un documento que contiene marcadores especiales, llamados *etiquetas*, que defi-

nen cómo debe ser presentado el contenido en un navegador Web. Un marcador inicial y un marcador final (a partir de ahora los llamaremos etiquetas) rodean al contenido, por ejemplo: <etiqueta>contenido</etiqueta>.

La etiqueta inicial, el contenido y la etiqueta final reciben el nombre de *elementos*. La etiqueta inicial y la etiqueta final están rodeadas por comillas angulares (< y >). La etiqueta final usa la misma palabra contenida en la etiqueta inicial precedida por una barra diagonal (/). De modo que si la etiqueta inicial es , la etiqueta final debe ser . En XML, las etiquetas distinguen entre mayúsculas y minúsculas, de modo que las palabras usadas en las etiquetas inicial y final deben tener los mismos caracteres. Por tanto, en XML, no puede usar (con *f* minúscula) en la etiqueta inicial y (con *F* mayúscula) en la etiqueta final. En HTML, las etiquetas no distinguen entre mayúsculas y minúsculas, de modo que se aceptan etiquetas con diferente uso de mayúsculas y minúsculas.

En HTML, las etiquetas que se pueden usar están predefinidas. Ejemplos de etiquetas HTML son h1 (<h1> y </h1>) para Header 1 y b (y) para negrita. Conocer HTML significa saber cuándo usar cada etiqueta predefinida. Por ejemplo, para que la palabra "Abbreviation" aparezca en negrita en el navegador, escribiría Abbreviation. Cuando el navegador lee esta combinación de etiqueta y contenido, elimina las etiquetas y muestra el contenido en negrita.

Una combinación aleatoria de etiquetas HTML y contenido no suele producir un documento HTML válido. Una página HTML debe tener una cierta estructura. El contenido del documento debe estar entre <html> y </html> y consta de una cabecera y un cuerpo. Cada una de estas secciones está delimitada por etiquetas (llamadas, evidentemente, *cabecera* y *cuerpo*) y tienen contenido, que puede estar rodeado por etiquetas de presentación. El listado A.1 muestra la estructura de un documento HTML. Casualmente, este listado también muestra cómo se incrustan comentarios en una página HTML: <!--EL COMENTARIO SE SITÚA AQUÍ -->

NOTA: Los comentarios se pasan por alto y no afectan a la presentación de la página en el navegador. Sólo se usan para mostrar información al lector humano del código fuente HTML. Algunos comentarios contienen códigos especiales que pueden comprender programas específicos (Por ejemplo el servidor Web), pero esto queda fuera del alcance de esta breve explicación de HTML.

Listado A.1. Estructura de un documento HTML

```
<html >
<head >
```

```

<!--SITUE AQUI EL CONTENIDO DE LA CABECERA -->
</head>

<body>
<!-- SITUE AQUI EL CONTENIDO DEL CUERPO -->
</body>

</html>

```

HTML también consta de un mecanismo para agregar más información a una etiqueta, llamados atributos. Un atributo especifica una propiedad que pertenece a una etiqueta, como el tamaño de una fuente. Por ejemplo para que la palabra "Meaning" aparezca con tamaño 4, debería escribir

```
<font size="4">Meaning</font>
```

Como puede ver en el ejemplo anterior, los atributos se escriben en la etiqueta inicial y hay un espacio de separación entre el nombre de la etiqueta y el nombre del atributo. Éstos toman la forma

```
nombre_atributo=valor_cadena
```

o mostrando el elemento por completo:

```
<etiqueta nombre atributo = valor_cadena >contenido</etiqueta>
```

En HTML, están predefinidos los atributos que puede usar con cada etiqueta, igual que las etiquetas. La etiqueta fuente, por ejemplo, tiene un atributo de tamaño. Los valores de atributo deben estar entre comillas dobles o sencillas (no importa cuál de las dos use, mientras las comillas de apertura sean del mismo tipo que las de cierre). Una etiqueta puede contener más de un atributo. Cada atributo está separado por un espacio. Por ejemplo, quizás quiera especificar el borde, altura y anchura de una tabla, por ejemplo

```
<table border="1" width="255" height="114"> </table>
```

En realidad, HTML también acepta valores de atributo que no están entre comillas. XML, por el contrario, necesita las comillas. Habrá observado una tendencia: XML tiene un conjunto de reglas más estricto que HTML.

El listado A.2 muestra un sencillo documento HTML, mezclando etiquetas (algunas con uno o más atributos) con contenido. En caso de que esté intentando descifrar las etiquetas HTML de este documento, aquí se muestra la creación de una tabla HTML (una tabla HTML tiene el aspecto de una tabla en un procesador de texto). La tabla está encerrada en una etiqueta `<table>`. Cada fila está encerrada en una etiqueta `<tr>` (fila de tabla). En cada fila se crea una celda usando la etiqueta `<td>` (divisor de tabla). El resto del documento HTML se explica por sí mismo. (No se preocupe si no entiende algún detalle al leer este documento HTML. Esta sección es sobre XML, de modo que trata HTML de un modo superficial.)

Listado A.2. Un sencillo documento HTML

```
<html>

<head>
<title>A Glossary in HTML</title>
</head>

<body>

<h1>Glossary</h1>
<div align="left">
  <table border="1" width="359" height="110">
    <tr>
      <td width="125" height="22">
        <b><font size="4">Abbreviation</font></b>
      </td>
      <td width="234" height="22">
        <b><font size="4">Meaning</font></b>
      </td>
    </tr>
    <tr>
      <td width="125" height="22">
        ADO
      </td>
      <td width="234" height="22">
        <b>A</b>ctive <b>D</b>ata <b>O</b>bjects
      </td>
    </tr>
    <tr>
      <td width="125" height="22">
        SOAP
      </td>
      <td width="234" height="22">
        <b>S</b>imple <b>O</b>bject <b>A</b>ccess <b>P</b>rotocol
      </td>
    </tr>
    <tr>
      <td width="125" height="22">
        UDA
      </td>
      <td width="234" height="22">
        <b>U</b>niversal <b>D</b>ata <b>A</b>ccess
      </td>
    </tr>
    <tr>
      <td width="125" height="22">
        XML
      </td>
      <td width="234" height="22">
        <b>X</b>tensible <b>M</b>arkup <b>L</b>anguage
      </td>
    </tr>
  </table>
</div>
```

```
</tr>
</table>
</div>

</body>

</html>
```

XML = HTML con etiquetas definidas por el usuario

Ahora estudiaremos XML y explicaremos los detalles que no se explicaron con anterioridad, de modo que pueda crear su primer documento XML. Un documento XML consta de tres partes: prólogo, cuerpo y epílogo. Sólo es necesario el cuerpo del documento. El prólogo y el epílogo pueden omitirse.

Esta es la estructura básica de un documento XML:

Prólogo:

```
Declaración XML (opcional)
<!-- Situe aquí los comentarios -->
Document Type Declaration (opcional)
<!-- Situe aquí los comentarios -->
```

Cuerpo:

```
Document Element
<Document>
<!-- Situe aquí el documento -->
</Document>
```

Epílogo:

```
<!-- Situe aquí los comentarios -->
```

Un documento XML comienza con un prólogo. Si excluimos los comentarios opcionales, el prólogo contiene dos elementos principales (que también son opcionales). La declaración XML necesita un atributo, que sirve para especificar la versión de la especificación XML a la que se ajusta el documento. La declaración XML también tiene dos atributos opcionales: uno para especificar la codificación de carácter usada y otro para especificar si el documento depende de una definición de tipo de documento (DTD). A continuación tiene un ejemplo de una declaración XML completa que usa los tres atributos.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Los atributos de la declaración XML deben usarse en el orden que aparece en el ejemplo. El atributo de versión es obligatorio y debe tener el valor "1.0". La

codificación de carácter de los documentos XML y las definiciones de tipo de documento se explican más adelante.

El elemento de documento debe estar entre la etiqueta de raíz. En el ejemplo anterior, esta etiqueta de raíz es la etiqueta `<Document>`, pero puede usar cualquier etiqueta para encerrar al elemento de documento. Por último, todas las etiquetas de un documento XML deben anidarse adecuadamente. Si un elemento está contenido en otro elemento recibe el nombre de *secundario* y el elemento contenedor recibe el nombre de *primario*. Aquí tiene un ejemplo:

```
<Book Category="Chess">
  <Title>My System</Title>
  <Author>Aron Nimzowitsch</Author>
</Book>
```

En el ejemplo anterior la etiqueta `<Book>` es primaria para dos secundarias, los elementos `<Author>` y `<Title>`. La anidación correcta requiere que los elementos secundarios estén siempre contenidos en sus elementos primarios. En otras palabras, la etiqueta final de un elemento secundario no puede aparecer después de la etiqueta final de su elemento primario, como en el siguiente ejemplo:

```
<Book>
  <Title>Improper Nesting in XML Explained
</Book>
</Title>
```

El epílogo, que sólo puede contener comentarios (además de espacios en blanco e instrucciones de proceso), suele omitirse. Ya está preparado para un primer acercamiento a un documento XML, como el mostrado en el listado A.3. Los números de línea no forman parte del documento y sólo aparecen para poder hacer una explicación línea a línea posterior más sencilla.

Listado A.3. Sencillo documento XML

```
1: <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2: <!-- Lista de libros sobre XML recomendados -->
3: <!-- Compilado el 17 de Marzo del 17, 2000 por PGB -->
4: <XMLBooks>
5:   <Book ISBN="0-7897-0242-9">
6:     <Title>XML By Example</Title>
7:     <Category>Web Development</Category>
8:     <Author>Benoit Marchal</Author>
9:   </Book>
10:  <Book ISBN="1-861003-11-0">
11:    <Title>Professional XML</Title>
12:    <Category>Internet</Category>
13:    <Category>Internet Programming</Category>
14:    <Category>XML</Category>
15:    <Author>Richard Anderson</Author>
```

```

16:      <Author>Mark Birbeck</Author>
17:      <Author>Michael Kay</Author>
18:      <Author>Steven Livingstone</Author>
19:      <Author>Brian Loesgen</Author>
20:      <Author>Didier Martin</Author>
21:      <Author>Stephen Mohr</Author>
22:      <Author>Nikola Ozu</Author>
23:      <Author>Bruce Peat</Author>
24:      <Author>Jonathan Pinnock</Author>
25:      <Author>Peter Stark</Author>
26:      <Author>Kevin Williams</Author>
27:    </Book>
28:    <Book ISBN="0-7356-0562-9">
29:      <Title>XML in Action</Title>
30:      <Category>Internet</Category>
31:      <Category>XML</Category>
32:      <Author>William J. Pardy</Author>
33:    </Book>
34:  </XMLBooks>

```

La línea 1 del listado A.3 contiene una declaración completa XML que incluye los tres atributos. Las líneas 2 y 3 son comentarios usados en este caso para indicar la función de este documento. A continuación está el cuerpo del documento XML, empezando por la línea 4 y terminando en la línea 34. Este documento no tiene epílogo, como suele ser habitual. El elemento de documento está entre la etiqueta `<XMLBooks>` (la etiqueta de inicio está en la línea 4, la etiqueta final en la línea 34). El elemento de documento tiene tres secundarios, cada uno encerrado entre una etiqueta `<Book>`. El secundario 1 empieza en la línea 5 y finaliza en la línea 9. El secundario 2 empieza en la línea 10 y finaliza en la línea 27. El secundario 3 empieza en la línea 28 y finaliza en la línea 33. Cada elemento `<Book>` tiene un atributo ISBN y una cantidad de secundarios: uno `<Title>`, uno o más `<Category>` y uno o más `<Author>`. Aquí puede apreciarse una ventaja significativa de XML sobre los tradicionales archivos de texto: XML está bien preparado para tratar con las estructuras primario/secundario.

Como creador del documento, merece la pena señalar que yo inventé las etiquetas y los atributos usados en este documento (XMLBooks, Book, ISBN, Title, Category, Author). Por ejemplo, otro autor podrá haber preferido usar `<ShelvingCategory>` en lugar de `<Category>`. También puede hacerlo usted mismo si esta especificación XML llega a alcanzar el objetivo número 6 (los documentos XML deben ser legibles para las personas y razonablemente claros).

Definiciones de tipo de documento

El documento XML mostrado en el listado A.3 tiene una estructura más definida que la estructura impuesta por XML. Una definición de tipo de documento

(DTD) proporciona un modo de especificar esta estructura. el modelo de datos correspondiente al modelo de datos. Se puede realizar la comparación con un esquema de base de datos que defina el modelo de datos de una base de datos. Esta comparación funciona perfectamente porque tanto una base de datos como un documento XML contienen datos estructurados. La DTD es el esquema correspondiente al documento XML. El listado A.4 muestra la DTD correspondiente al listado A.3.

Listado A.4. Esquema DTD correspondiente al documento XML

```
<?xml version="1.0"?>
<!-- El elemento superior,
XMLBooks, es una lista de libros -->
<!ELEMENT XMLBooks (Book+)>

<!-- Un elemento Book
contiene 1 Title, 1 o mas Category,
y 1 o mas Author -->
<!ELEMENT Book (Title,Category*,Author+)>

<!-- Un Book tiene 1 atributo requerido -->
<!ATTLIST Book ISBN ID #REQUIRED>

<!-- Los elementos Title, Category, y Author
contienen texto -->
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Category (#PCDATA)>
<!ELEMENT Author (#PCDATA)>
```

La estructura de la DTD es muy parecida a la de Extended Backus-Naur Form mencionada anteriormente. La DTD es un conjunto de reglas sucesivas que describen cómo ensamblar los datos en el modelo de documentos XML. Cada regla describe un elemento específico o un atributo que el modelo puede contener. Un documento XML es válido si puede reducirse a una sola regla específica de la DTD, sin ninguna entrada libre, mediante la repetida aplicación de las reglas.

A continuación tiene una descripción de la sintaxis usada en esta DTD. Observe que la DTD usa una sintaxis diferente de la de los documentos XML.

Cada elemento se describe usando una línea de descripción de elemento.

```
<!ELEMENT element_name (element_content)>
```

`element_name` usa la etiqueta para identificar cada elemento. En `element_content`, debe colocar otros elementos o `#PCDATA` para indicar que el elemento contiene texto. Los elementos hoja son elementos que no tienen secundarios. Estos elementos suelen especificarse como contenedores de `#PCDATA`.

Los caracteres especiales tras un nombre de elemento indican la cardinalidad de los elementos contenidos. La cardinalidad indica cuántos de estos elementos

pueden existir y si el elemento es opcional o necesario. Hay cuatro modos de indicar cardinalidad.

- Un elemento contenido sin ningún símbolo especial (como `Title` en el listado A.4) debe aparecer exactamente una vez en el elemento que se define (cardinalidad: 1).
- Un elemento contenido seguido por un signo de interrogación (?) es opcional y sólo puede aparecer una vez en el elemento (cardinalidad: 0..1).

Los siguientes dos modos definen elementos repetidos, uno para los requeridos y otro para los opcionales.

- Un elemento contenido seguido por un signo de adición (+) (como `Book` y `Author` en el listado A.4) es necesario y puede aparecer repetido (cardinalidad: 1..N).
- Un elemento contenido seguido por un asterisco (*) (como `Category` en el listado A.4) es opcional y puede aparecer repetido (cardinalidad: 0..N).

```
<!ATTLIST element_name attribute_name attribute_content  
optionality>
```

Las listas de atributo se definen en una línea separada. `element_name` es de nuevo la etiqueta a la que pertenece el atributo. `attribute_name` es el nombre del atributo (Por ejemplo, `ISBN` en el listado A.4). El contenido del atributo se define usando una serie de palabras clave. La más común es `CDATA` que indica que el atributo recibe datos de carácter. La opcionalidad se indica mediante la palabra clave `#REQUIRED` para los atributos necesarios y `#IMPLIED` para los atributos opcionales.

Esquemas XML

El 2 de Mayo del 2001, el consejo encargado de controlar los estándares XML anunció que un importante miembro de la familia XML había alcanzado el estatus de estándar (una recomendación propuesta, como `www.w3.org` lo llama). Este estándar recibe el nombre de Esquemas XML y está destinado a reemplazar a la DTD como el sistema preferido de validar documentos XML.

Esquemas XML ofrece dos ventajas evidentes sobre la DTD:

- Un esquema XML es un documento XML
- Los esquemas XML permite especificar las características de datos (como tipo, tamaño y precisión) de los elementos y atributos

Un documento de esquema tiene este aspecto:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- Situe aquí el contenido del esquema-->

</xsd:schema>

```

El atributo `xmlns:xsd` del elemento de esquema es una declaración de nombre de espacio, que estudiaremos en la siguiente sección. Observe que el valor de este atributo ha cambiado a lo largo del tiempo, de modo que si se encuentra con un esquema con un valor diferente en este atributo (por ejemplo, `www.w3.org/2000/10/XMLSchema`), ese esquema fue creado correctamente de acuerdo con una versión de borrador del estándar Esquema XML.

El contenido del esquema consta de definiciones para los elementos y atributos que el esquema puede contener. Un elemento se define como se indica a continuación

```

<xsd:element name="theElementName">
<!--Situe aquí los detalles específicos del elemento-->
</xsd:element>

```

y un atributo se definen como se indica a continuación

```

<xsd:attribute name="theAttributeName">
<!--Situe aquí los detalles específicos del atributo-->
</xsd:attribute>

```

Puede agregar documentación con comentarios o insertar un elemento de anotación dentro del elemento o definición de atributo. El elemento de anotación contiene un elemento de documentación en el que puede explicar los detalles específicos del elemento o del atributo.

```

<xsd:annotation>
<xsd:documentation>Some explanation here...</xsd:documentation>
</xsd:annotation>

```

Puede agrupar los elementos y los atributos insertándolos en una etiqueta `complexType`.

```

<xsd:complexType>
</xsd:complexType>

```

Este tipo de agrupación es necesario cada vez que se encuentra con una definición de elemento como la siguiente:

```

<ELEMENT Book (Title,Category*,Author+)>

```

Los elementos agrupados en una secuencia deben ser mostrados en el orden en el que están definidos.

```
<xsd:sequence>
</xsd:sequence>
```

Así, si define un elemento `Book` como se indica a continuación, el elemento `Book` debe contener los elementos `Title`, `Category` y `Author` exactamente en este orden (por ejemplo, `Title`, `Author` y `Category` no sería válido).

```
<xsd:element name="Book">
xsd:complexType>
<xsd:sequence>
<xsd:element name="Title">
</xsd:element>
<xsd:element name="Category"/>
<xsd:element name="Author"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Se considera que la cardinalidad de los elementos es uno. Si quiere crear un elemento repetidor, puede hacerlo agregando el atributo `maxOccurs="unbounded"` a la definición del elemento. Si quiere crear un elemento opcional, puede hacerlo agregando el atributo `minOccurs="0"` a la definición del elemento. Por supuesto, puede combinar estos atributos para crear un elemento repetidor opcional.

Finalmente, puede especificar el tipo de datos de un elemento con el atributo `type="xsd:datatype"`. En nuestro ejemplo, sólo usamos el tipo de datos de cadena. El esquema XML permite una gran variedad de tipos de datos, como entero, largo, fecha, hora, doble, flotante, etc. El listado A.5 enumera el esquema XML correspondiente a la DTD mencionada anteriormente. La extensión de archivo de esquemas XML es `.xsd` y por eso a veces se les llama XSD.

Listado A.5. Esquema XML correspondiente a DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Esquema W3C para una lista de libros -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="XMLBooks">
    <xsd:annotation>
      <xsd:documentation>The top-level element,
XMLBooks, is a list of books.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>A Book element contains 1
Title, 1 or more Category, and 1 or more Author.</
xsd:documentation>
          </xsd:annotation>
        </xsd:complexType>
```

```

        <xsd:sequence>
            <xsd:element name="Title" type="xsd:string">
                <xsd:annotation>
                    <xsd:documentation>The Title, Category,
and Author elements contain text.</xsd:documentation>
                </xsd:annotation>
            </xsd:element>
            <xsd:element name="Category"
type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="Author" type="xsd:string"
maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="ISBN" type="xsd:string"
use="required" id="isbn">
            <xsd:annotation>
                <xsd:documentation>A Book has 1 required
attribute.</xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

El listado A.6 muestra cómo un documento XML puede hacer referencia a su esquema XML asociado.

Listado A.6. Documento XML que hace referencia a su esquema XML asociado

```

<?xml version="1.0" encoding="UTF-8"?>
<XMLBooks xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="./Books.xsd">
    <Book ISBN="0-7897-2242-9">
        <Title>XML By Example</Title>
        <Category>Web Development</Category>
        <Author>Benoit Marchal</Author>
    </Book>
    <Book ISBN="0-7356-0562-9">
        <Title>XML in Action</Title>
        <Category>Internet</Category>
        <Category>XML</Category>
        <Author>William J. Pardy</Author>
    </Book>
</XMLBooks>

```

Espacios de nombre XML

La extensibilidad de XML es, a la vez, una bendición y una maldición. Al permitir a cualquier persona crear sus propias etiquetas se corre el riesgo de crear

una nueva torre de Babel. Afortunadamente, los programadores de los estándares XML percibieron el peligro e idearon una solución. Llamada espacios de nombre. Ya conoce el concepto de los espacios de nombres gracias a su estudio de C# (el mismo concepto aparece en C++, Java y otros lenguajes .NET). La implementación varía un poco entre un caso y otro, pero la idea es siempre la misma.

Se asocia un nombre único a un prefijo y se usa este prefijo para calificar los nombres que podrían colisionar sin el prefijo. Como XML está basado en Web, los diseñadores decidieron usar URL como los nombres únicos.

El espacio de nombres usado en un esquema XML se especifica agregando el atributo `targetNamespace="www.myurl.com"` al esquema. Este espacio de nombres se define agregando un atributo especial `xmlns` al elemento de esquema. Puede anexar el prefijo de espacio de nombres usando dos puntos para separar el atributo `xmlns` de `prefix`. El diseñador del esquema debe asegurarse de que el valor de este atributo es único. Esto suele conseguirse usando la URL de la compañía.

```
xmlns:prefix="http://www.myurl.com"
```

Tras definir un prefijo de espacio de nombres, debe anexarlo a todos los elementos contenidos en el espacio de nombres.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Esquema W3C para una lista de libros -->
<xsd:schema
  targetNamespace="www.myurl.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:book="www.myurl.com">
  <xsd:element name="XMLBooks">
    <xsd:annotation>
      <xsd:documentation>The top-level element,
XMLBooks, is a list of books.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>A Book element contains 1
Title, 1 or more Category, and 1 or more Author.</
xsd:documentation>
          </xsd:annotation>
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Title" type="xsd:string">
                <xsd:annotation>
                  <xsd:documentation>The Title, Category,
and Author elements contain text.</xsd:documentation>
                </xsd:annotation>
              </xsd:element>
              <xsd:element name="Category"
type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
```



```

        <xsd:element name="Author" type="xsd:string"
maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="ISBN" type="xsd:string"
use="required" id="isbn">
        <xsd:annotation>
            <xsd:documentation>A Book has 1 required
attribute.</xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

El siguiente documento XML muestra cómo crear un documento XML que haga referencia a un esquema usando nombres de espacio. Esto se consigue agregando tres atributos al elemento de raíz. El primer atributo define el prefijo usado por el espacio de nombres y la cadena única asociada a este espacio de nombres. El segundo atributo especifica qué versión del esquema XML se está usando. Por último, el tercer atributo le indica qué espacio de nombres está usando el esquema XML y dónde está ubicado el esquema XML.

```

<?xml version="1.0" encoding="UTF-8"?>
<book:XMLBooks
  xmlns:book="www.myurl.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="www.myurl.com .\Books.xsd">
  <Book ISBN="0-7897-2242-9">
    <Title>XML By Example</Title>
    <Category>Web Development</Category>
    <Author>Benoit Marchal</Author>
  </Book>
  <Book ISBN="0-7356-0562-9">
    <Title>XML in Action</Title>
    <Category>Internet</Category>
    <Category>XML</Category>
    <Author>William J. Pardy</Author>
  </Book>
</book:XMLBooks>

```

Como la mayoría de los elementos de un documento XML pertenecen al mismo espacio de nombres, se puede crear un espacio de nombres predeterminado y omitir el prefijo del espacio de nombres, por ejemplo, `xmlns="www.myurl.com"`. Para terminar, se pueden incluir varias declaraciones de espacios de nombres en el mismo documento XML. Esto se consigue agregando todos los atributos de espacio de nombres al elemento de raíz. No obstante tenga en cuenta que un documento sólo puede apuntar a un esquema XML.